FACULTY OF ENGINEERING
AND ARCHITECTURE

Federated Learning in Network Intrusion Detection

Bram Van de Walle
Student number: 01707633

Supervisors: Prof. dr. Bruno Volckaert, Dr. ir. Tim Wauters
Counsellors: Laurens D'hooge, Prof. dr. Bruno Volckaert, Dr. ir. Tim Wauters

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Information Engineering Technology

Academic year 2020-2021

GHENT
UNIVERSITY

Acknowledgements

First and foremost, I would like to thank Laurens D'hooge for assisting me throughout all steps in the creation of this master's thesis. His feedback and assistance was vital in order for me to come to the thesis as is.

Secondly, I would like to thank my father, Rik Van de Walle, who has thoroughly read my thesis and has given important feedback. In addition, I would like to thank my father, my mother, Mieke Van Hyfte, my stepfather Peter De Bruyne and my stepmother Brenda Delcloo for always taking good care for me, for supporting me in everything I do and want to accomplish and for guiding me where needed. Especially in the past few months for keeping me motivated to do my absolute best in order to finish my studies in beauty with this master's thesis. In addition to my parents, also my grandparents Valère Van de Walle and Daisy De Meester deserve to be thanked because they have been welcoming me almost every single week with open arms in the last years. They too played an important role in giving me moral support in the last few months finishing my studies.

Also, thanks to Prof. dr. Bruno Volckaert and Dr. ir. Tim Wauters for giving me the opportunity to do research about federated learning in network intrusion detection.

This is, of course, not an exhaustive list of people that deserve to be thanked. Therefore, also a big thank you to all who supported me in the last few months.

Bram Van de Walle

Federated Machine Learning in Network Intrusion Detection

Bram Van de Walle

Supervisor(s): Laurens D'hooge, Prof. dr. Bruno Volckaert, Dr. ir. Tim Wauters

Abstract— **This thesis has conducted research to the use of federated learning in network intrusion detection. Network intrusion detection systems monitor the network traffic and try to detect attacks if they occur. Such intrusion detection systems (IDSs) can use machine learning models that classify network traffic flows captured by the IDSs as benign or malicious. Such machine learning models require datasets to be trained with. Organizational networks that consist of two (or more) geographically separated networks, in which each network has its own IDS that uses machine learning models, must have local training datasets for each of the IDSs. If, over time, one of the IDSs learns to recognize a new attack type, it would be useful that the other IDSs could learn from this IDS. A simple solution is to share the new training dataset with all IDSs. However, this may not always be feasible due to privacy/security policies, because sending the training sets requires too much network bandwidth, or because it is computationally too expensive to retrain all models in the IDSs. This master's thesis found that federated learning is a possible solution that allows for multiple IDSs to learn from each other during training. Seven different experiments give insights in why and how using federated learning in network intrusion detection systems can be useful. In these experiments, the CIDDS-001 datasets are used that provide captured network traffic flows of both benign and malicious flows. The framework that is used for performing the federated learning experiments is TensorFlow Federated.**

Keywords— **Federated Learning, Machine Learning, Random Forest Classifier, Multi-Layer Perceptron, Network Intrusion Detection, TensorFlow Federated, CIDDS-001**

I. Introduction

Almost all organizations today have information technology (IT) as a vital part of their business model. Since companies may store sensitive information in their IT infrastructure, IT has become a coveted target for hackers. As been proven multiple times by the damage that can be done by attackers hacking their way into corporate IT infrastructure, companies must try to limit the chances of hackers being able to break into the companies' IT infrastructure. Possible means of doing so are by imposing user authentication before employees can gain access to IT infrastructure, firewall rules to impose network policies or installing anti-virus software on the computers of the employees. However, none of these solutions are bulletproof because of the risk of human error. Hackers can use social engineering approaches to gain information about employees or even obtain user credentials. Even more worrying is that the flaws that attackers exploit, may not even come from a mistake made by any of the employees of the company itself. There could be flaws in any of the software that the company uses. Even software that one might think of as being secure and reliable, can contain flaws. Recently, on two separate occasions, hackers were able to use flaws in widely used software and breach into tens of thousands of governmental and private computer networks. The first started in March 2020 when Russian hackers were able to compromise an update of SolarWinds' network management software Orion and install malware in organizations and governing bodies that were updating their SolarWinds Orion software. The scope of this breach was about 18,000 networks [1], [2]. More recently, another very serious attack occurred. Early March 2021, Chinese hackers exploited vulnerabilities in Microsoft Exchange and were able to hack into hundreds of thousands of organizations worldwide [3].

Hence, it is important that companies invest in trying to avoid letting hackers break into their IT infrastructure. However, since human errors cannot be avoided at all times, prevention alone does not suffice. What if an attacker did manage to break into the infrastructure? Will this ever be discovered? If not, nobody will ever be able to handle the breach. To make sure that an intrusion can be dealt with, it must be detected. To lower the damage done and thereby the costs of the attack, the detection must happen quickly. This is where Intrusion Detection Systems come in handy.

Intrusion Detection Systems (IDS) are hardware or software applications that deal with the task of monitoring systems or networks with the purpose of detecting malicious activities [4], [5]. Once malicious activity has been detected, the IDS must generate alarms/notifications so that the intrusion can be dealt with appropriately. Either by a human operator or by an automated response. There are different types of IDSs: they can be host-based or network-based, signature-based or anomaly-based, stateless or stateful, centralized or distributed [6]. In any case, the goal of an IDS is to detect attacks as soon as possible in order to limit the damage done to the IT infrastructure of the company and to the company itself.

II. Goals of the thesis

In this thesis, research is conducted in the field of network-based/anomaly-based intrusion detection systems where the IDS uses a machine learning model to classify monitored traffic as benign or malicious. These machine learning models must be trained in order to be able to correctly classify flows of network traffic. Therefore, an IDS must contain a local training dataset on which its machine learning model can be trained. Such a training set contains flows that are similar to what the IDS will monitor when used in real life scenarios. The starting point of this thesis is an organization that has two geographically separated networks. In each of the two networks, an IDS is placed that monitors the network traffic right before it enters the Local Area Network (LAN). An example of such a network setup is depicted in Figure 1.

As mentioned, each IDS has its own local training dataset. The training dataset can, for example, consist of labeled net-

B. Van de Walle, student Master of Science in Information Engineering Technology, Ghent University (UGent), Gent, Belgium. E-mail: Bram.VandeWalle@UGent.be .

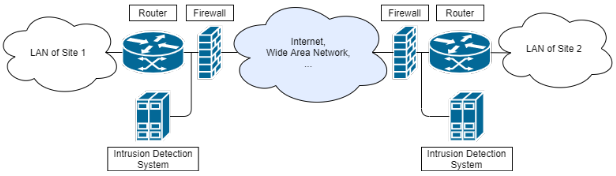


Fig. 1. Example network setup of organization with two, geographically separated, sites

work flows that the IDS has captured over time. How the creation of such a local training set can be done, is out of the scope of this thesis. However, it is possible that the local dataset of one IDS contains flows of an attack type that the other IDS does not have and vice versa. The consequence of this is that one IDS can detect certain attacks that the other cannot because the latter will not learn to recognize the attack type during the training phase. A straightforward solution is to exchange the local datasets between the two IDSs. However, this may not always be possible since the local training datasets contain captured network flows that possibly contain privacy sensitive data and thus certain policies (e.g. GDPR or corporate privacy/security policies) may prevent the IDSs to share their local datasets. Another option is to exchange the internal parameters of the machine learning models that are used in the IDSs that classify the monitored network traffic. If the same model is used in both IDSs and during training, both models exchange and aggregate their internal parameters, maybe the two models can learn from each other. The concept of exchanging the internal parameters and aggregating them is called federated learning. The goal of this thesis is to find an answer to the following question: “Can two network intrusion detection systems that each have their own local training datasets learn from each other when one training dataset lacks captured flows from a certain attack type while the other training dataset does not?”

In order to be able to answer the aforementioned question, seven experiments are executed where each experiment is based on the one before or is introduced because the results of the previous experiment led to new questions. Before the experiments are discussed, section III gives an overview of the related work and section IV gives some required knowledge to be able to follow the discussion of the experiments. Finally, in section V the experiments, the results and the conclusions are discussed.

III. Related Work

A. Coburg Intrusion Detection Data Sets

As mentioned in the introduction, network intrusion detection systems that internally use machine learning models to classify the network traffic flows they monitor, require a training dataset to be trained with. Throughout the experiments in this thesis, the datasets that represent the local training datasets of the two IDSs depicted in Figure 1 will come from the Coburg Intrusion Detection Data Sets (CIDDS). More specifically, the CIDDS-001 datasets will be used. The researchers of the CIDDS-001 project emulated the network of a small business environment with the open source cloud computing infrastructure OpenStack. They used Python scripts in combination with Linux tools like `nmap` to generate benign and malicious traffic [7]. Inside the emulated network, the network traffic flows are monitored at two places: at an internal router and at an external server. Capturing these flows is done with Cisco’s network monitoring protocol NetFlow [8]. The researchers used 11 different NetFlow attributes which are stored in the CIDDS-001 datasets: *duration of the flow*, *start timestamp of the flow*, *number of transmitted bytes*, etc. Next to these 11 features, each flow is labeled with 4 attributes: *Class*, *AttackType*, *AttackID* and *AttackDescription*. The *class* indicates whether a flow is “normal”, “attacker”, “victim”, “suspicious” or “unknown”. The *attack type* indicates which attack was executed: “bruteForce”, “dos”, “pingScan”, “portScan” or “—” for normal (i.e. benign traffic) flows. The label *attack id* assigns a unique ID to each attack and each flow of that attack gets assigned this attack id. Finally, the *attack description* provides additional information about the executed attack corresponding to the flow. Throughout a period of four weeks, the network traffic was generated and captured [9].

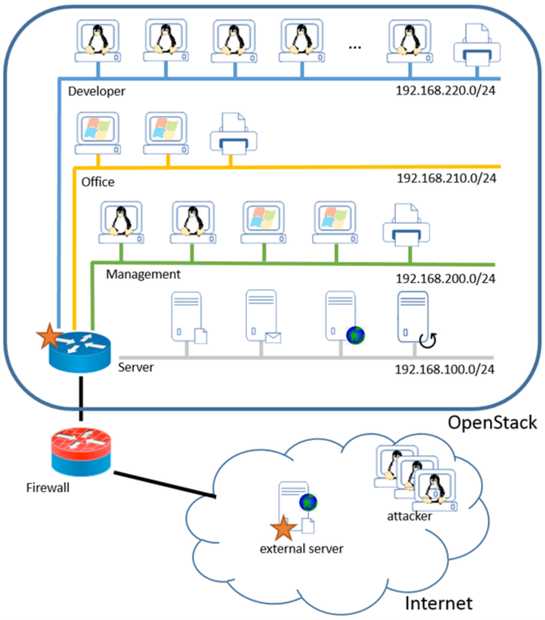


Fig. 2. Emulated small business network environment (*source:* [7])

B. Evaluation of existing machine learning based network intrusion detection algorithms

In this thesis, anomaly-based network intrusion detection systems are considered that use machine learning models to classify the network traffic flows it monitors. In [10], Abhishek Verma and Virender Ranga have evaluated multiple existing machine learning models using the CIDDS-001 datasets. Both supervised (i.e. the k-Nearest Neighbors (KNN), Support Vector Machine (SVM), decision tree (DT), random forrest classifier (RFC), artificial neural network (ANN), Naive Bayes (NB) and deep learning (DL) models) and unsupervised (i.e k-means clustering (KMC), expectation maximization clustering (EMC) and self-organizing maps (SOM) models) machine learning algorithms are evaluated. Most of the algorithms that are learned in a supervised manner score very good in the evaluation. KNN, SVM, DT, RFC and DL all have an accuracy of more than 94% on the CIDDS-001 data captured at the external server, and more than 99% accuracy on the CIDDS-001 data captured internally in the OpenStack network. The unsupervised machine learning models KMC, EMC and SOM have a much lower score. All scores are below 50%, with the exception of KMC for the flows captured at the internal router (where the accuracy is 99.6%). Therefore, only supervised machine learning algorithms will be used in this thesis.

C. Federated Learning

In 2017, Google proposed a concept called *federated learning* in [11] as an answer to problems with machine learning models that have to learn on huge datasets and/or datasets containing privacy sensitive data. A lot of people have smartphones these days which contain useful information that can be used to enhance the user experience. For example, Google could collect text written by people to enhance their text entry models. Traditionally, such models were trained on centralized data, but text written by smartphone users should not be centralized for privacy reasons. An alternative is to let the smartphones learn a model on their own local data, centralize the locally computed updates to a central server which in turn aggregates these locally computed updates into a global model. This global model is then redistributed among the participating smartphones. The process of locally computing updates, centralizing and then updating the updates can be done repeatedly.

In [12], S. Rahman et al. further explore federated learning by comparing it to an on-device learning approach called *SecProb* and the centralized approach. The centralized approach is considered to be the best case scenario because no compromises are made with respect to training a model: all local training data are available for training the model. S. Rahman et al. found that, after enough training rounds, their federated learning model outperforms the SecProbe model and approaches the accuracy reached by the centralized model.

IV. Experimental Prerequisites

This section discusses how the CIDDS-001 datasets are used in the experiments and how they are preprocessed. Also, the algorithmic choices are explained, while the next section discusses the actual experiments and the conclusions thereof.

A. Content and use of the CIDDS-001 datasets

It was already mentioned that two IDSs are considered for the experiments. Each of the IDSs will be a federated learning client and thus requires a local training dataset. The traffic that is captured inside the CIDDS-001 datasets was captured over a time span of four weeks. For each week, a dataset is provided for both the internal captured flows and the external captured flows, totalling eight datasets. However, only the datasets of the first two weeks with flows captured internally contain flows from all attack types “port scan”, “ping scan”, “DoS” and “brute force”. Therefore, these two datasets, *internal week 1* and *internal week 2*, will be used in the experiments where each dataset represents the local dataset available for each IDS shown in Figure 1.

The goal of classification models is that given a set of features (also known as the feature vector), the model can predict the correct class/label corresponding to the feature vector. With the CIDDS-001 datasets, the feature vector consists of the captured NetFlow attributes and the class/label that a model has to predict is one of the label attributes “Class”, “AttackType”, “AttackID” or “AttackDescription”. The last two label attributes are merely metadata about the attacks that will never be available in real life scenarios. In some papers (e.g. [10]) the classification label used is the attribute “Class”. Although it can be useful to know whether a network flow is either “normal”, “attacker”, “victim”, “suspicious” or “unknown”, if a model can predict the specific attack type in case of malicious flows, the IDS can be more specific in the alerts it generates or can take specific actions to prevent further damage. Therefore, in the experiments in this thesis, the attribute that is chosen to be the classification label, is the attribute “AttackType”.

B. Preprocessing the CIDDS-001 datasets

Before the CIDDS-001 datasets *internal week 1* and *internal week 2* can be used in the experiments, some preprocessing steps need to executed. The first four preprocessing steps are executed in all experiments while the fifth preprocessing step is executed depending on which model is used in the experiments.

B.1 Remove abnormalities

There are two abnormalities in the CIDDS-001 datasets. The first abnormality is the appearance of an extra column *Flows* inside the datasets for which no documentation is provided in [7] or [9]. The only value in this column is a “1”, making this attribute a useless feature for the classification models. Therefore, this column is dropped. The second abnormality is found in the column *Bytes* (which specifies the number of bytes transmitted in a flow). In flows for which the number of transmitted bytes exceeds one million, the value is encoded with an “M” to denote the million. For example, if the number of transmitted bytes is 6,000,000 the value for the attribute *Bytes* will be “6 M”. This encoding is reversed so that all values in the column *Bytes* are integers.

B.2 One-hot encode categorical attributes

Next, the categorical attributes in the CIDDS-001 datasets are one-hot encoded. That means that each of the possible values in a categorical attribute, becomes a binary column. The value of the resulting column indicates whether the value corresponding to the column applies or not. The only categorical attribute in the CIDDS-001 dataset is the attribute *Proto* which indicates the protocol used in the captured flow: “ICMP”, “IGMP”, “TCP”, or “UDP”.

B.3 Unfold OR-concatenated attributes

The following step in the preprocessing phase is unfolding the OR-concatenated values in the column *Flags*. If the protocol of a flow is “TCP”, the TCP flags URG, ACK, PSH, RST, SYN and FIN of the flow are captured in the column *Flags* using a six-character string. Each dot in the string `"......"` is replaced with “U”, “A”, “P”, “R”, “S” and/or “F” respectively, depending on whether or not the corresponding TCP flag was set. If the protocol of the flow is not “TCP”, the string `"......"` is used as is. In this preprocessing step, this column is unfolded into six separate binary columns indicating the use of the TCP flag.

B.4 Drop some columns for classification

Some of the attributes that are available in the CIDDS-001 datasets have to be removed prior to training because they are highly correlated with the target, yet contribute no true improvement to model generalization. Most often these features fall under sample metadata. The columns that are dropped in this preprocessing step are: *Date first seen*, *Src IP Addr*, *Dst IP Addr*, *Class*, *AttackID* and *AttackDescrption*.

B.5 Normalize non-binary columns

The final step in the preprocessing phase is normalizing the non-binary columns. However, normalization is not always required. Classification algorithms that use decision trees (such as the random forest classifier) do not require the data to be normalized because each node in a decision tree partitions the data in two groups based on one feature. Whether the data is normalized or not, has no influence on how the data is partitioned. The parameter that splits the data would be scaled just as the data is scaled during normalization. Classification algorithms that execute calculations using multiple features at once, such as the multi-layer perceptron (MLP) neural network, do require the data to be normalized since using multiple features at once may result in certain features to have a bigger weight on the classification model than other features. Thus, in experiments where the MLP model is used, the non-binary features are normalized with either min-max normalization or z-score normalization.

C. Classification models

Throughout the experiments, two different machine learning models are used as classifiers. The first two experiments start using the random forest classifier (RFC) since this model showed great results on the CIDDS-001 data in [10] as discussed in III. However, in later experiments, TensorFlow Federated's implementation of federated learning is used to execute the federated learning experiments and TensorFlow Federated does not support the use of RFC models. Therefore, the multi-layer perceptron neural network as proposed by [13] is evaluated in experiments 3 and 4. As the results of the MLP model approaches the results of RFC, it is further used in the federated learning experiments 5, 6 and 7.

V. Experiments & Conclusions

The goal of this thesis was to find an answer to the question: "Can two network intrusion detection systems learn from each other using federated learning if the network flow classification models behind each of the IDSs have a local training dataset in which flows of one attack type are not available while the other local training dataset does have flows of that attack type available?" The answer to this question will be discussed throughout this section.

Experiment 1 is a simple classification problem using RFC models on the CIDDS-001 datasets *internal week 1* and *internal week 2*. The datasets are preprocessed as discussed in IV-B. Afterwards they are balanced out. In *internal week 1*, 1,626 flows are randomly selected from each attack type. In *internal week 2*, 2,731 flows are randomly selected from each attack type. These two datasets are split into two training and testing datasets. The first RFC model *rfc_week1* is trained with the training dataset coming from *internal week 1*. Analogously for *rfc_week2* and *internal week 2*. Then each RFC model is tested twice. First with the test sets and a second time with the full dataset (i.e. the datasets before *internal week 1* and *internal week 2* are balanced out) coming from "the other week". The results shown in Tables I and II show that the RFC models are very suitable to classify the CIDDS-001 network flows with respect to the classification label "attack type".

TABLE I
PRECISION SCORES OF TRAINED MODEL "RFC_WEEK1" IN EXPERIMENT 1 (ALL VALUES IN PERCENTAGES)

Attack type	Results of **rfc_week1**	
	Testing set week 1	Full dataset week 2
Normal	99.40	99.72
BruteForce	94.21	86.04
DoS	100.0	99.96
PingScan	92.83	90.85
PortScan	96.94	92.60

TABLE II
PRECISION SCORES OF TRAINED MODEL "RFC_WEEK2" IN EXPERIMENT 1 (ALL VALUES IN PERCENTAGES)

Attack type	Results of **rfc_week2**	
	Testing set week 2	Full dataset week 1
Normal	98.75	98.93
BruteForce	100.0	96.37
DoS	100.0	99.98
PingScan	91.07	96.58
PortScan	91.29	95.60

The first experiment is followed by the so-called "base experiment". This experiment is very reminiscent of experiment 1. However, all "brute force" network flows are removed from the training dataset of *rfc_week1* and 3,359 flows of the other attack types are randomly chosen. Analogously, all "ping scan" network flows are removed from the training dataset of *rfc_week2* and 3,366 flows of the other attack types are randomly chosen. The trained models are tested in a similar way as seen in experiment 1. The results can be found in Tables III and IV. Without surprise, once the models are trained and tested with a test set that contains flows of all attack types, the models cannot correctly predict the flows of which the attack type was not available in their training dataset.

The base experiment serves as a motivation for why federated learning may be useful in the context of network intrusion detection. The base experiment shows that if an organization has two geographically separated networks where each network has its own IDS in which the classification model is trained with a local training dataset, it is possible that one IDS can detect attacks that the other cannot and vice versa. This is unfortunate since the organization would benefit from both IDSs being able to detect all attacks. Sharing the local training dataset could be the solution, but may be disallowed due to certain policies (pri-

TABLE III
PRECISION SCORES OF TRAINED MODEL "RFC_WEEK1" IN EXPERIMENT 2 (ALL VALUES IN PERCENTAGES)

Attack type	Results of **rfc_week1**	
	Testing set week 1	Full dataset week 2
Normal	100.0	99.90
BruteForce	n/a	0.000
DoS	99.96	99.92
PingScan	97.02	90.92
PortScan	95.73	92.63

TABLE IV
PRECISION SCORES OF TRAINED MODEL "RFC_WEEK2" IN EXPERIMENT 2 (ALL VALUES IN PERCENTAGES)

Attack type	Results of **rfc_week2**	
	Testing set week 2	Full dataset week 1
Normal	98.53	98.89
BruteForce	99.89	96.25
DoS	100.0	99.97
PingScan	n/a	0.000
PortScan	95.83	98.22

vacy/security policies) since the training datasets may contain sensitive data, as the training datasets consist of captured network traffic. On top of that, sending the training datasets over the network and having to retrain the models can be too costly. However, the parameters of the classification models of each client can be exchanged and may be aggregated using federated averaging.

For the federated learning experiments 5, 6 and 7 (cfr. infra), the framework TensorFlow Federated is used. Unfortunately, TensorFlow Federated does not support RFC models to be used in federated learning. Therefore, a new classification model is introduced in experiment 3. The model that is used is a multi-layer perceptron neural network of which the configuration is based on the MLP presented in N. Oliveira et al.'s paper [13]. The same experiment that N. Oliveira et al. executed in their paper (which, to be clear, does not make use of federated learning) is executed in experiment 3. Their approach leads to creating a preprocessed dataset that is subject to great imbalance. Hence, the prediction scores of the attack types "brute force" and "ping scan" are very poor while the prediction scores of "normal", "DoS" and "port scan" are very good as shown in Table V.

Since the results of the third experiment that introduces the MLP model is subject to the negative effects of training a model with imbalanced data, the MLP model is reused in the forth experiment in which it is trained with a balanced dataset. Now, the results are good for all attack types and they are similar to the results of the RFC models used in the first experiments as shown in Table V.

Knowing all of this, it is time to start experimenting with federated learning. Experiment 5 tries to identify whether or not an MLP model that is trained using federated learning can reach competitive prediction scores when compared to the RFC and MLP models that are trained without federated learning. Two federated learning clients are considered by creating two local training datasets out of *internal week 1* and *internal week 2*. These two local training datasets are balanced out (in the same way as done in experiment 1) so that no negative effect due to imbalanced training datasets is experienced. Then, the global MLP model is trained with federated learning and when tested, the conclusion is that the MLP model trained using federated learning is indeed competitive with the RFC and MLP models that were trained without federated learning. The results can be seen in Table V. Thus, the loss of averaging multiple weights of the local MLP models is minimal.

Now that it is known that federated learning is capable of classifying the attack flows, the sixth experiment repeats the base experiment, but in a federated learning setup. This means that there are two clients and each of the clients gets assigned a local dataset. The same flow distributions as in the base experiment are reused, i.e. all "brute force" flows are removed from *internal week 1* and all "ping scan" flows are removed from *internal week 2* while the other attack types receive 3,359 and 3,366 flows respectively. Then, TensorFlow Federated's federated learning implementation, with federated averaging as aggregation algorithm, is used to see whether the global MLP model does learn to recognize flows from the attack types "brute force" and "ping scan", although only one of the clients contains flows from these attack types. The results in Table V show that the global MLP model indeed starts to learn how to classify flows of the attack types "brute force" and "ping scan". However, there is a big difference in the score of "brute force" and the score of "ping scan". The scores of the other attack types are comparable with the results in experiment 4. Compared to the base experiment, the first client is now able to predict 20% of the "brute force" flows as opposed to 0% without federated learning, but also loses in capability of correctly predicting "ping scan" flows. The score drops from about 95% to 67%. The second client improves its score on "ping scan" from 0% to 67% but also loses in capability of predicting the attack type "brute force": the score drops from 98% to 20%. The conclusion for the sixth experiment is that federated learning allows for their clients to start detecting attacks they previously could not detect. But, federated learning is no silver bullet. With the advantage of federated learning, also comes a disadvantage. Namely, that clients can also lose the ability to correctly predict some attacks due to the fact that the weights of the local MLP models of both clients are averaged.

TABLE V
PRECISION SCORES OF TRAINED RFC MODELS IN EXPERIMENTS 3 TO 6 (ALL VALUES IN PERCENTAGES)

Attack type	exp. 3	exp. 4	exp. 5	exp. 6
Normal	99.56	90.33	96.42	98.08
BruteForce	6.179	97.15	81.04	19.76
DoS	99.92	100.0	99.89	100.0
PingScan	0.1831	94.97	92.26	67.00
PortScan	90.21	85.65	85.85	94.52

Finally, a seventh experiment that also uses federated learning and provides additional tests is presented. A first shortcoming of the sixth experiment is that only flows of the attack types "brute force” and “ping scan” are removed from the local datasets of respectively client 1 and client 2. What is the result if other combinations of attack types are removed? A second remark on the sixth experiment is that removing all flows of certain attack types is quite abrupt. Although such scenarios are not impossible, the usability of federated learning is not only limited to scenarios where certain datasets have no flows of a certain attack type while others do. Therefore, some additional tests are provided in the seventh experiment.

In the seventh experiment, instead of removing all flows of an attack type, a small portion of flows of the attack type is left in the local training dataset of a client. The experiment starts with creating more or less balanced datasets (3,359 flows for all attack types except 1,626 for “brute force” in *internal week 1* and 3,366 flows for all attack types except 2,731 for “ping scan” in *internal week 2*) for each client that can be used as a baseline dataset. Then, six tests are executed in which every time, for each of the baseline datasets, 75% of the available flows of one of the attack types is removed. In each test, a different combination of attack types is chosen. This way, each of the combinations of attack types is tested. It turns out that the results of these tests are impressive, as can be seen in Table VI. In this table, the attack types from which 75% of the flows were removed during each test, are indicated by accentuating the corresponding prediction scores in bold and italic. Almost all scores are well above 90%, also for the attack types from which 75% of the flows in the balanced dataset is removed. But, like in experiment 6, “brute force” deviates from this in two occasions. The first test where brute force has a lower score is the test where the combination is “brute force” and “ping scan”, this is to be expected since it is the same combination as used in experiment 6. The second test that has the same fate for the score of “brute force” is the test where the combination “brute force” and “port scan” is used. In both tests where the scores for “brute force” deviate from the rest, the biggest group of the samples is predicted to be “port scan”. This is also to be expected because this was also observed in experiment 6. A possible explanation for this effect is that the feature distribution of the “brute force” flows is similar to the distribution of the “port scan” flows and until the model has not seen enough “brute force” flows, it cannot separate them correctly from “port scan” flows. But it has to be noted that the deviated score in this seventh experiment is much better than seen in experiment 6. Here, the scores are respectively 80% and 68% which is a big difference with the 20% as seen in experiment 6.

So, after experiments 6 and 7, the overall conclusion about federated learning is that it is certainly valuable to be used in the context of network intrusion detection. Federated learning enables multiple network intrusion detection classification models to learn from each other by sharing and aggregating the weights of the models, rather than sharing the local datasets.

TABLE VI

PRECISION SCORES OF ALL TESTS IN EXPERIMENT 7 (ALL VALUES IN PERCENTAGES)

Attack type	test 1	test 2	test 3
Normal	97.05	96.18	97.21
BruteForce	***93.43***	***80.10***	***68.36***
DoS	***100.0***	99.93	99.93
PingScan	95.66	***95.76***	94.54
PortScan	90.46	90.77	***92.38***

Attack type	test 4	test 5	test 6
Normal	93.58	94.31	94.46
BruteForce	93.44	92.59	94.29
DoS	***99.88***	***99.88***	99.93
PingScan	***95.30***	94.81	***91.91***
PortScan	87.22	***86.75***	***89.62***

ACKNOWLEDGMENTS

First and foremost, I would like to thank Laurens D’hooge for assisting me throughout all steps in the creation of this master’s thesis. His feedback and assistance was vital in order for me to come to the thesis as is. Secondly, I would also like to thank my father, Rik Van de Walle, who has thoroughly read my thesis and has given important feedback. Also, thanks to Prof. dr. Bruno Volckaert and Dr. ir. Tim Wauters for giving me the opportunity to do research about federated learning in network intrusion detection. This is, of course, not an exhaustive list of people that deserve to be thanked. Therefore, also a big thank you to all who supported me in the last few months.

REFERENCES

[1] Bill Whitaker, “SolarWinds: How Russian spies hacked the Justice, State, Treasury, Energy and Commerce Departments,” 2021.

[2] Lucian Constantin, “SolarWinds attack explained: And why it was so hard to detect,” 2020.

[3] Margaret Brennan, “”Hack everybody you can”: What to know about the massive Microsoft Exchange breach,” 2021.

[4] Setareh Roshan, Yoan Miche, Anton Akusok, and Amaury Lendasse, “Adaptive and online network intrusion detection system using clustering and Extreme Learning Machines,” *Journal of the Franklin Institute*, vol. 355, no. 4, pp. 1752–1779, 2018.

[5] AT&T, “Intrusion Detection Systems (IDS) explained,” 2021.

[6] Michele Pagano, “Binary and multi-class classification in network anomaly detection using deep neural networks,” 2020.

[7] Markus Ring, Sarah Wunderlich, Dominik Grüdl, Dieter Landes, and Andreas Hotho, “Flow-based benchmark data sets for intrusion detection,” in *Proceedings of the 16th European Conference on Cyber Warfare and Security (ECCWS)*, pp. 361–369. ACPI, 2017.

[8] Paessler, “All-in-one NetFlow Analyzer PRTG,” 2021.

[9] Markus Ring, Sarah Wunderlich, Dominik Gruedl, Dieter Landes, and Andreas Hotho, “Technical Report CIDDS-001 data set,” in *Proceedings of the 16th European Conference on Cyber Warfare and Security (ECCWS), to appear*, vol. 001, pp. 1–13. ACPI, 2017.

[10] Abhishek Verma and Virender Ranga, “On evaluation of network intrusion detection systems: Statistical analysis of CIDDS-001 dataset using machine learning techniques,” *Pertanika Journal of Science and Technology*, vol. 26, no. 3, pp. 1307–1332, 2018.

[11] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas, “Communication-efficient learning of deep networks from decentralized data,” *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017*, vol. 54, 2017.

[12] Sawsan Abdul Rahman, Hanine Tout, Chamseddine Talhi, and Azzam Mourad, “Internet of Things intrusion Detection: Centralized, On-Device, or Federated Learning?,” *IEEE Network*, vol. 34, no. 6, pp. 310–317, 2020.

[13] Nuno Oliveira, Isabel Praça, Eva Maia, and Orlando Sousa, “Intelligent cyber attack detection and classification for network-based intrusion detection systems,” *Applied Sciences (Switzerland)*, vol. 11, no. 4, pp. 1–21, 2021.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Almost all organizations today have information technology (IT) as a vital part of their business model. The use of information technology ranges from the mainframes and big data centers that the largest of companies use, all the way to the self-employed who uses commodity hardware (e.g. to create invoices). IT is used for communication between all parties involved in a company: employees, clients, suppliers, governing bodies... IT is also used for inventory management, for storage of all sorts of data (of which some may be sensitive), etc. [5]. Put shortly, information technology has been very important in the improvement of efficiency in the workflow of companies, and it will remain vital in the future.

Since companies may store sensitive information in their IT infrastructure, IT has become a coveted target for hackers. As been proven multiple times by the damage that can be done by attackers hacking their way into corporate IT infrastructure, companies must try to limit the chances of hackers being able to break into the companies' IT infrastructure. Possible means of doing so are by imposing user authentication before employees can gain access to IT infrastructure, firewall rules to impose network policies, installing anti-virus software on the computers of the employees. However, none of these solutions are bulletproof because of the risk of human error. Hackers can use social engineering approaches to gain information about employees or even obtain user credentials. Even more worrying is that the flaws that attackers exploit, may not even come from a mistake made by any of the employees of the company itself. There could be flaws in any of the software that the company uses. Even software that one might think of as being secure and reliable, can contain flaws. Recently, on two separate occasions, hackers were able to use flaws in widely used software and breach into tens of thousands of governmental and private computer networks. The first started in March 2020 when Russian hackers were able to compromise an update of SolarWinds' network management software Orion and install malware in organizations and governing bodies that were updating their SolarWinds Orion software. The scope of this breach was about 18,000 networks [6, 7]. More recently, another very serious attack occurred. Early March 2021, Chinese hackers exploited vulnerabilities in Microsoft Exchange and were able to hack into hundreds of thousands of organizations worldwide [8].

The costs of cybercrimes worldwide are not to be overseen. Cybersecurity Ventures (researcher and publisher of global cyber economy, cybersecurity facts and statistics) expects a global cost in 2021 of 6 trillion USD (i.e. $\$ 6 \times 10^{12}$) [9]. In 2018, the Center for Strategic and International Studies (CSIS) presented a report in partnership with McAfee in which James Lewis concluded that the annual cybercrime costs worldwide in 2017 were between $445 billion and $608 billion while the costs three years earlier in 2014 were estimated by CSIS to be between $345 billion and $445 billion [10]. This is a significant growth in costs in three years time. In McAfee's fourth report on the costs of cybercrime released in 2020, the authors estimated that since 2018 the global costs of cybercrime exceed $1 trillion [11]. Although the $1 trillion estimated by McAfee is significantly different from Cybersecurity Ventures' estimate of $6 trillion, these amounts are huge. If compared to the Gross Domestic Product (GDP) of the largest economies of the world, the costs of cybercrime are indeed huge. The costs of cybercrime are equal to the GDP of the Netherlands (which is ranked 17^{th} worldwide with respect to the GDP) and are equal to more than a third of the GDP of the fifth biggest country (with respect to the GDP) France [4]. The expected outcomes of the GDP of the top five countries in 2021, according to the International Monetary Fund (IMF) in October 2020, can be seen in Table 1.1.

As if all of this isn't already bad enough, experts from Cybersecurity Ventures expect global cybercrime costs to grow by 15 percent per year over the next period of five years. If so, the outcome in 2025 would be a global cost of $10.5 trillion[1] according to Cybersecurity

[1] $6 trillion in 2021, growth of 15% per year means $1.15^4 \times 6$ trillion dollars in 2025, or $10.5 trillion

Table 1.1: Expected Gross Domestic Product (GDP), rank and share with respect to the global GDP in 2021 of top five countries (*source:* [4])

Country	GDP 2021 (billions of $)	Rank	Share (%)
United States	21,921.59	1	24.1
China	16,834.59	2	18.5
Japan	5,103.18	3	5.61
Germany	4,318.49	4	4.74
France	2,917.67	5	3.21

Ventures' estimations or $2 trillion[2] according to the estimations of McAffee. Due to all the damages done to the IT infrastructures and the serious financial costs of cybercrimes, the incentives for companies to innovate and invest in new information technologies could be at risk in the future [9].

1.1 Intrusion Detection System

As mentioned and motivated by the costs of cybercrime in previous paragraphs, it is important that companies invest in trying to avoid letting hackers break into their IT infrastructure. However, since human errors cannot be avoided at all times, prevention alone does not suffice. What if an attacker did manage to break into the infrastructure? Will this ever be discovered? If not, nobody will ever be able to handle the breach. To make sure that an intrusion can be dealt with, it must be detected. To lower the damage done and thereby the costs of the attack, the detection must happen quickly. This is where Intrusion Detection Systems come in handy.

Intrusion Detection Systems (IDS) are hardware or software applications that deal with the task of monitoring systems or networks with the purpose of detecting malicious activities [12, 13]. These malicious activities could compromise the six security goals on these systems or networks: confidentiality, authentication, access control (authentication), data integrity, non-repudiation and availability [14]. Once malicious activity has been detected, the IDS must generate alarms/notifications so that the intrusion can be dealt with appropriately. Either by a human operator or by an automated response.

[2] $1 trillion in 2020, growth of 15% per year means 1.15^5 × 1 trillion dollars in 2025, or $2.01 trillion

There exist two types of intrusion detection systems. The first type is anomaly-based network intrusion detection. Intrusion detection systems of this type try to find abnormal patterns in the systems or networks. However, a more accurate name of this type of intrusion detection systems would be “machine learning based” as opposed to “anomaly-based” since mostly supervised machine learning models for classification are proposed in the domain of intrusion detection. The classification models try to classify normal patterns and abnormal patterns as such. These abnormal patterns are potentially due to malicious activity. In recent years, this type of IDSs has been subject to a lot of research, certainly with the advances made in the domain of machine learning [2, 3, 12, 15, 16, 17, 18, 19, 20, 21, 22]. Thanks to machine learning, anomaly detection and classification have been made easier and show better results. Traditionally, experts had to manually determine thresholds for a system to classify some data as “normal” or as “anomalous”. Nowadays, experts let machine learning models determine these thresholds themselves by training the models with training data [23]. This can be done with supervised, unsupervised or semi-supervised machine learning algorithms. The advantage of using an anomaly-based intrusion detection system is that they can detect new types of attacks because everything that deviates from normal behavior (i.e. benign activities in the systems or networks the IDS is monitoring) can be detected. But, there is also a major drawback with this type of intrusion detection systems. They can generate a lot of false positives, that is generating alarms or notifications because it has recognized an anomalous pattern while the anomalous pattern is not due to an anomalous activity. This can result in a lot of work for system administrators who have to respond to these alarms raised by the IDS. They have to manually determine whether the alarm raised by an IDS is in fact coming from malicious activity or not [12].

The other type of intrusion detection systems is signature-based intrusion detection. These intrusion detection systems compare the activities they are monitoring to a predefined set of known patterns that correspond to certain intrusions. If the IDS finds an activity that matches such a pattern of an intrusion, it must raise an alarm or notification. The advantage of signature-based intrusion detection systems is that they can be very accurate against well-known attacks. On the other hand, the disadvantage of these types of systems is that the signature databases need to remain up-to-date in order to be able to detect new types of attacks. This makes the signature based intrusion detection systems less usable for finding new types of attacks [12].

In the case of the IDS finding an intrusion, the IDS must respond. The response can either be a passive response, or an active response. A passively responding IDS sends a notification by email, a text message or any other means of communication to someone, a system administrator for example, to inform that person of a possible network intrusion. The person receiving such a notification must take the appropriate actions to figure out the severity of the alert and must handle the possible intrusion. On the other hand, the response of an IDS could be an active response. If such an IDS detects an intrusion, it could actively change the firewall rules to prohibit any further communication between the possible attacker and the attacked system(s). Another possible action an active IDS may take is killing the processes being attacked [13]. In any case, the actions taken by a person or automatically by the IDS itself must ensure that there is as little as possible damage done to the infrastructure or company.

Intrusion detection can be done at network level, or at host level. A Network Intrusion Detection System (NIDS) monitors the ingress and egress traffic from the network it is monitoring. Such a system can be placed in the demilitarized zone (DMZ) of the network. This is the part of an organization's network that is in between the company's Local Area Network (LAN) and an untrusted network (e.g. the Internet) [24]. An example of a network with a NIDS placed in the DMZ can be found in Figure 1.1. All the network traffic that is coming from the Internet (ingress traffic) and is going into the Internet from the enterprise's network (egress traffic) is monitored by an intrusion detection system [13].

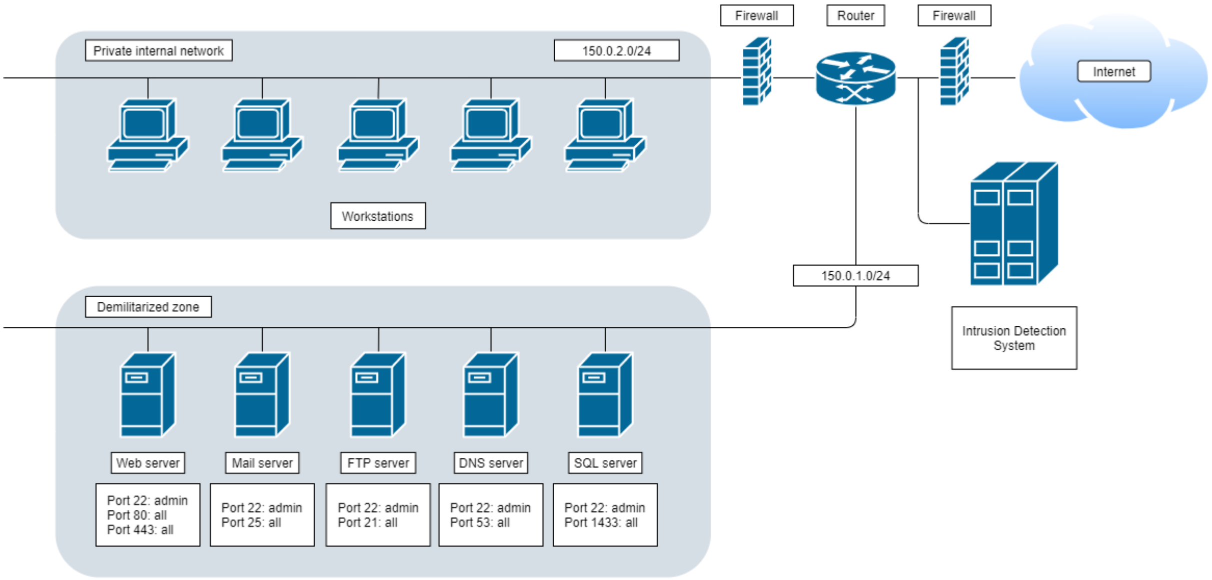


Figure 1.1: Example network with a network intrusion detection system in the demilitarized zone

A Host Intrusion Detection System (HIDS) however works quite differently. With a host intrusion detection system configuration, all nodes within an enterprise's network that have access to or can be accessed from the untrusted network (e.g. the Internet) have a HIDS agent installed. Such a configuration can be seen in Figure 1.2. The NIDS agents analyze the configuration and applications specific activity, monitor modifications to the file system, environment variables, network traffic, etc. If any of the activity on the host that is being monitored looks suspicious to the NIDS agent, it will generate an alarm.

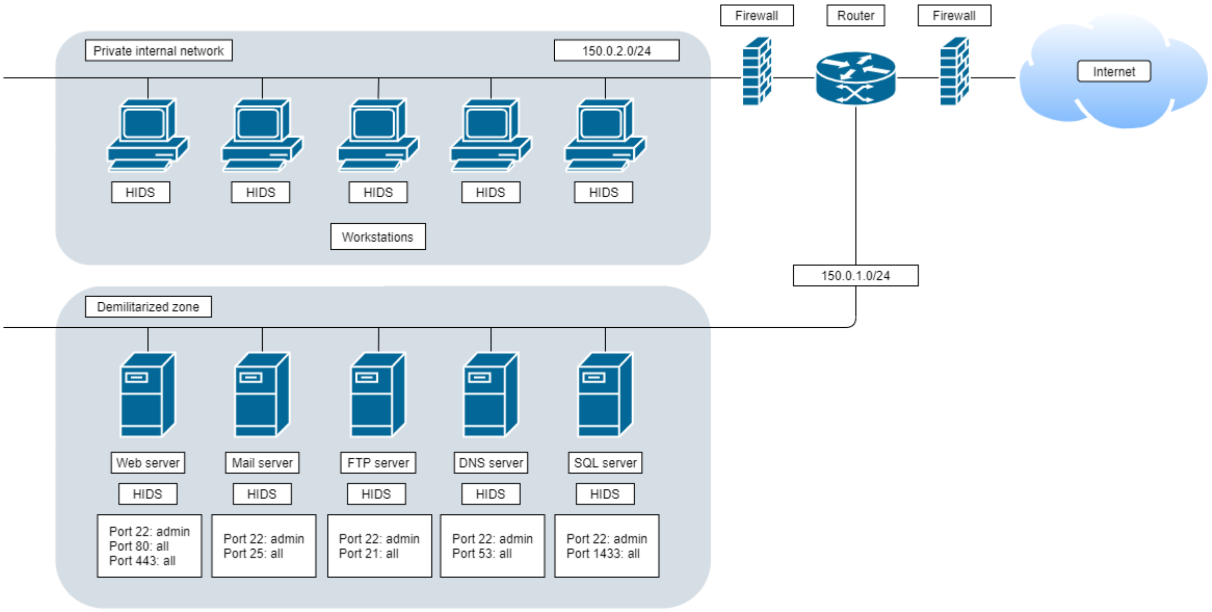


Figure 1.2: Example network with a host intrusion detection system with HIDS agents installed on all workstations and servers

In everything that follows in this thesis, the focus will be on intrusion detection systems that are of the type "anomaly-based" and on the level of the network. If not stated otherwise, the reader should keep in mind that every time the term "intrusion detection system" is used, the focus lays on anomaly-based network intrusion detection systems.

At this point, the reader should be familiar with what an Intrusion Detection System does and why a company or governing body or anyone else who uses a network (maybe except for household networks) should at least consider using one. In following paragraphs, the goal of this thesis will be explained.

1.2 Goals of the thesis

The goal of this thesis is to develop a new strategy for detection of network intrusions. Nowadays, network intrusion detection systems are often based on machine learning models. To be able to train these machine learning models, datasets containing captured network traffic are required. These datasets can come from academic institutes such as the Canadian Institute for Cybersecurity (CIC). Once a machine learning model is trained, it is tested with a test set. Both the training set and the test set come from the same dataset, but the trained model has never seen the data in the test set during the training process. One of the current problems to be solved in the field of intrusion detection is that trained models can score good on the test set coming from the same dataset as where the training set comes from, but can score poorly on a test set that comes from another dataset. Even though both datasets are extracted from the same network setup, by the same research group and have the types of features captured. This statement will be proven in the section §4.3 using the CIDDS-001 datasets. The reason for the poor scoring on test data from another data set could be due to the fact that the underlying characteristics of the captured data can be different. For example, the data in the first dataset may contain data of a class which is not present in the other dataset. If that is the case, the model trained on the second dataset can never correctly classify these types of data because it has never been trained to classify them. However, these results are not preferable for systems that will eventually run in a production environment. In production, the goal of an IDS is that it works well on all types of attacks or that it at least can learn from future data and thus the IDS will eventually work well on all attacks.

In this thesis, research will be conducted on whether or not the concept of Federated Learning (FL) can solve this problem. Federated learning is a concept where nodes, in this case IDSs, can learn from each other. Each of the nodes has a local model, in this case a model that classifies network traffic as “benign” or as “malicious”. All nodes can only see, and thus can only learn from, local data (i.e. network traffic that the IDS is monitoring). When multiple nodes learn different patterns related to malicious activity in a network, the optimal scenario would be that all nodes could learn from each other. If all nodes can learn from all other nodes, every participating node will eventually be able to detect all types of malicious activity that any of the participating nodes has learned.

With the concept of federated learning in mind, one node can be set up to be receiving data from dataset X while another node is set up to be receiving data from dataset Y. Dataset X lacks an attack type which is available in dataset Y and vice versa. Both nodes should try to improve their local model based on the data from the dataset they are receiving. With federated learning, the improved node should notify its peers with an updated model so that other nodes can learn from the improved model. The goal is to eventually come to a scenario in which both nodes can detect the types of attacks that were not present in their own dataset.

2
Related work

2.1 Coburg Intrusion Detection Data Sets

In this thesis, the Coburg Intrusion Detection Data Sets (CIDDS) will be used to evaluate the constructed federated learning based network intrusion detection system. CIDDS-001 is a labelled flow-based dataset of network traffic that can be used for the evaluation of anomaly-based intrusion detection systems. The dataset was created using an emulation of a small business environment. The researchers of this project used the open source cloud computing infrastructure OpenStack to emulate this environment. The setup was created to mimic a typical small business setup with a subnetwork for the software development department, one for the office department, one for the management department and one where typical servers reside: a mail server, a web server, a file server and a backup server. They used Python scripts to emulate normal user behavior of the clients. The dataset contains data captured using the NetFlow protocol. NetFlow is a network monitoring protocol that was created by Cisco and allows to capture IP data on a network interface [25]. Table 2.1 gives an overview of the attributes of the data that can be found in the CIDDS-001 dataset [1, 26].

Table 2.1: Attributes in the CIDDS-001 datasets

Nr.	Name	Description
1	Src IP	Source IP Address
2	Src Port	Source Port
3	Dest IP	Destination IP Address
4	Dest Port	Destination Port
5	Proto	Transport Protocol (e.g. ICMP, TCP, or UDP)
6	Date first seen	Start time flow first seen
7	Duration	Duration of the flow
8	Bytes	Number of transmitted bytes
9	Packets	Number of transmitted packets
10	Flags	OR concatenation of all TCP Flags
11	Tos	Type of service
12	Class	Class label (normal, attacker, victim, suspicious or unknown)
13	AttackType	Type of Attack (portScan, dos, bruteForce, —)
14	AttackID	Unique attack id. All flows which belong to the same attack carry the same attack id.
15	AttackDescription	Provides additional information about the set attack parameters (e.g. the number of attempted password guesses for SSH-Brute-Force attacks)

The network that was created by the researchers can be found in Figure 2.1. As mentioned before, OpenStack is used to emulate the internal network of a small business. This internal network contains four subnetworks: developer, office, management and server. Within this network, network traffic is captured using the NetFlow protocol at two different points indicated in Figure 2.1 by two stars. The first point is at the internal router. The second point is at an external server that runs a file synchronization service (Seafile) and a web server. The file synchronization service is meant to be used by internal clients from the development and management subnetworks. The web server is meant to be used by whomever is interested to visit the website, both for internal and external clients [1].

The CIDDS-001 datasets can be downloaded from the website of *Coburg University of Applied Sciences*[1]. After unzipping the dataset, four subdirectories can be found: `attack_logs`, `client_confs`, `client_logs` and `traffic`. The first three contain extra

[1] https://www.hs-coburg.de/cidds/

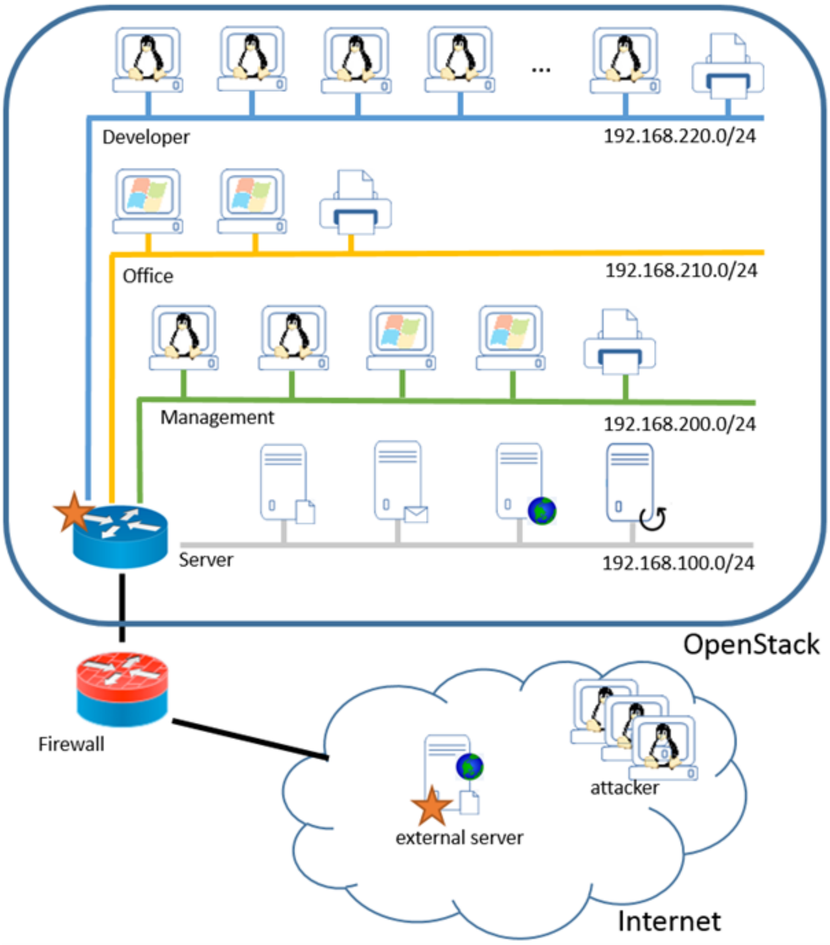


Figure 2.1: Emulated small business network environment (*source:* [1])

metadata about the configuration of the clients as well as the logs that the clients created during the experiment. The fourth directory contains two subdirectories `ExternalServer` and `OpenStack`. In these directories, the actual captured data reside. The files in the directory `ExternalServer` contain the data captured at the external server while the files in the directory `OpenStack` contain the data captured at the internal router. The files in each of the directories are named by applying following convention:

```
CIDDS-<version>-<origin>-<period>.csv
```

Since the first version of CIDDS is used, `<version>` will always be replaced with `001`. The origin depends on where the data is captured: `external` for data captured at the external server (i.e. for files under the directory `ExternalServer`) and `internal` for data captured at the internal router (i.e. for files under the directory `OpenStack`). The final part tells when the network traffic was captured: `week1`, `week2`, `week3` or `week4`.

2.2 Evaluation of existing machine learning based network intrusion detection algorithms

The authors Abhishek Verma and Virender Ranga have evaluated multiple machine learning based network intrusion detection algorithms on the CIDDS-001 dataset [2]. In this work, both supervised and unsupervised machine learning algorithms are considered:

- supervised learning techniques
  - k-Nearest Neighbours (KNN)
  - Support Vector Machine (SVM)
  - Decision Trees (DT)
  - Random Forests (RF)
  - Artificial Neural Networks (ANN)
  - Naive Bayes (NB)
  - Deep Learning (DL)
- unsupervised learning techniques
  - k-Means Clustering (KMC)
  - Expectation Maximization Clustering (EMC)
  - Self-Organizing Maps (SOM)

Most of the algorithms that are learned in a supervised manner score very good. KNN, SVM, DT, RF and DL all have an accuracy of more than 94% on the CIDDS-001 data captured at the external server, and more than 99% accuracy on the CIDDS-001 data captured internally in the OpenStack network. With accuracy being:

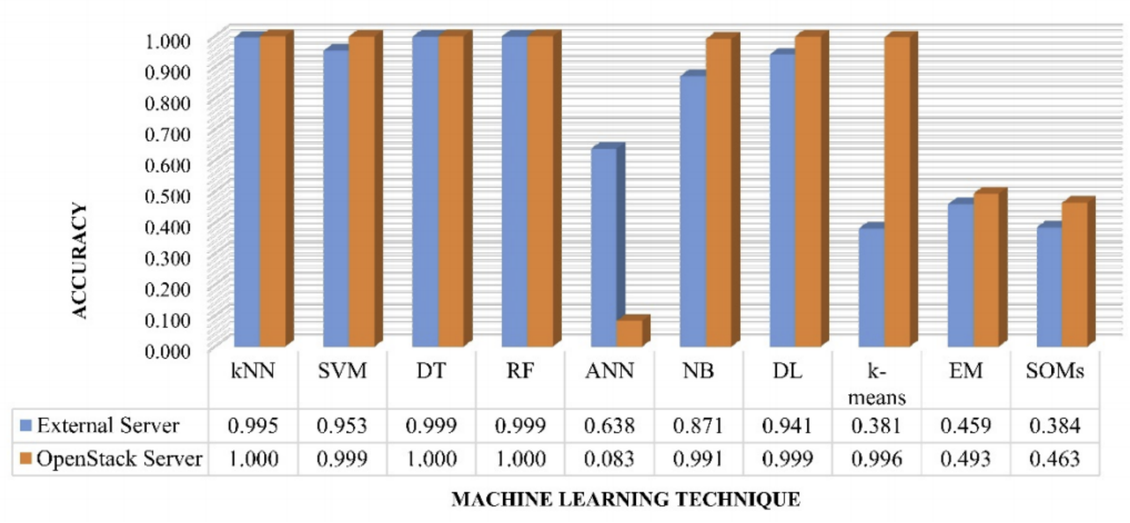
$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.1}$$

where

- TP = True Positives = number correctly classified instances as attack
- TN = True Negatives = number of correctly classified instances as normal

- FP = False Positives = number of incorrectly classified instances as attack
- FN = False Negatives = number of incorrectly classified instances as normal.

The results of their findings with respect to the accuracy of the models can be found in Figure 2.2.



	kNN	SVM	DT	RF	ANN	NB	DL	k-means	EM	SOMs
External Server	0.995	0.953	0.999	0.999	0.638	0.871	0.941	0.381	0.459	0.384
OpenStack Server	1.000	0.999	1.000	1.000	0.083	0.991	0.999	0.996	0.493	0.463

Figure 2.2: Performance evaluation of machine learning techniques on CIDDS-001 datasets by Abhishek Verma and Virender Ranga (*source: [2]*)

Although these results are very good for some of the evaluated algorithms, the setup assumes that all the data on which the algorithms are trained are available locally. This is, however, an assumption that is not always true in production environments. In Figure 2.3, an example network setup is depicted in which the organization has two geographically separated sites. Each site contains its own Local Area Network (LAN). On top of this, computers in different LANs are able to communicate with each other. This can be established by connecting the LANs to the public Internet, or by setting up a Wide Area Network (WAN) that connects both LANs, etc.

In these situations, one can try to transfer all the data to a central server that is responsible for training the intrusion detection model. But this is not always eligible or desired. Both intrusion detection systems will capture information about flows of network traffic that they see. The first reason why centralizing these flows is not desirable is that sending these captured network flows to a central server can result in too much use of the available bandwidth just for the purpose of intrusion detection. This could hinder regular business operations and is not desirable. On the other hand, sending network flows to

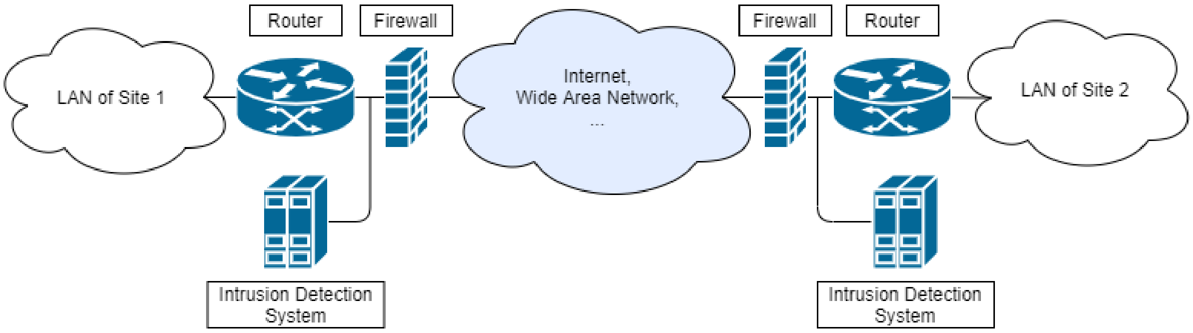


Figure 2.3: Example network setup of organization with two, geographically separated, sites

another server could violate privacy regulations. The captured network flows might contain sensitive data which therefore may not be transferred to another location (e.g. due to government constraints, like GDPR) [27]. But an organization may also decide not to copy these network flows to another location because it does not want to take the risk of someone eavesdropping the captured network flows.

With all of this in mind, one could choose to let both intrusion detection systems work completely independent from each other. Both IDSs could have their services running locally, and therefore detecting and learning new intrusions on the data coming only from their own LAN. However, this might not be very efficient and can result in waste of resources. If the IDS of site 1 detects a new anomalous pattern in its local data which corresponds to a new type of attack, only that IDS will have knowledge about the new type of intrusion. The IDS of site 2 will not benefit from the new findings of the IDS of site 1. This implies that when the LAN of site 2 is attacked using the new type of attack, which is learned already by the IDS of site 1, the IDS of site 2 first has to detect this new attack before it can raise any alarms. However, instead of setting up the IDSs as two independent entities, one could use a collaborative learning approach like federated learning. This makes sure that the captured network traffic flows do not have to be centralized but allows for all the participating IDSs to learn from each other.

2.3 Federated Learning

In 2017, Google proposed a concept called *federated learning* in [28] as an answer to problems with machine learning models that have to learn on huge datasets and/or datasets containing privacy sensitive data. A lot of people have smartphones these days which contain useful information that can be used to enhance the user experience. For example, Google could collect text written by people to enhance their text entry models. Traditionally, such models were trained on centralized data, but text written by smartphone users should not be centralized for privacy reasons. An alternative is to let the smartphones learn a model on their own local data, centralize the locally computed updates to a central server which in turn aggregates these locally computed updates into a global model. This global model is then redistributed among the participating smartphones. The process of locally computing updates, centralizing and then updating the updates can be done repeatedly.

Based on Google's concept of federated learning, in [27], Q. Yang et al. have proposed a general definition of federated learning that extends the concept proposed by Google to a general concept for all privacy-preserving decentralized collaborative machine learning techniques. The conventional method for N data owners $\{F_1, ..., F_N\}$ that want to train a machine learning model with their local data $\{D_1, ..., D_N\}$ is to centralize this into a single dataset $D = D_1 \cup ... \cup D_N$ to train a central model M_C. Federated learning on the other hand takes another approach. All N data owners $\{F_1, ..., F_N\}$ collaboratively train a model M_F without having F_i to share its local dataset D_i to its peers. An important additional requirement for the federated model M_F is that its accuracy α_F should be very close to the accuracy α_C which is reached by the centralized model M_C. If δ is a non-negative real number and

$$|\alpha_F - \alpha_C| < \delta, \tag{2.2}$$

M_F has δ-accuracy loss.

In [15], S. Rahman et al. compare the performance of a federated learning scheme to an on-device learning approach and a centralized approach. The on-device learning approach used is *SecProbe* which is proposed by L. Zhao et al. in [29]. The centralized approach is considered to be the best case scenario because no compromises are made with respect to training a model: all local training data are available for training the model. However, as mentioned above, this comes with risks in the privacy-preservation requirements. S. Rahman et al. found that, after enough training rounds, their federated learning model outperforms the SecProbe model and approaches the accuracy reached by the centralized model.

Li et al. have proposed a federated deep learning model *DeepFed* in which multiple industrial cyber-physical systems (CPSs) participate in training a collaborative privacy-preserving intrusion detection model [3]. These CPSs refer to large-scale, geographically-dispersed, complex and heterogeneous Industrial Internet-of-Things (IIoT) systems. On each of the participating CPSs, a local deep learning model is present which is mainly made up of a Gated Recurrent Unit (GRU) module, a Convolutional Neural Network (CNN) module and a multilayer perceptron (MLP) module. The GRU module and the CNN module both process the feature vector of the captured network flow. The results of these two modules are concatenated to a single vector that is further fed to the MLP module. The result of the MLP module is then processed by a softmax layer which assigns a probability to each of the output classes. The schematic overview of this DeepFed module can be found in Figure 2.4.

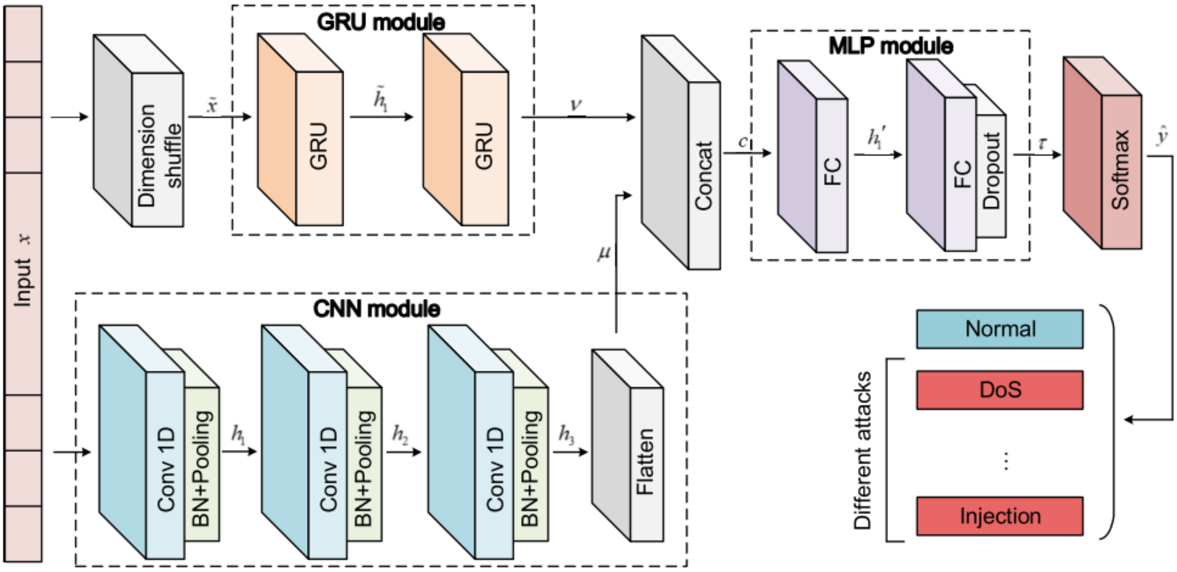


Figure 2.4: Architecture of DeepFed's model that is present on each of the federated learning participating cyber-physical systems (*source: [3]*)

.

3

Experimental Prerequisites: Dataset Preprocessing & Algorithmic Choices

This chapter discusses the experimental prerequisites of this thesis. This comes down to giving insight into the dataset preprocessing steps and the algorithmic choices. First, an overview of the programming language and frameworks is given together with the specifications of the computer that was used to run all experiments of this thesis on. Second, a deeper look is taken at the actual content of the CIDDS-001 datasets. In this section, the motivations behind why specific parts of the CIDDS-001 datasets are used and why others are not, is described. Additionally, the choice for which CIDDS-001 attribute as target for classification is motivated in this section. Third, all the preprocessing steps that are performed on the CIDDS-001 datasets are presented. And finally, a section is committed to describing the two classification models that are used throughout the thesis.

3.1 Programming environment and hardware

This thesis contains multiple experiments that allow to answer the question: "Can multiple machine learning based network intrusion detection systems learn from each other?" All experiments are written in the programming language Python, version Python 3.8.10 [30]. For loading, getting access to and manipulating the datasets used for training and testing the machine learning models, the pandas framework, version 1.2.4, is used [31]. On top of that, two machine learning frameworks were used throughout the experiments: scikit-learn version 0.24.2 [32] and TensorFlow version 2.2.0 [33].

The computer that is used to run the experiments on is an HP Pavilion Power Laptop 15-cb0xx which has an Intel Core i7-7700HQ CPU and 16 GB of RAM. The operating system installed on this laptop is Windows 10 Home version 21H1.

3.2 Dataset

The datasets that are used in the experiments are the week 1 and week 2 internal datasets available in CIDDS-001. The origin of these datasets is already thoroughly discussed in §2.1. Once the CIDDS-001 datasets are downloaded, the data need further preprocessing before they can be used. But, before the preprocessing is discussed, an overview of the content in the CIDDS-001 datasets is given.

3.2.1 Content of the CIDDS-001 datasets

Motivating partial dataset use

As mentioned in §2.1, the CIDDS-001 datasets contain captured network traffic of benign and malicious traffic sent to a simulated organizational network. The traffic was generated over a time span of 4 weeks. For each week, a dataset is provided for both internal and external captured data (cfr. §2.1). Throughout these four weeks, the authors of the CIDDS-001 datasets generated benign and malicious traffic. Table 3.1 gives an overview of the number of attacks that were executed throughout these four weeks.

It is clear that the two datasets *internal week 1* and *internal week 2* contain the most

Table 3.1: Overview of executed attacks of which network flows were captured and are available in the CIDDS-001 datasets (*source: [1]*)

	OpenStack (internal)				Exernal Server			
	PortScan	PingScan	DoS	BruteForce	PortScan	PingScan	DoS	BruteForce
week1	16	10	11	5	0	0	0	0
week2	8	6	7	7	2	0	0	4
week3	0	0	0	0	5	0	0	7
week4	0	0	0	0	1	0	0	3

attacks and they contain captured network flows of attacks of all types. The datasets *internal week 3*, *internal week 4* and *external week 1* do not contain any attack flows at all. The remaining datasets *external week 2*, *external week 3* and *external week 4* only contain attack flows of PortScan and BruteForce attacks. With this overview in mind, and the fact that only two participating nodes in the federated learning setup are considered in the experiments, the datasets *internal week 1* and *internal week 2* are used as they provide the most attack flows.

Motivating the choice for which attribute to use as classification label

Table 2.1 in the previous chapter (Related Work), gives an overview of what features of the captured network flows are available in the CIDDS-001 datasets (first eleven attributes) and some labels given to the captured network flows (last four attributes). The goal of classification models is that given a set of features (also known as the feature vector), the model can predict the correct class/label corresponding to the feature vector. With the CIDDS-001 datasets, the feature vector consists of the first eleven attributes listed in Table 2.1 and the class/label that a model has to predict is one of the last four attributes. Some papers (e.g. [34], [35]) choose the attribute named “Class” (i.e. the twelfth attribute in Table 2.1) to be the classification label. Although it can be useful to know whether a network flow is either normal, attacker, victim, suspicious or unknown, if a model can predict the specific attack type in case of malicious flows, the IDS can be more specific in the alerts it generates or can take specific actions to prevent further damage. Therefore, in the experiments in this thesis, the attribute that is chosen to be the classification label, is the attribute “AttackType” (i.e. the thirteenth attribute in Table 2.1). For the remaining two metadata attributes “AttackID” and “AttackDescription”, it does not make sense to pick these attributes as classification labels. The “AttackID” is a unique id, generated per attack to bundle multiple flows that belong to the same attack. Since each new attack receives a new attack id of which the value is not related to the features of the flows corresponding to the new attack, this cannot be used as a classification label. The attribute “AttackDescription” provides additional information about the executed attack and is also not eligible for being a classification label.

Attack distribution in partial dataset

At this point, the motivations behind why the datasets *internal week 1* and *internal week 2* are chosen to work with and the motivations behind the choice for “AttackType” as classification label, are given. This paragraph takes a deeper look into the actual content of the chosen datasets with respect to the classification label. In Table 3.2, a detailed overview of the number of flows of each attack type is given. Note that the value “Normal” in the column “Attack type” resembles benign flows while the other values in the same column resemble malicious flows. As can be seen, there is a great imbalance in the number of flows for the different types of captured network flows in both datasets. In *internal week 1*, 83.0% of the flows in the dataset are normal flows and in *internal week 2*, the normal flows make up 82.6% of the dataset. Also, within the malicious attack flows, the different types of attacks are not equally represented. Such imbalanced datasets can be problematic when training machine learning models. Certainly in cases where the imbalance is severe (as is the case for the CIDDS-001 datasets) since the minority classes will be more or less neglected during training. The model sees much more training samples from the majority classes tweaking the parameters more and more to the likes of these majority classes. As a consequence, the minority classes are not learned by the model. Therefore, before the machine learning models are trained, the training datasets are balanced out. There is one exception to this rule. In one experiment, the training dataset is not balanced. The motivation for why there is one exception to this rule will become clear once the experiments are discussed.

Federated usage of the data

It is clear now that in the experiments provided in this thesis, datasets *internal week 1* and *internal week 2* of the CIDDS-001 datasets are used. This paragraph discusses how these datasets are used in the federated learning setup. As mentioned in section 2.2, it is not always the case that data that might be used by machine learning models to

Table 3.2: Detailed overview of the number of different types of flows in the datasets *internal week 1* and *internal week 2*. The attack type “normal” resembles benign traffic flows while the other values in the first column resemble specific types of malicious flows

Attack type	Number of flows in dataset *internal week 1*	Number of flows in dataset *internal week 2*
Normal	7,010,897	8,515,329
PortScan	183,511	82,407
PingScan	3,359	2,731
DoS	1,252,127	1,706,900
BruteForce	1,626	3,366
Total	8,451,520	10,310,733

train with, is available on a single computer. This is also true in the field of network intrusion detection. It is not uncommon for (perhaps somewhat bigger) organizations to have multiple geographically separated networks. In each network, the organization might have put an intrusion detection system that monitors the network traffic. Figure 3.1 shows a hypothetical example of such an organizational network setup with two geographically separated LANs. Each subnetwork of a site has its own intrusion detection system that captures network traffic. Now, the question in this thesis is: “Can one IDS learn form another IDS using federated learning?”. For this, each participating IDS needs to have its own local dataset that is independent from any other local dataset. The two datasets *internal week 1* and *internal week 2* of CIDDS-001 are ideal for this scenario. Both datasets are independent from each other because they were created at different periods in time. Therefore, each of the two datasets is regarded as a local dataset of an intrusion detection system as show in Figure 3.1. In the experiments in this thesis, it is assumed that the data in *internal week 1* is captured by one of the two IDSs and that the data in *internal week 2* is captured by the other IDS.

3.2.2 Preprocessing the CIDDS-001 datasets

Now that the context around the use of the CIDDS-001 datasets *internal week 1* and *internal week 2* is given, the preprocessing of the datasets is discussed next. Important to note is that the first four preprocessing steps (i.e. (1) remove abnormalities, (2) one-hot encode categorical attributes, (3) Unfold OR-concatenated attributes and (4) drop some columns for classification) are executed before all experiments in this thesis. Whether

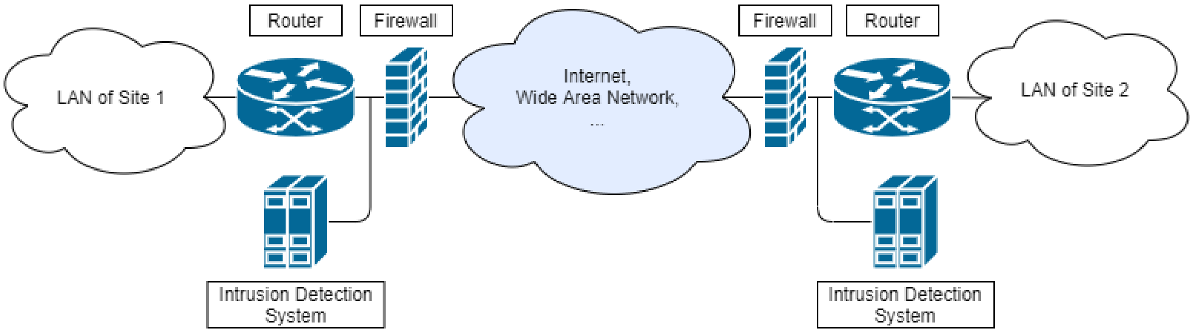


Figure 3.1: Example network setup with two, geographically separated, LANs in which each IDS is a participant of federated learning

the fifth and final preprocessing step (i.e. (5) normalize non-binary columns) is executed, depends on which model is used in an experiment. If an experiment uses the multi-layer perceptron neural network, the normalization preprocessing step is required. For the experiments that make use of the random forest classifier, no normalization step is required. The reason for this separation is further discussed in the corresponding subsection.

The first step to take after loading any of the datasets in CIDDS-001 is to clean the data and transform some of the columns to make them usable for classification models to learn with. All the CIDDS-001 datasets, both internal and external, are available as a comma-separated values (CSV) file. For each week, for both the internally captured network flows and for the externally captured network flows, there is one CSV file. Hence, the CIDDS-001 dataset consists of eight CSV files. Using pandas' function `pandas.read_csv()`, a CSV file can be loaded and made available as a `pandas.DataFrame` object for easy data access or manipulation. The columns that are available in the `pandas.DataFrame` objects after loading the dataset CSV files were already listed in previous chapter (Related Work) in Table 2.1.

In the following five subsections, the five steps for preprocessing the CIDDS-001 datasets are provided. Each step is given in sequential order as executed in the experiments of this thesis.

(1) Remove abnormalities

Upon inspection of the CIDDS-001 datasets, two abnormalities are found. First, there is an extra column `Flows` for which there is no documentation to be found in [26] (i.e. the source of Table 2.1 that lists the attributes of the CIDDS-001 datasets). However, in [1], the authors of the CIDDS-001 datasets briefly mention that this is a NetFlow attribute. But the meaning of the attribute is not further clarified. Furthermore, the only value this attribute has, for all flows in all datasets in CIDDS-001, is “1”, making this attribute a useless feature for the classification models. Presumably, the attribute `Flows` indicates an aggregation of measurements of captured network flows. It could be possible to aggregate NetFlow measurements (sums of duration, payload sizes...) for multiple/consecutive network flows and that the number of flows aggregated is given by the attribute `Flows`. But, since there is no indication in the documentation of CIDDS-001, this remains an assumption. In conclusion for this first abnormality: the undocumented column `Flows` is removed and thus ignored in its entirety. After removing the column `Flows`, the columns in the `pandas.DataFrame` objects holding the CIDDS-001 datasets, do correspond with the list of attributes listed in Table 2.1.

The second abnormality is found in the column `Bytes`. Some of the values contain a non-integer encoding (although not completely) for this attribute. If the number of transmitted bytes in a flow exceeds a million, the value in the column is encoded with an “M” to denote the million. The number $6,000,000$ is thus represented as a string $6\ M$ in the dataset. This encoding must be reversed in order to be able to perform mathematical calculations on the values of the column `Bytes` when normalizing the datasets. In addition, the paper [36] also states that next to the encoding of 10^6 as “M”, 10^3 is encoded as “K” as well. However, all `Bytes` attributes in all datasets are checked for containing the character “K” and there is no such entry (note: it is more correct to say that the `Bytes` attributes are checked for containing the character “k” after making sure that all characters in the column `Bytes` are in lower case). The code used to check this is shown in Listing 1.

In the code snippet to check if any 10^3 values are encoded as “K” in the `Bytes` column, the first thing that is done after importing the pandas library is fetching the CIDDS-001 dataset that needs to be checked from disk by reading the corresponding CSV file. After doing so, the CIDDS-001 dataset is available as a `pandas.DataFrame` object, here the `dataset` object. Then, a subset of that dataset is queried for where the values in the `Bytes` column contain a “K” or “k” character. The reason for checking for both capital and lower case “K”/“k” is to be sure that no entries that must be transformed, are missed.

```
import pandas as pd

dataset = pd.read_csv('<path to a CIDDS-001 CSV file>')

selection = dataset.where(dataset['Bytes'].str.lower(
    ).str.contains('k')).dropna()

print(f'Number of entries in dataset containing encoding for 1000: {len(selection)}')
```

Listing 1: Code snippet used to check if any 10^3 values encoded as “K” in the column “Bytes”

This is done by first lowering all characters in the `Bytes` column and then comparing flow by flow whether the `Bytes` attribute contains a “k”. Converting the case to lower case and checking for the string to contain a “k” is effectively the same as checking for either “K” or “k”. The `dropna()` after the `where()` is required because the `where()` function transforms, but does not remove, rows in the dataset object that do not evaluate to `True` in the condition of the `where()` function. The `where()` function transforms these rows to rows with only “NA” values. These values indicate “missing values” in the pandas framework. But only rows with non-missing values (i.e. rows for which the condition in the `where()` function did evaluate to `True`) are of interest. Therefore, all rows that contain missing values are dropped with `dropna()` to eventually obtain a `pandas.DataFrame` object that contains only rows where the `Bytes` column contains either character “K” or “k”. Finally, the length of that filtered dataset (i.e. the `selection` DataFrame) is printed out. This is done for all CIDDS-001 datasets, and it turns out that the lengths of all filtered datasets are 0, indicating no entries where 10^3 is encoded with a “K”.

(2) One-hot encode categorical attributes

After fixing the two anomalies that are found in the CIDDS-001 datasets, the categorical attributes in the CIDDS-001 datasets are being one-hot encoded. That means that each of the possible values in a categorical attribute, becomes a binary column to indicate whether that value applies or not. As an example of how one-hot encoding works, lets imagine a column with three possible values “a”, “b” and “c” as shown in Table 3.3.

For each of the three values, a column is created. For every row, the column that corresponds to the original value in the one-hot encoded column will have the value “1”. All

Table 3.3: Example column to be one-hot encoded

Index	Char
1	a
2	c
3	b
4	c

the other one-hot encoding columns in that row will have the value “0”. The one-hot encoded equivalent of the column shown in Table 3.3 thus becomes Table 3.4.

Table 3.4: Result of one-hot encoding the example column

Index	a	b	c
1	1	0	0
2	0	0	1
3	0	1	0
4	0	0	1

In the CIDDS-001 datasets, there is just one single categorical column which must undergo a one-hot encoding transformation. The categorical column is `Proto` which indicates what transport protocol was used for the captured network flow. The possible values in this column are: `ICMP`, `IGMP`, `TCP` and `UDP`.

(3) Unfold OR-concatenated attributes

The second column that needs to be transformed is the column `Flags`. This column contains string-values of six characters long representing an `OR`-concatenation of some of the TCP Flags that were set while the network traffic was captured. If, at the time of capturing the network flows, a flow was not making use of a TCP connection, then the `Proto` attribute of that flow was not set to “TCP” and no TCP flags in the attribute `Flags` were set. The value of the `Flags` attribute of such flows thus becomes the string `"......"`. If the protocol used in a flow is TCP, and any of the flags URG, ACK, PSH, RST, SYN and/or FIN were set (i.e. the bit value was 1 and not 0) the corresponding dot (i.e. the dot at the same position as the position of the flag as listed here) in the string `"......"` is replaced with respectively “U”, “A”, “P”, “R”, “S” and “F”. For example, if the flags ACK and FIN were set to 1, the value in the column `Flags` corresponding to that flow would be `".A...F"`. Since each character in this string represents a feature of the network traffic flow, the column `Flow` is expanded to six binary columns where each column represents whether or not the flag corresponding to that column was set or not.

(4) Drop some columns for classification

Some of the attributes that are available in the CIDDS-001 datasets have to be removed prior to training because they are highly correlated with the target, yet contribute no true improvement to model generalization. Most often these features fall under sample metadata.

The first type of columns that should be dropped are those that will not have useful contributions to the model. These columns are `AttackID` and `AttackDescription`. They give extra metadata on the flows that correspond to an attack and are not real features of captured data. Therefore, these columns should be dropped.

Further, also the columns `Date first seen`, `Src IP Addr` and `Dst IP Addr` should not be used to train the model with. The reason that these columns should be dropped is because the model should not be dependent on either the exact start, the specific sender and the specific receiver of the network traffic flow. The model should eventually become widely applicable and not depend on the source, destination or on the timestamp of the start of the traffic.

The last column that must be dropped is the column `Class`. As discussed in section §3.2.1, two attributes in the CIDDS-001 datasets could serve as classification label that the classification models will learn to predict. Since the choice is made for `AttackType`, the `Class` attribute must be removed.

The final list of attributes, with previous preprocessing steps and thus the transformed columns taken into account, can be found in Table 3.5. Note that also the names of the columns for which the content did not change (i.e. columns `Duration`, `Src Port`, `Dest Port`, `Packets`, `Bytes` and `AttackType` in Table 2.1) got renamed to become all lower case and a single word (spaces to separate two words are replaced with an underscore).

Table 3.5: Attributes in the CIDDS-001 datasets after one-hot encoding categorical attribute “Proto”, unfolding column “Flags”, removing unused columns and renaming column headers

Nr.	Name	Description
1	duration	Duration of the flow
2	icmp	Binary column indicating if flow uses ICMP protocol (comes from one-hot encoding `Proto`)
3	igmp	Binary column indicating if flow uses IGMP protocol (comes from one-hot encoding `Proto`)
4	tcp	Binary column indicating if flow uses TCP protocol (comes from one-hot encoding `Proto`)
5	udp	Binary column indicating if flow uses UDP protocol (comes from one-hot encoding `Proto`)
6	src_port	Source port
7	dst_port	Destination port
8	packets	Number of transmitted packets
9	bytes	Number of transmitted bytes
10	tcp_urg	Binary column indicating if flow uses the TCP flag URG (comes from unfolding `Flags`)
11	tcp_ack	Binary column indicating if flow uses the TCP flag ACK (comes from unfolding `Flags`)
12	tcp_psh	Binary column indicating if flow uses the TCP flag PSH (comes from unfolding `Flags`)
13	tcp_rst	Binary column indicating if flow uses the TCP flag RST (comes from unfolding `Flags`)
14	tcp_syn	Binary column indicating if flow uses the TCP flag SYN (comes from unfolding `Flags`)
15	tcp_fin	Binary column indicating if flow uses the TCP flag FIN (comes from unfolding `Flags`)
16	tos	Type of service
17	attack_type	Name of the attack in case of malicious flows, “—” in case of benign flows, this attribute is the classification label

(5) Normalize non-binary columns

Depending on how certain classification algorithms work, after removing some abnormalities in the CIDDS-001 datasets, transforming some of its columns (`Proto` and `Flags`), dropping unused features for classification and renaming the column headers as discussed in the last four subsections, they require the non-binary features to be normalized to prevent them to have a much bigger/smaller weight on the classification models than other columns. Classification algorithms that use decision trees (such as the random forest classifier) do not require the data to be normalized because each node in a decision tree partitions the data in two groups based on one feature. Whether the data is normalized or not, has no influence on how the data is partitioned. The parameter that splits the data would be scaled just as the data is scaled during normalization. Classification algorithms that make calculations using multiple features at once, such as the multi-layer perceptron neural network, do require the data to be normalized since using multiple features at once may result in certain features to have a bigger weight on the classification model than other features.

Throughout the experiments, the two aforementioned classification algorithms (random forest classifier and multi-layer perceptron neural network) are used. Hence, depending on which model is used in an experiment, a normalization technique is used on the dataset or not. In experiments where an RFC model is used, no normalization precedes the training phase while min-max normalization is used for experiments that use an MLP model.

Ideally, the normalization technique that is used can deal with outliers in the dataset. This requires that the parameters used in the transformation are not influenced by possible outliers. Parameters of a column that can be sensitive to outliers are, among others, the extrema (i.e. the minimum and the maximum), the mean and the standard deviation. There are different normalization techniques that can be used: min-max normalization, z-score normalization, robust scaling, etc. where certain techniques handle outliers better than others.

Min-max normalization is mathematically described as:

$$m_{i,j} = \frac{x_{i,j} - min_j}{max_j - min_j}, \quad \forall \text{ rows i in columns j to be normalized} \tag{3.1}$$

where $x_{i,j}$ is the original value in the i^{th} row and j^{th} column, $m_{i,j}$ is the new value in the i^{th} row and j^{th} column, min_j is the smallest value in the j^{th} column and max_j is the largest value in the j^{th} column.

Z-score normalization is mathematically described as:

$$z_{i,j} = \frac{x_{i,j} - \mu_j}{\sigma_j}, \quad \forall \text{ rows i in columns j to be normalized} \tag{3.2}$$

where $x_{i,j}$ is the original value in the i^{th} row and j^{th} column, $z_{i,j}$ is the new value in the i^{th} row and j^{th} column, μ_j is the mean of all values in column j and σ_j is the standard deviation of all values in column j.

Robust scaling is mathematically described as:

$$r_{i,j} = \frac{x_{i,j} - median_j}{p_{j,75} - p_{j,25}}, \quad \forall \text{ rows i in columns j to be normalized} \tag{3.3}$$

where $x_{i,j}$ is the original value in the i^{th} row and j^{th} column, $r_{i,j}$ is the new value in the i^{th} row and j^{th} column, $median_j$ is the median of the values in the j^{th} column and $p_{j,\alpha}$ represents a value in the j^{th} column that is greater than $\alpha\%$ of all values in that column. Note that the median is the same as $p_{j,50}$.

Equations 3.1 and 3.2 show that min-max normalization and z-score normalization both rely on parameters that are sensitive to outliers and thus can make the normalized dataset skewed. Min-max normalization uses both the minimum and maximum properties of a column that must be normalized. Therefore, min-max normalization is highly sensitive to outliers. As for the z-score normalization technique, it is also sensitive to outliers because it uses the mean and standard deviation properties of a column. However, z-score normalization is usually not as sensitive to outliers as min-max normalization is because a single outlier (a few outliers) has (have) less impact on the mean and standard deviation than it (they) has (have) on the minimum or (and/or) maximum of a column. But anyway, it can be problematic if the outliers are extreme outliers. The third normalization technique that is presented above, the robust scaling, has no parameters that are sensitive to outliers and thus handles datasets with outliers much better than min-max normalization and z-score normalization. The outliers do not disappear, but when using robust scaling, they do not skew the rest of the dataset [37].

Based on the information given about min-max normalization, z-score normalization and robust scaling, it would seem to be appropriate to choose for the robust scaling technique. However, at the time when the federated learning experiments were executed and the robust scaling technique was used on the datasets, during training a sudden drop in the classification accuracy occurred. Until this day it is unclear why exactly this happens. But, that also makes the experiments using robust scaling as normalization technique unusable. However, if z-score normalization is used and further no changes are made to the code, the sudden drop in classification accuracy does not occur. Therefore, z-score normalization is used instead of robust scaling as the second best option. However, even though z-score normalization might not have been the first choice concerning outliers in the datasets, z-score normalization certainly performs good enough for the experiments in this thesis.

The columns that must be normalized, independently of whether z-score normalization or min-max normalization is used, are the following non-binary features from Table 3.5: `duration`, `src_port`, `dst_port`, `packets`, `bytes` and `tos`.

This final step in the preprocessing process concludes everything that is done in this thesis to preprocess the CIDDS-001 datasets. Next section will briefly describe the classification models used in this thesis. As mentioned before, the classification models used are the random forest classifier and a multi-layer perceptron feedforward artificial neural network.

3.2.3 Classification models

Classification models are models that try to predict to which class a set of features/observations belong. This set of features is also called a feature vector. Basically, these models are trained to correctly predict to which class a feature vector belongs by evaluating and adapting the inner parameters of the model while looping through a training set. This training set has the same feature attributes as the feature vectors that a trained model will see once it is put into production. Hopefully, the distribution of the features in the training set resembles real life scenarios so that once a model is trained and has to predict the class of a new observation, it can do so. However, if the training dataset does not (fully) resemble real life distributions, the classification model will not be able to (always) predict correctly.

In case of network intrusion detection systems, the features of network traffic are collected by monitoring the network flows. The goal of intrusion detection systems is then to predict whether the flows it monitors are benign or malicious. And in the case a flow is malicious, hopefully the IDS can determine which attack is being executed. There exist a lot of different classification algorithms: k-Nearest Neighbors, Support Vector Machines, decision trees, random forest classifiers, artificial neural networks, etc.

With all of this in mind, the question asked in this thesis is: “If two network intrusion detection models each have a training dataset that does not fully resemble real life network flow distributions, such as a training set that lacks network flows of a certain attack type, can it learn from another model which did learn to correctly predict that attack type using federated learning?”

In this thesis, two classification algorithms are used: the random forest classification algorithm and a multi-layer perceptron neural network. They are both briefly described in this section.

Random forest classifier

The first type of classification algorithms that is used in this thesis is the random forest classifier. A random forest classifier is a model that is built out of multiple independent decision trees. A decision tree is a very simple tool that can be used to classify a feature vector. An example of a decision tree is shown in Figure 3.2 where the decision tree tries to identify which piece fruit it observes.

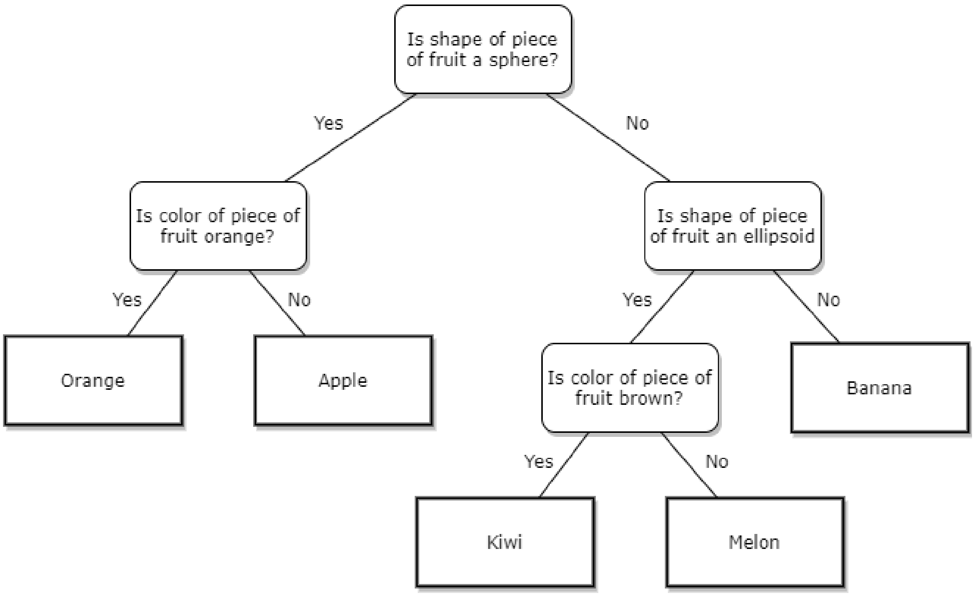

Is shape of piece of fruit a sphere?
Yes
No
Is color of piece of fruit orange?
Is shape of piece of fruit an ellipsoid
Yes
No
Yes
No
Orange
Apple
Is color of piece of fruit brown?
Banana
Yes
No
Kiwi
Melon

Figure 3.2: Example of a decision tree that tries to identify which piece of fruit it observes

The idea behind a random forest is to use multiple such decision trees together and the classification label that gets the most votes becomes the decision of the random forest. So, multiple decision trees work together to come to one conclusive decision. It is, however, important that the decision trees are independent (e.g. by using a different set of attributes, different conditions in each decision node, etc.). Random forests generally outperform single decision trees because an error of one or of some of the forest's decision trees does not necessarily make the decision of the forest erroneous. As long as there are more decision trees making a good decision than there are decision trees making a wrong decision, the final decision of the forest will still be correct. This concept is called the wisdom of crowds, in which the decision trees protect other trees from making individual mistakes [38].

In the experiments that make use of a random forest classifier, the RFC model that is available in the scikit-learn framework is used. When obtaining an object of the class `sklearn.ensemble.RandomForestClassifier`, all default values are kept as they are defined by the scikit-learn developers, except for the maximum depth which is set to 7 (cfr. experiment 1 in section §4.2 for the details).

Multi-layer perceptron neural network

The second technique used in this thesis to predict the attack type of network traffic flows is the multi-layer perceptron neural network. An MLP neural network is one type of artificial neural network. Artificial neural networks are mathematical systems that try to mimic what biological neural networks do. In order to do so, an artificial neural network uses artificial neurons to make calculations that resemble what biological perceptrons do.

An example of what an artificial neuron does is depicted in Figure 3.3. Given a set of input values $[x_0, x_1, x_2, ..., x_n]$, a neuron multiplies each input x_i with a weight w_i followed by an addition of the neuron's bias b. The result of these multiplications and additions is then passed through a non-linear activation function φ. The result of the activation function is the output y of the artificial neuron [39].

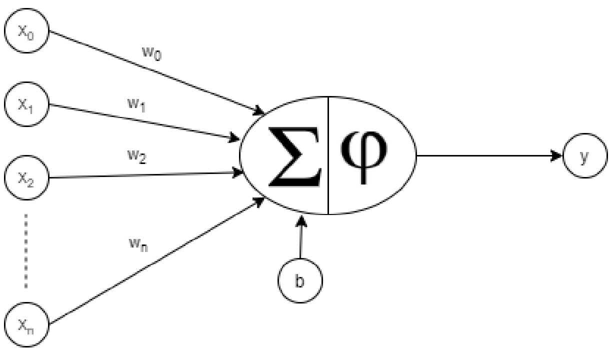


Figure 3.3: Illustration of what an artificial neuron does

In mathematical terms, Figure 3.3 corresponds to:

$$y = \varphi(\sum_{i=0}^{n} w_i x_i + b) \tag{3.4}$$

There are a lot of functions that can be used as an activation function, such as the sign function, the sigmoid function, the hyperbolic tangent, etc. but in this thesis, the Rectified Linear Unit (ReLU) is used. The formula corresponding to the ReLU activation function is given by Equation 3.5.

$$ReLU(x) = max(0, x) \tag{3.5}$$

As mentioned before, artificial neurons are the building blocks of artificial neural networks. Two types of such artificial neural networks are the single-layer perceptron neural networks and the multi-layer perceptron neural networks. A single-layer perceptron neural network consists of three main components: the input layer, a single hidden layer and an output layer. The hidden layer is built out of m artificial neurons that work as described above where each neuron's input is equal to the input nodes of the input layer. The input layer is an array of features, and the output layer is the result of all the calculations done in the neural network. The output layer also exists of a series of artificial neurons and its input nodes are the values that the hidden layer outputs. An illustration of a single-layer neural network is shown in Figure 3.4. The output must be interpreted depending on the use case, but in classification problems, there is an output node for each possible class and the value in an output node tells how certain the neural network is that the input feature vector corresponds to that class. In other words, the output layer is an array of probabilities. But neural networks are not limited to classification problems, they can also be used for generating new music or images, etc. but that is out of the scope of this thesis.

Multi-layer perceptron neural networks are logical extensions of the single-layer perceptron neural network. They are also made up of three components where the input layer and output layer are the same as described in previous paragraph. The difference with the single-layer neural networks is that the multi-layer neural network has multiple hidden layers as opposed to one single hidden layer, as shown in Figure 3.5. All hidden layers in a multi-layer neural network are fully connected, which means that the input vector in hidden layer h_i equals the output vector of layer h_{i-1}, with the exception for hidden layer

h_0 for which the input vector equals the input layer of the whole neural network.

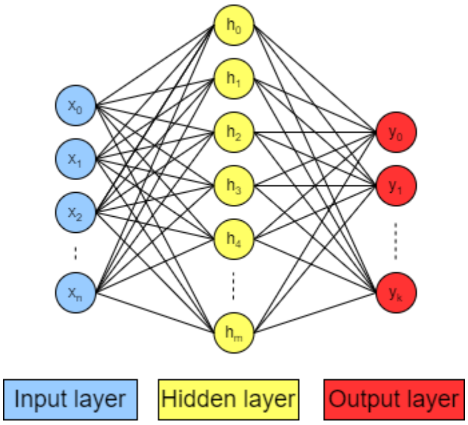


Figure 3.4: Illustration of a single-layer neural network

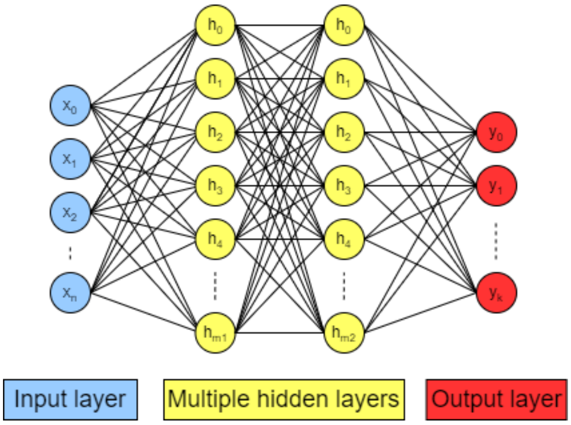


Figure 3.5: Illustration of a multi-layer neural network

Once such a neural network is created and its weights and biases are randomly initialized, the output it would generate once a feature vector is passed through the network, will be meaningless. In order to change the weights of a neural network so that the values it outputs have an interpretable meaning, the neural network needs to be trained. During the training phase, a training dataset is used that contains labeled samples of observations. This way, when the model calculates its prediction on a feature vector in the training dataset, it can compare its own prediction with the label, which is the desired prediction. If the prediction does not match with the label, the model can change its weights and biases slightly so that the prediction on that feature vector would become slightly closer to the desired result.

A neural network can be thought of as an enormous multivariable function. During training, the i^{th} input vector from the training dataset is filled into this function and the output vector o_i is compared to the desired target label vector t_i (i.e. a vector that stores the desired value for each of the output nodes; in classification problems, the value corresponding to the correct class is targeted to become “1” while the other nodes should become “0”). To quantify the error that each output node makes in the current state of the neural network (i.e. with the current weights and biases), the quadratic cost is used to become a quantification of the error that the model makes on the i^{th} input vector of the training dataset:

$$C_i = \frac{1}{2}||t_i - o_i||^2 \tag{3.6}$$

Calculating the cost for each sample in the training set and averaging the costs gives the cost function of the neural network, also known as the error function or loss function:

$$C = \frac{1}{n}\sum_{i=1}^{n} C_i = \frac{1}{2n}\sum_{i=1}^{n}(t_i - o_i)^2 \tag{3.7}$$

where n is the total number of training samples. Note that here, the *mean square error* cost function is presented. This is certainly not the only option when choosing the cost function, some other options are (each having their own properties and use cases): mean absolute error, categorical cross entropy, binary cross entropy... [40].

This (or any other) cost function needs to be minimized so that the model becomes a better predictor. Minimizing a function can be done with the help of derivatives. Since a neural network cannot control what it will see as its input, but it can control what its internal weights and biases are, the gradient of the cost function is calculated with respect to the weights and biases. Calculating the gradient will tell how the cost function changes when the weights and biases of the model are changed. Calculating the gradient of the cost function is done with an algorithm called backpropagation [41, 42]. How this exactly works is out of the scope of this thesis. The key takeaway is that backpropagation eventually comes up with the gradient of the cost function. Once the gradient is known, another algorithm called gradient descent is used to find out how the weights and biases must move in order to minimize the cost function. The specifics of the algorithm are also out of scope for this thesis. However, some final words about gradient descent are following. It is mentioned that the cost function is calculated on all training samples. This would result in enormous calculations. Therefore, instead of calculating the cost function over the complete training dataset at once, the training dataset is divided into random batches that are smaller in size. The process of how the neural network is being trained as explained before is now used on each batch instead of using the whole training dataset at once. This approach is called stochastic gradient descent [43] in which backpropagation is still used.

For the experiments in this thesis, the MLP neural network that is described in the paper [36] is used. To obtain an object that represents such an MLP, the framework TensorFlow is used. TensorFlow provides a class `tf.keras.Sequential` with which each layer of the MLP neural network can be built. The detailed overview of the MLP of paper [36] can be found in Table 3.6.

Table 3.6: Overview of the layers and its configurations of the MLP used in this thesis

Keras layer	Number of nodes	Activation	Dropout
Input	16	-	-
Dense	100	ReLU	0.2
Dense	100	ReLU	0.2
Dense	5	SoftMax	-

The input layer has 16 nodes since the CIDDS-001 flows have 16 features after preprocessing as described in section §3.2.2 and as can be seen in Table 3.5. The input values are not passed through any activation function and are directly passed through to the first hidden layer. There are three dense layers in the MLP which represent fully connected layers. As mentioned before, that means that each of the dense layers receives all values of the nodes of the previous layer as its input vector. The first and second dense layers represent the two hidden layers of the multi-layer perceptron neural network and use the rectified linear unit function as their activation function, which is described above in Equation 3.5. Each of the hidden layers is followed by a dropout layer which is a layer that is only used during training in order to prevent the model from overfitting the training data. Creating an overfitted neural network is bad for generalization since the model then learns the noise in the training data which is not present in new data. An overfitted model might perform great on the training data, but not on new data (e.g. on a test set). To prevent this, during training, the results of the neurons in the hidden layers are randomly being dropped out. The rate at which they are dropped is given in the table. A dropout of 0.2 means that a node in the hidden layer has 20% chance of being dropped out at each step during training. Dropping out means setting the value of a neuron to 0. If a neuron gets a zero value, this is effectively the same as dropping the size of that layer with one, creating a new model configuration. By doing so, the MLP in its whole does not get the chance to optimize itself for picking up the noise in the training dataset [44]. Finally, the last layer, which is the output layer, is also fully connected with its previous layer and has 5 nodes. The reason for the five nodes is that there are five different attack types in the CIDDS-001 dataset: `Normal`, `BruteForce`, `DoS`, `PingScan` and `PortScan`. The activation function in the output layer is not ReLU but SoftMax in order to obtain the probabilities. The SoftMax activation function maps any real number to a value between 0 and 1. The mathematical description of the SoftMax activation function is:

$$SoftMax(z)_i = \frac{e^{z_i}}{\sum_{j=0}^{K} e^{z_j}} \forall i = 1, ..., K \tag{3.8}$$

where z is the output vector before performing the activation function, K is the size of that output vector (in this case, $K = 5$). The SoftMax function takes a set of numbers which it normalizes into a set of probabilities [45].

The output of this MLP is interpreted as follows: the output node that has the greatest probability will be considered the prediction of the MLP.

Finally, as mentioned before, TensorFlow is used to create a `tf.keras.Sequential` object representing the multi-layer perceptron neural network as described in Table 3.6. The code for creating such an object is given in Listing 2.

```python
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(16,)),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(5, activation='softmax')
])
```

Listing 2: Code snippet used to create the multi-layer perceptron neural network used in this thesis

At this point, all experimental prerequisites are discussed. This chapter has described what steps are taken to preprocess the CIDDS-001 datasets and it also discusses the algorithms that are used in the experiments. In the next chapter, the experiments themselves are discussed.

4

Experiments

4.1 Introduction to the experiments

In chapters 1 (Introduction) and 3 (Experimental Prerequisites), the goals of this thesis were already put forward. It comes down to answering the question: “Can two network intrusion detection systems that each have their own local training datasets learn from each other when one training dataset lacks captured flows from a certain attack type while the other training dataset does not?” Chapter 3 (Experimental Prerequisites) already explained that datasets *internal week 1* and *internal week 2* are used in which captured network flows of normal traffic and traffic from four different attack types (DoS, brute force, ping scan and port scan) are stored. If one classification model is trained with data coming from *internal week 1* in which flows of a certain attack type (e.g. DoS) are removed and the other classification model is trained with data coming from *internal week 2* in which flows of another attack type (e.g. brute force) are removed, is it then possible that the first and second classification models can learn from each other? In the experiments described in this chapter, federated learning is used to try to make this possible.

There are seven different experiments, each of which is either based on the one before or is introduced because the results of an experiment led to new questions. All of the experiments are available in a public GitHub repository that can be found at `https://github.com/bravdwal/MScProject`. The experiments themselves reside in a directory `experiments` which contains three subdirectories `01 Base`, `02 TF` (TF standing for TensorFlow) and `03 TFF` (TFF standing for TensorFlow Federated). In each of these subdirectories, the Jupyter notebooks containing the Python code of the experiments can be found. In the names of these Jupyter notebook files, there is always a version number: v2.1, v2.2, v3.1, etc. These version numbers will be referenced to throughout the chapter to indicate which notebook is being discussed.

All of the experiments start out with preprocessing the CIDDS-001 datasets *internal week 1* and *internal week 2*. Section §3.2.2 already discussed the different preprocessing steps taken before every experiment. However, in order to not always have to execute every single preprocessing step, cleaned up versions of the datasets *internal week 1* and *internal week 2* are generated and saved in feather files[1]. The preprocessing steps that are executed to come to the "cleaned up version" of a CIDDS-001 dataset consist of the following steps from section §3.2.2: (1) Remove abnormalities, (2) One-hot encode categorical attributes and (3) Unfold OR-concatenated attributes. Also, the column header names of the original datasets are changed to the names given in Table 3.5. This also means that the preprocessing steps (4) Drop some columns for classification and (5) Normalize non-binary columns must be executed per experiment. This is necessary since the details may be different in some experiments as opposed to in others. For creating this cleaned up version of the CIDDS-001 datasets, an auxiliary method `clean_cidds_001()` is provided in the repository in the file `utils/cidds_001.py`. It expects a `pandas.DataFrame` object as parameter holding the data of a CIDDS-001 dataset. This auxiliary method performs the three preprocessing steps and returns a new `pandas.DataFrame` object containing the new cleaned up version of the original CIDDS-001 dataset. This new dataset can then be saved in a feather file using the method `pandas.DataFrame.to_feather()`. So, the first thing that is done in each experiment is loading the two cleaned up datasets. Then, the preprocessing phase is completed by executing the remaining two preprocessing steps (i.e. (4) Drop some columns for classification and (5) Normalize non-binary columns). When the preprocessing phase has succeeded, the two datasets are balanced out, except for one of the experiments in which the datasets are not balanced out. For this task too, an auxiliary method `get_balanced_cidds()` is provided in `utils/cidds_001.py` which expects a `pandas.DataFrame` object containing a CIDDS-001 dataset and returns

[1] Feather is an optimized binary file format that can be used to store data frames [46]

a balanced version of the dataset with respect to the attribute `attack_type` since this attribute will be the classification label. Finally, the two balanced datasets are split into two training and two testing datasets that can be further used in the experiment.

If in any case, the first steps taken in an experiment do not match with what is described in the previous paragraph, the changes will be clearly stated in the corresponding section.

In this thesis, two appendices which are related to the experiments are included. In Appendix A, the confusion matrices of the results of all experiments are provided. In Appendix B, the histograms of the distributions of the flows with respect to the classification label "AttackType" of all datasets that are used throughout the seven experiments in this thesis are presented.

4.2 Experiment 1: Getting to know RFC

This experiment corresponds to the notebook in the GitHub repository with version v2.1.

The first experiment is a predecessor of the so called "base experiment". The base experiment will be used to compare the results of the federated learning experiments with. This will make it possible to validate the results of the experiments that provide an answer to the question whether or not two classification models can learn from each other in the context of network intrusion detection. The base experiment uses a random forest classifier and before the actual base experiment is executed, the simplest classification problem concerning the CIDDS-001 *internal week 1* and *internal week 2* datasets is solved. In other words, the datasets are used as they are after preprocessing and balancing the datasets and the RFC models are trained and tested with these datasets. This is done to get a feel for both the random forest classifier and the datasets.

The experiment starts with loading the cleaned up versions on the *internal week 1* and *internal week 2* datasets via reading the feather files with `pandas.read_feather()`. The `pandas.DataFrame` objects are then shuffled in order to randomize the datasets. However, the shuffling is seeded with a random state set to 13 for reproducibility purposes. After shuffling the datasets, they are balanced out and unused columns are dropped. Note that no normalization step is introduced because the classification model used in this experiment is an RFC model. An RFC uses decision trees and thus it is not necessary to introduce a normalization step.

At this point, the *internal week 1* and *internal week 2* datasets are preprocessed and balanced out. For this experiment, balancing out a dataset means creating a new dataset that contains all network flows of the lowest represented attack type and an equal amount of network flows of all other attack types. In the previous chapter, in section §3.2.1, Table 3.2 gave a detailed overview of the number of different flows for each attack type that are available in the unmodified CIDDS-001 *internal week 1* and *internal week 2* datasets. In *internal week 1*, the attack type `BruteForce` has the lowest number of captured network flows with 1,626 available flows. In *internal week 2*, the attack type `PingScan` has the lowest number of captured network flows with 2,731 available flows. Therefore, in the balanced datasets, for all other attack types, respectively 1,626 and 2,731 network flows are randomly chosen. This makes the content of the datasets used in this experiment as presented in Table 4.1.

Table 4.1: Distribution of flows with respect to the attribute “attack type” for the datasets used in Experiment 1

Attack type	Number of flows in preprocessed dataset *internal week 1*	Number of flows in preprocessed dataset *internal week 2*
Normal	1,626	2,731
BruteForce	1,626	2,731
DoS	1,626	2,731
PingScan	1,626	2,731
PortScan	1,626	2,731

Using these two datasets, the features (i.e. first 16 rows in Table 3.5) are separated from the labels (i.e. the attribute `attack_type`) and these features and labels are further separated in training and testing datasets using the auxiliary method `sklearn.model_selection.train_test_split()`. After this, the datasets are fully prepared to be used for training and testing the RFC models.

The next step in this first experiment is creating two RFC models. The first RFC model `rfc_week1` will be trained with the balanced dataset coming from *internal week 1* while the second RFC model `rfc_week2` will be trained with the balanced dataset coming from *internal week 2*. The machine learning library `scikit-learn` provides a class that implements the RFC model with the class `sklearn.ensemble.RandomForestClassifier`. This implementation of the RFC model is used in this thesis. To obtain an object of the RFC model, the constructor of the class `sklearn.ensemble.RandomForestClassifier` is called without modifying the default values for the parameters, except for the maximum depth (cfr. infra). It will turn out that the accuracy of the trained model is high enough. Therefore, the other hyperparameters (e.g. the number of decision trees which defaults to 100, the minimum number samples that need to be in a leaf which defaults to 1, etc. [47]) do not need to be modified/optimized.

Using the default value for the hyperparameter "maximum depth" (i.e. `None`), is not a good idea because this means that an individual tree can have unlimited depth until all leaves contain samples of the same class, or the leaves contain less than the minimal number of samples required to split the leaf (which defaults to 2) [47]. This may result in the individual trees to overfit the training data since having an "unlimited" depth can result in the RFC model to exactly tweak the trees to fit the training data perfectly. This is bad for generalization. Therefore, this first experiment was executed four times[2] where the maximum depth was either equal to `None`, 5, 7 or 10. The precision scores of the tests for which the maximum depth was equal to `None`, 7 and 10 were very similar compared to each other while the results when the maximum depth was equal to 5, were significantly lower. Hence, the maximum depth of the RFC models is chosen to be 7: the depth is not too high so that overfitting is prevented but the scores are still high.

Once `rfc_week1` and `rfc_week2` are trained with the training set based on *internal week 1* and the training set based on *internal week 2* respectively, each model is tested twice. The first test is letting the trained models predict the labels of their test sets. The second test is letting the trained models predict the labels of all samples in the CIDDS-001 dataset it has not yet been confronted with. This means that `rfc_week1` will be tested with the dataset *internal week 2* and `rfc_week2` will be tested with the dataset *internal week 1*. The precision scores[3] of these tests can be found in Table 4.2.

It is clear by having a look at the results in Table 4.2 that the RFC model scores quite good. In Table 4.2, only the precision metric is given but the full confusion matrix can be found in Appendix A. The scores for the attack types ping scan and port scan are a bit lower compared to the other attack types, but they still have a good score of more than 90%.

This first experiment showed that even with a pretty small dataset, due to balancing out the number of flows with respect to the attack type, an RFC model still scores quite good on the CIDDS-001 datasets *internal week 1* and *internal week 2*.

[2] The results of each of the tests are available in the public GitHub repository.

[3] A precision score is the percentage of flows of an attack type that are predicted correctly

Table 4.2: Precision scores of trained RFC models in Experiment 1 (all values in percentages)

Attack type	Results of **rfc_week1**		Results of **rfc_week2**	
	Testing set week 1	Full dataset week 2	Testing set week 2	Full dataset week 1
Normal	99.40	99.72	98.75	98.93
BruteForce	94.21	86.04	100.0	96.37
DoS	100.0	99.96	100.0	99.98
PingScan	92.83	90.85	91.07	96.58
PortScan	96.94	92.60	91.29	95.60

4.3 Experiment 2: Base experiment

This experiment corresponds to the notebook in the GitHub repository with version v2.2.

The second experiment is called the "base experiment" since the results of this experiment will later be used to validate the results of the federated learning experiments. Analyzing the results of the federated learning experiments and comparing them with the results of this base experiment, allows for answering the question whether or not classification models can learn from each other in the context of network intrusion detection.

This base experiment is very reminiscent of the first experiment. The only difference between the two experiments is the beginning and the results. After loading the cleaned up versions of *internal week 1* and *internal week 2*, there is an extra step in comparison to experiment 1. In the cleaned up version of *internal week 1*, all of the flows of the attack type "brute force" are removed from the dataset. Analogously, in the cleaned up version of *internal week 2*, all of the flows of the attack type "ping scan" are removed. There is no specific reason why the attack types "brute force" and "ping scan" are chosen, this is just a random selection of two (different) attack types.

The reason for removing all these flows is to see what the effect is on the results when the trained models are tested. Although the answer will not be surprising, seeing the effect in the results will be important to validate whether or not federated learning allows to let models learn to classify classes it has not available in its local dataset.

Once these flows are removed, the datasets are balanced out, taking into account the lowest represented attack types (excluding the removed attack types). Table 3.2 shows that the lowest represented attack types are "ping scan" for *internal week 1* and "brute force" for *internal week 2* once the flows “brute force” and “ping scan” are removed, respectively. Hence, for the other attack types, 3,359 flows from the preprocessed dataset of *internal week 1* and 3,366 flows from the preprocessed dataset of *internal week 2* are randomly chosen. The distribution with respect to the attack type in the balanced datasets for experiment 2 becomes as shown in Table 4.3.

Further, just like in experiment 1, the unused columns are dropped, the features are separated from the labels and the two training sets and two test sets are derived from these features and labels. This makes sure that there are a training set and test set available based on the preprocessed dataset of *internal week 1*, as well as a training set and test set based on the preprocessed dataset of *internal week 2*.

Table 4.3: Distribution of flows with respect to the attribute “attack type” for the datasets used in Experiment 2

Attack type	Number of flows in preprocessed dataset *internal week 1*	Number of flows in preprocessed dataset *internal week 2*
Normal	3,359	3,366
BruteForce	0	3,366
DoS	3,359	3,366
PingScan	3,359	0
PortScan	3,359	3,366

The rest of this experiment is the same as experiment 1: two RFC models are created using `pandas`’ class `sklearn.ensemble.RandomForestClassifier`. The model `rfc_week1` is trained with the preprocessed dataset that is based on *internal week 1* while the model `rfc_week2` is trained with the preprocessed dataset that is based on *internal week 2*. When the models are trained, they are tested with two test sets: once with the test set that comes from the same week and once with the complete dataset of the other week. Concretely, `rfc_week1` is tested with the test set based on *internal week 1* and also with the full dataset *internal week 2*. The trained model `rfc_week2` is tested analogously. The precision scores of the results of these tests are available in the Table 4.4. The full confusion matrix for the tests are available in Appendix A.

As mentioned before, the results are not surprising. There are no brute force and ping scan scores available for the tests on the test sets of the models since the single flow samples of brute force and ping scan were already present in the training datasets and

Table 4.4: Precision scores of trained RFC models in Experiment 2 (all values in percentages)

Attack type	Results of **rfc_week1**		Results of **rfc_week2**	
	Testing set week 1	Full dataset week 2	Testing set week 2	Full dataset week 1
Normal	100.0	99.90	98.53	98.89
BruteForce	n/a	0.000	99.89	96.25
DoS	99.96	99.92	100.0	99.97
PingScan	97.02	90.92	n/a	0.000
PortScan	95.73	92.63	95.83	98.22

thus not available in the test sets. On top of that, `rfc_week1` has not correctly predicted a single flow of the brute force flows of the dataset *internal week 2*. Similarly, `rfc_week2` has not correctly predicted a single flow of the ping scan flows of the dataset *internal week 1*. All other precision scores are very good, which is to be expected since in experiment 1, the scores were high too. As said before, although these results are predictable, the value of this experiment is that now the results of the federated learning experiments are comparable with a scenario in which two models are not trained using federated learning. If, after federated learning, the two models are able to have a higher precision score for respectively "brute force" and "ping scan" when tested with the full dataset of the "other week", then the conclusion will be that federated learning indeed allows for two classification models to learn from each other in the context of network intrusion detection, without having to share the local datasets.

Mapping this experiment to a real life scenario comes down to an organization having two geographically separated networks with each having its own intrusion detection system. Each IDS is trained with its own local dataset. Let's presume that network 1 has `rfc_week1` in its IDS, then it will be able to recognize network traffic corresponding to the attack types DoS, ping scan and port scan, but it will not be able to recognize flows of brute force attacks. At the same time, if the second network has `rfc_week2` in its IDS, the IDS will recognize flows of brute force, DoS and port scan attacks, but it will not be able to recognize flows of ping scan attacks. In such case, it is a pity that the IDS of network 1 cannot detect brute force attacks while the IDS of network 2 does, and vice versa with ping scan attacks. A solution could be, of course, to exchange the local datasets so that both IDSs possess flows of all attack types. However, as discussed in 2.2, this is not always feasible due to e.g. GDPR constrains or other security policies since sharing captured network flows may violate privacy restrictions, but it can also be too costly (computationally or too much use of the available bandwidth) to send the local training data and retrain the models. The question is, as mentioned already a few times by now: "Can this limitation be overcome using federated learning by which not the local training datasets are exchanged but rather the model parameters?"

4.4 Intermezzo 1: Federated averaging

For the federated learning experiments, TensorFlow Federated (TFF) will be used. This is an open-source framework provided by TensorFlow which allows performing machine learning on decentralized data [48]. The framework provides an implementation of the federated averaging algorithm which is described in [28] and which will be used in he federated learning experiments. The downside of this approach is however that Experiments 1 and 2 use RFC models but TFF does not support the use of these models in the federated learning algorithms. Therefore, before the actual federated learning experiments can be discussed, a new type of classification model must be introduced. For this, the multi-layer perceptron neural network will be used as a classifier, as discussed in section §3.2.3. The two upcoming experiments 3 and 4 are used to introduce this new classifier in order to get familiarized with the MLP model.

However, since the federated averaging algorithm is mentioned in the previous paragraph, and since it is used in experiments 5, 6 and 7, a summary of this algorithm is given here already [49]. First of all, the server (a node in the federated learning setup that gets assigned this role) that keeps track of the global MLP model and updates the weights using the federated averaging algorithm, assigns random weights to a new MLP model. Then, T iterations of federated learning are executed:

1. The server distributes the weights of the current global model to all the clients participating in the federated learning setup.

2. Each client updates its local MLP model according to the weights it received from the server.

3. Each client starts training its local MLP model using the local training dataset. As a consequence, the clients will update the weights of their local MLP model while trying to minimize the local cost function.

4. Each client sends the weights of its local MLP model to the server.

5. The server aggregates all the weights it receives from the clients using the federated averaging algorithm, which is mathematically described in equation 4.1 (*source*: [28]).

$$w_{t+1} = \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^{k} \tag{4.1}$$

where w_{t+1} resembles the updated weights matrix of the global MLP after t federated learning iterations, K is the number of participating clients, w_{t+1}^{k} resembles the weights matrix of the local MLP of client k after training in the t^{th} iteration, n_k is the number of data samples in the local training set of client k and n is the total sum of the sizes of all local training datasets. The fraction $\frac{n_k}{n}$ is introduced in order to get a weighted averaging according to how large the training sets are of each client [28].

Note the terminology “server” and “client”. A server is a node that is assigned with the task of aggregating the weights of all local MLPs of participating clients and redistributing the weights of the updated global MLP model after the aggregation (i.e. federated averaging in this thesis). In the context of network intrusion detection, all participating IDSs are clients. One of these IDSs can also be given the role of server but that role must not necessarily be given to an IDS. A dedicated computer can also be given the role of server in the federated learning setup.

4.5 Experiment 3: Getting to know MLP and TensorFlow

This experiment corresponds to the notebook in the GitHub repository with version v3.1.

As mentioned in Intermezzo 1, before continuing to the federated learning experiments, a new classification model must be introduced. The classification model that is used in this experiment, is already discussed in the previous chapter (Experimental Prerequisites) in section §3.2.3. As a quick recap of what layers are used in the MLP discussed in §3.2.3, the code listing for creating the model is repeated in 3.

The MLP model contains an input layer that has 16 nodes, one for each feature in the captured network flows (after preprocessing cfr. features in Table 3.5), two hidden layers

```python
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(16,)),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(5, activation='softmax')
])
```

Listing 3: Code snippet used to create the multi-layer perceptron neural network used in this thesis (repetition from Listing 2)

(i.e. the two middle `tf.keras.layers.Dense` objects) and an output layer that has five nodes. Each output node contains a probability that corresponds with the certainty that the model thinks the corresponding attack type is the label of the feature vector. The output node that has the largest probability will eventually be the final prediction of the MLP model.

In order to get familiar with this new classification model, the experiment described in paper [36] is repeated in this experiment. First of all to get familiar with using an MLP model in the TensorFlow framework, but also to see if the new classification model performs as good as the RFC model in experiments 1 and 2.

N. Oliveira et al. discuss in their paper [36] the difference between single-flow and multi-flow network intrusion detection. In this experiment, only the single-flow approach is imitated. Also, N. Oliveira et al. test each approach with three different classification models: an LSTM (long short-term memory) neural network, an MLP neural network and an RFC classifier. Since at this point only the MLP is of interest, only that model will be used in this experiment.

This experiment also uses the cleaned up versions of the datasets *internal week 1* and *internal week 2* but in a different way as used in experiments 1 and 2. As described by N. Oliveira et al. in [36], the captured network traffic that was recorded between 2017/03/17 14:18:05 and 2017/03/20 14:42:17 is isolated from *internal week 1*. No traffic between these timestamps resides in *internal week 2* since the traffic in this dataset is recorded in the week after the capturing of network traffic for *internal week 1*. Hence, no flows from *internal week 2* are isolated. The distribution of the flows in the isolated selection with respect to the attack type is shown in Table 4.5. The isolated selection from *internal week 1* will be used as the training data for the MLP model.

Table 4.5: Distribution of flows with respect to the attribute "attack type" for the datasets used in Experiment 3

Attack type	Number of flows in the isolated selection
Normal	2,092,587
BruteForce	1,262
DoS	390,932
PingScan	1,068
PortScan	50,136

As can be seen in the flow distribution, this training dataset is very much imbalanced. However, this is how N. Oliveira et al. executed their experiments so this dataset will be used as training set in this experiment.

Further, just as in experiments 1 and 2, the unused columns are dropped and the features are separated from the labels. After that, another step is added in the preprocessing phase of the experiment as compared to experiments 1 and 2: feature normalization. This was not required in the experiments where RFC models are used, but it is necessary for the MLP model. Since N. Oliveira et al. used min-max normalization in their experiment, min-max normalization is also used here in the third experiment on the isolated selection from *internal week 1*.

At this point, the training set is prepared to be used to train the MLP. When the model is trained, it is tested with the cleaned up dataset *internal week 2*. Before the testing starts, the dataset is also preprocessed just like the training dataset is preprocessed (i.e. dropping unused columns, splitting features from labels and normalizing the features). The results of the test are shown in Table 4.6 by giving the precision scores of the predictions of the trained model (again, the confusion matrix of this experiment can be found in Appendix A).

The attack types for which there were enough samples in the training dataset show similar precision scores as seen in experiments 1 and 2. However, the imbalance is too big for the brute force and ping scan attacks to be recognizable after training. Therefore, in order to be certain that the MLP model as described in [36] is a good choice for the federated learning experiments (i.e. it must have similar precision scores as seen in experiments 1

Table 4.6: Precision scores of trained MLP model in Experiment 3 (all values in percentages)

Attack type	Results of the MLP model
Normal	99.56
BruteForce	6.179
DoS	99.92
PingScan	0.1831
PortScan	90.21

and 2), a second test with the MLP is executed before starting the federated learning experiments, but this time with a balanced training dataset. This will be discussed in Experiment 4.

Note that in Table 4.6, there is no comparison between the results obtained in this experiment and the results obtained by N. Oliveira et al. The reason for this is that N. Oliveira et al. do not clearly state what data is used for training and what data is used for testing. They mention the use of *internal week 1* and *internal week 2*, but the dates by which their isolated selection of data is defined, both fall within the time frame of week 1. Before starting with the single-flow experiment using the MLP model, N. Oliveira et al. mention the use of the "CIDDS-001 preprocessed data" for training and testing the MLP model. Because the use of the datasets *internal week 1* and *internal week 2* is mentioned, but the dates indicate only the use of *internal week 1*, it is not 100% clear what is meant with "the CIDDS-001 preprocessed data". Therefore, directly comparing the results of experiment 3 with the results of N. Oliveira et al.'s experiment, may not be appropriate. On top of that, they do not numerically specify the scores of the results for each attack type since the results are shown in a histogram. Also, the metric (i.e. the F1-score) that is used differs from what is used in Table 4.6 (i.e. precision). However, the results of N. Oliveira et al. also show high scores for the attack types "Normal" (F1-score of about 100%), "DoS" (F1-score of about 100%) and "PortScan" (F1-score of about 98%) while the scores are lower for "BruteForce" (F1-score of about 80%) and "PingScan" (F1-score of about 0%). But it has to be noted that the results for "BruteForce" differ quite a bit between the third experiment of this thesis and the experiment executed by N. Oliveira et al.

4.6 Experiment 4: Methodological improvement by retraining MLP with a balanced training dataset

This experiment corresponds to the notebook in the GitHub repository with version v3.2.1.

Because the results of experiment 3 suffered from the imbalance in the training set, a second experiment is introduced in order to test whether or not the MLP model described in [36] can show similar results as seen in experiments 1 and 2.

The problem in experiment 3 was the imbalance in the training dataset. In order to solve this problem, and to obtain as much training data for each attack type as possible, the cleaned up versions of *internal week 1* and *internal week 2* are jointly used to obtain a new baseline dataset which is more or less balanced. Since the ping scan and brute force attacks have the least amount of flows in the datasets *internal week 1* and *internal week 2*, all of the flows of these attacks from both datasets are used. For all other attack types, $3,000$ flows from each dataset are randomly chosen to be put in the new balanced dataset. This makes the distribution of flows with respect to the attack types in the new balanced dataset as shown in Table 4.7.

Table 4.7: Distribution of flows with respect to the attribute “attack type” for the dataset used in Experiment 4

Attack type	Number of flows coming from *internal week 1*	Number of flows coming from *internal week 2*	Total number of flows in the resulting balanced dataset
Normal	3,000	3,000	6,000
BruteForce	1,626	3,366	4,992
DoS	3,000	3,000	6,000
PingScan	3,359	2,731	6,090
PortScan	3,000	3,000	6,000

This new balanced dataset contains flows from the cleaned up versions of *internal week 1* and *internal week 2*. This means that before the flows can be used for training and testing, the unused columns must be removed, the features must be separated from the labels and the features must be normalized. The normalization technique that is used in this experiment is z-score normalization as opposed to min-max normalization in experiment 3. The change of normalization techniques is motivated by the fact that z-score normalization will be used further in the federated learning experiments as discussed in §3.2.2. After this preprocessing phase, the preprocessed dataset is split into a training and testing dataset (80% for the training dataset and 20% for the testing dataset). Then, the MLP model is created and trained with the training dataset. When the model is trained, it is tested with the testing dataset. The precision scores of the results of the test are shown in Table 4.8 (and the confusion matrix is given in Appendix A).

Table 4.8: Precision scores of trained MLP model in Experiment 4 (all values in percentages)

Attack type	Results of the MLP model
Normal	90.33
BruteForce	97.15
DoS	100.0
PingScan	94.97
PortScan	85.65

The prediction scores of the MLP model now do approach the scores seen in experiments 1 and 2 much better than seen in experiment 3 and thus, this model will be used further for the federated learning experiments.

4.7 Intermezzo 2: TensorFlow Federated

Before continuing to the fifth experiment, an intermezzo is presented to highlight the key points of how TensorFlow Federated works. The federated learning flow that is used in this thesis is an adaptation from the tutorials that are provided by TensorFlow Federated [50, 51]. Before stepping into the actual experiments, first a summary is given of how a federated learning setup can be created with TensorFlow Federated and how the setup can start training.

First of all, it has to be noted that the federated learning setup used in TensorFlow Federated is currently not a really distributed setup since the experiment is run on a single host. This means that all resources are actually available on the host, but TensorFlow Federated isolates the clients and the server processes behind the scenes in order to mimic the distributed setup. In TensorFlow Federated, the federated learning setup is represented by a `tff.templates.IterativeProcess` object. This object contains the functionality to support the execution of the tasks of the clients and servers, where each task is an isolated process in order to imitate the distributed properties of federated learning. For this, the `tff.templates.IterativeProcess` uses an object of the class `tff.learning.framework.ServerState`. This object knows how to create a classification model used by the clients and keeps track of the current weights of the global classification model. At the start of each round/iteration of federated learning (which is started by calling `next()` on the `tff.templates.IterativeProcess` object), the `tff.learning.framework.ServerState` object distributes the current weights of the global classification model to the clients. Then, the `tff.templates.IterativeProcess` lets the clients train their local models initialized with the new weights on their local training datasets. When all clients have finalized their training phase, the new weights of each of the clients are sent back to the `tff.learning.framework.ServerState` object which then starts the aggregation in order to obtain the new weights of the global classification model [52, 53, 54].

The way TensorFlow Federated handles the configuration of the clients of a federated learning setup is the easiest. Essentially, a client consists of a local training dataset and a model that uses this local training dataset to get trained. In TensorFlow Federated, such a local training dataset is represented by a `tf.data.Dataset` object which contains the features and labels of all the clients' local training data. Such a `tf.data.Dataset` object must be created by the user of TensorFlow Federated. However, the classification models of the clients are managed by the `tff.templates.IterativeProcess` object that represents the federated learning setup. Creating the `tf.data.Dataset` can be easily done with an auxiliary method provided by TensorFlow Federated. If `x` is a NumPy[4] array containing the features of all samples of a local training dataset and `y` is a NumPy array containing the corresponding labels, then a `tf.data.Dataset` object can be created as shown in the following code listing:

```python
# create a local training dataset for a federated learning client
client_dataset = tf.data.Dataset.from_tensor_slices((x, y))
```

Listing 4: Code snippet showing how to create the local training dataset of a client

Creating the object representing the federated learning setup (i.e. the `tff.templates.IterativeProcess` object) is a bit less straightforward as opposed to creating the object representing a local training dataset. First of all, the MLP model that is created as a `tf.keras.Sequential` object in experiments 3 and 4 must be converted into a `tff.learning.Model` object. Luckily, TensorFlow Federated provides a function for this task: `tff.learning.from_keras_model()`. But first, an auxiliary function is provided for creating a keras model (i.e. the `tf.keras.Sequential` as used in experiments 3 and 4) representing the MLP model:

```python
def create_keras_model():
    return tf.keras.Sequential([
        tf.keras.layers.Input(shape=(16,)),
        tf.keras.layers.Dense(100, activation='relu'),
        tf.keras.layers.Dropout(rate=0.2),
        tf.keras.layers.Dense(100, activation='relu'),
        tf.keras.layers.Dropout(rate=0.2),
        tf.keras.layers.Dense(5, activation='softmax')
    ])
```

Listing 5: Code snippet showing the auxiliary function that creates a keras model representing the MLP model

Then, using this function, a similar function is defined that creates a `tff.learning.Model` object representing the MLP model that can be used by TensorFlow Federated:

[4] NumPy is a Python library for scientific computing [55]

```
def model_fn():
    keras_model = create_keras_model()
    return tff.learning.from_keras_model(
        keras_model,
        input_spec=tff_input_element_spec,
        loss=tf.keras.losses.CategoricalCrossentropy(),
        metrics=[tf.keras.metrics.CategoricalAccuracy(),
            tf.keras.metrics.CategoricalCrossentropy()]
    )
```

Listing 6: Code snippet showing the auxiliary function that creates the MLP model that can be used by TensorFlow Federated

Once such a function is provided, the `tff.templates.IterativeProcess` object that represents the federated learning system can be created. TensorFlow Federated provides a few functions that can create such a `tff.templates.IterativeProcess` object, each with their own aggregation algorithms. In this thesis, the federated averaging algorithm is used as aggregation algorithm as described in Intermezzo 1. Therefore, the corresponding auxiliary function provided by TensorFlow Federated is used:

```
iterative_process = tff.learning.build_federated_averaging_process(
    model_fn=model_fn,
    client_optimizer_fn=lambda: tf.keras.optimizers.SGD(learning_rate=0.02),
    server_optimizer_fn=lambda: tf.keras.optimizers.SGD(learning_rate=1.0)
)
```

Listing 7: Code snippet showing how an object is created that represents the federated learning system used in this thesis

This `iterative_process` object can now be used to start as many iterations of federated averaging as needed by calling the function `next()` on this object. This `next()` function expects two parameters: the first parameter is expected to be an object representing the current `tff.learning.framework.ServerState` and the second parameter is an array of all the `tf.data.Dataset` objects representing the local training datasets of the clients. The first `tff.learning.framework.ServerState` object is obtained by calling the `initialize()` function on the `iterative_process` object as shown in following code listing:

```
# initializing a state (i.e.\ a tff.learning.framework.ServerState object)
state = iterative_process.initialize()

# from now on, each federated learning iteration returns the new state and
# some metrics obtained during training
state, metrics = iterative_process.next(state, [client_dataset1, client_dataset2,...])
```

Listing 8: Code snippet showing how a server state is initialized, and how to execute a single federated learning iteration

To test the model that corresponds to the global model of which the weights are managed by the `tff.learning.framework.ServerState` object, the weights can be assigned to a `tf.keras.Sequential` model. This model can then be used to test the weights trained by the federated learning system. Assigning the weights to a `tf.keras.Sequential` model can be done as shown in following code snippet:

```
model = create_keras_model()
state.model.assign_weights_to(model)
```

Listing 9: Code snippet showing how the weights of a server state object can be assigned to a keras model

4.8 Experiment 5: Getting to know TensorFlow Federated

This experiment corresponds to the notebook in the GitHub repository with version v4.1.

Experiment 4 showed that the MLP model described in [36] gives sufficient precision scores when using a balanced dataset. Therefore, in this experiment, federated learning is introduced using the TensorFlow Federated framework in combination with the aforementioned MLP model. Intermezzo 2 already described how to create and use a federated learning system with the framework TensorFlow Federated. In this fifth experiment, such a federated learning system is used in which two clients participate in federated learning. For each of the clients a `tf.data.Dataset` object is created that represents their local training datasets. These training datasets are, for now, balanced datasets based on the cleaned up version of the datasets *internal week 1* and *internal week 2*. The balanced datasets thus have a distribution with respect to the attack type as seen in experiment 1 (cfr. Table 4.1). From these balanced datasets, the unused columns are removed and the features are separated from the labels. Since federated learning is done with the MLP model, and not with an RFC model as in experiments 1 and 2, the features must be normalized. For this, z-score normalization is used. The motivation why z-score normalization is used, was discussed in section §3.2.2. Once the features are normalized, the final preprocessing step is to split the datasets into two training datasets and two testing datasets. For each dataset 80% of the data goes to the training dataset while 20% goes to the testing dataset. The two training datasets (the features and their labels) are then used to create the `tf.data.Dataset`s representing the local training datasets of the clients. Listing 4 already showed how this is done. Once the client datasets are created, the federated learning system is initialized and 20 rounds of federated learning are executed (by calling the `next()` function on the `tff.templates.IterativeProcess` process 20 times, cfr. Intermezzo 2).

Once the federated learning system has established to train the weights of the global MLP model, the model must be tested. In the previous paragraph, it is mentioned that the two preprocessed datasets based on *internal week 1* and *internal week 2* each were split into a training and a testing set. Now, the two test sets are joined together to obtain one larger test set. Further, a keras MLP model (by using the auxiliary function as shown in Listing 5) is created and initialized with the weights of the federated learned model (as shown in Listing 9). If the model is initialized with the global weights trained by federated learning, the model is tested with the test set as described in the beginning of this paragraph. The precision scores of this test can be found in Table 4.9 (and the confusion matrix can be found in Appendix A).

Table 4.9: Precision scores of federated learning trained MLP model in Experiment 5 (all values in percentages)

Attack type	Results of the global MLP model
Normal	96.42
BruteForce	81.04
DoS	99.89
PingScan	92.26
PortScan	85.85

These results are quite good. If compared with the test in experiment 4, where the same

MLP model is trained without federated learning where all training data is "locally" available and thus where there is no loss of information due to federated averaging, these results are impressive. The only attack type that has a significant drop in score is the brute force attack where the score is 10% lower compared to experiment 4. This can be due to the training samples of the brute force attack being a relatively bad selection. However, all other attack types did approach the scores of experiment 4 very closely. Even more, the normal attack type exceeds the score of experiment 4 by about 5%. The reason for the "worse" score in experiment 4 could have been a relatively bad selection of training data for the normal flows.

The conclusion after this fifth experiment is that federated learning is indeed a viable solution for training network intrusion detection classification models. But, the question remains whether or not it allows for models to learn from each other when certain models do not have samples of a certain attack type available in their local training dataset while others do. Experiments 6 and 7 will be bring the answer to this question.

4.9 Experiment 6: Repeat base experiment with FL

This experiment corresponds to the notebook in the GitHub repository with version v4.2.

Experiment 5 shows that federated learning is indeed a usable approach for network intrusion detection since the precision scores in the federated learning setup are almost as good as the precision scores in the non-federated-learning setup. In this sixth experiment, the base experiment (cfr. experiment 2) is repeated but in a federated learning setup. This is done to see whether with federated learning, the global model eventually does learn to recognize flows of the attack types that are not available in both clients' local training datasets. If so, when the federated learning server sends these global weights back to the clients, the clients will be able to correctly predict flows of all attack types as opposed to without federated learning where each model could not correctly predict the flows of one attack type.

As always, the experiment starts with loading the cleaned up versions of the datasets *internal week 1* and *internal week 2*. Then, the features of both datasets are normalized using z-score normalization as discussed in §3.2.2. Following the normalization, since this is a repetition of the base experiment, all of the flows of the attack type "brute force" are removed from the *internal week 1* dataset and all of the flows of the attack type "ping scan" are removed from the *internal week 2* dataset. The next step in the preprocessing phase of this experiment is balancing out the dataset. Just like in the base experiment, balancing out these preprocessed datasets where all of the flows of certain attack types are removed, must be done with respect to the lowest represented attack type in each preprocessed dataset (i.e. excluding the removed attack type). After balancing out the datasets, they have the same distributions of flows with respect to the attack types as the preprocessed datasets in the base experiment (cfr. Table 4.3). Finally, the features and labels are separated from each other followed by splitting the datasets into two training datasets and testing datasets.

At this point, there is a preprocessed dataset based on the dataset *internal week 1* and a preprocessed dataset based on the dataset *internal week 2*. In this experiment, each dataset is seen as a local training dataset of a federated learning client. Therefore, each preprocessed dataset is converted to a `tf.data.Dataset`.

Then, the federated learning setup is created as discussed in Intermezzo 2 and the global model is trained with 20 federated learning iterations. When the global model is trained, it is tested with the testing datasets coming from *internal week 1* and *internal week 2*. Instead of performing two separate tests, the two test sets are joined together to obtain a bigger test set. The results of the test are shown in Table 4.10 as precision scores (the confusion matrix can be found in A).

Table 4.10: Precision scores of federated learning trained MLP model in Experiment 6 (all values in percentages)

Attack type	Results of the global MLP model
Normal	98.08
BruteForce	19.76
DoS	100.0
PingScan	67.00
PortScan	94.52

The results of this experiment show that federated learning does enable classification models to learn from each other because the precision scores of "brute force" and "ping scan" are certainly better than shown in experiment 2. But, a score of 20% for brute force is nothing to write home about. On the other hand, a precision score of 67% is quite good, coming from 0% in experiment 2.

Concerning the results of this experiment, it is also interesting to have a look at the confusion matrix which is presented in Table 4.11. The confusion matrix shows that when the global MLP model wrongfully classifies a brute force flow, most often the prediction is “port scan”: 70% of the brute force flows were predicted as port scan flows, 20% of the brute force flows were correctly predicted, 10% of the brute force flows were predicted as normal flows and none of the brute force flows are predicted as DoS or ping scan flows. It is clear that the label predicted by the MLP model, when the prediction is wrong, is not randomly selected from the other labels. This may indicate that the features of brute force flows are reminiscent to port scan flows and until the model has sufficiently been able to distinguish brute force flows from port scan flows, the model thinks that some brute force flows are most likely port scan flows. The same can be seen at the ping scan flows that are wrongfully predicted. Again, when the MLP model is wrong about ping scan flows, most often the predictions are “port scan” (95% of the wrong predictions of ping scan flows got the label “port scan” while the remaining 5% got the label “normal”).

Table 4.11: Confusion matrix of experiment 6

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	1328	2	7	1	16
Brute force	67	134	0	0	477
DoS	0	0	1330	0	0
Ping scan	11	0	0	473	222
Port scan	33	1	0	38	1241

The next experiment, i.e. experiment 7, is an additional experiment based on this experiment that keeps a bit more flows from the attack type that is removed to see how much it allows for improving the scores of the global model.

4.10 Experiment 7: Adaptation of base experiment with FL

This experiment corresponds to the notebooks in the GitHub repository with versions v4.3 to v4.8.

In experiment 6, the base experiment is repeated in a federated learning setup where the datasets are manipulated such that eventually not a single flow of the attack type “brute force" is available in *internal week 1* and analogously for "ping scan" in *internal week 2*. Although the results show that the global model does start to pick up these removed attack types, only a 20% precision score for the brute force attack is not quite enough to be useful in real life situations. At the same time, a precision score of 67% is not bad taking into account that the client with *internal week 2* as local training dataset would have correctly predicted 0% of the ping scan flows without federated learning. A second remark about the base experiment is that only the combination of removing brute force flows in one dataset and ping scan flows in the other dataset is considered. With these two remarks in mind, this seventh experiment is introduced where some additional tests are executed. First of all, when removing flows from a certain attack type, not "all the flows" are removed, rather 25% of the flows are kept in the balanced dataset and the other 75% of the flows are removed. This is done to see what the impact is on the scores of the model, having 25% flows available in the dataset as opposed to having no flows available. Further, the same test (i.e. with 25% available) is executed on all possible combinations of attack types (except for "normal", the attack type "normal" is never removed).

The preprocessing steps in each of the tests of this experiment are exactly the same as in experiment 6, except for how the datasets are balanced and how the flows of a certain attack type are removed. After loading the cleaned up versions of *internal week 1* and *internal week 2*, both datasets are (more or less) balanced out, considering both (1) avoiding as much imbalance as possible and (2) having as much flows available as possible for training and testing. Therefore, a baseline balanced dataset is created with a distribution of flows (with respect to the attack type) as shown in Table 4.12.

Table 4.12: Distribution of flows with respect to the attribute "attack type" for the datasets used in Experiment 7

Attack type	Number of flows from *internal week 1*	Number of flows from *internal week 2*
Normal	3,359	3,366
BruteForce	1,626	3,366
DoS	3,359	3,366
PingScan	3,359	2,731
PortScan	3,359	3,366

If, in a certain test, the combination of attack types "brute force" and "DoS" are chosen of which 75% of the flows are removed in respectively *internal week 1* and *internal week 2*, the resulting dataset would be as shown in Table 4.12 where the "1,626" flows of brute force in *internal week 1* is replaced with $\lfloor 0.25 \times 1,626 \rfloor = 406$ and where the "3,366" flows of DoS in *internal week 2* is replaced with $\lfloor 0.25 \times 3,366 \rfloor = 841$.

All other steps of the experiment, both with respect to the remaining preprocessing steps, splitting the datasets into training and testing datasets, and with respect to creating the federated learning setup in TensorFlow Federated, are the same as discussed in experiment 6. Therefore, these steps will not be repeated here.

Before the results of all the tests in this experiment are shown, Table 4.13 shows what notebook versions in the GitHub repository correspond to what combination of attack types from which flows are removed in respectively *internal week 1* and *internal week 2*. This is mainly done to improve the brevity of the column headers in Table 4.14 where the results of the tests are given. Additionally, Table 4.14 repeats the results of experiments 5 and 6, indicated by *v4.1* and *v4.2* respectively, to give an overview of the results of all federated learning experiments.

Table 4.13: Overview of what notebook versions correspond to what combinations of attack types

Notebook version number	Attack type of which 75% of flows is removed in *internal week 1*	Attack type of which 75% of flows is removed in *internal week 2*
v4.3	Brute force	DoS
v4.4	Brute force	Ping scan
v4.5	Brute force	Port scan
v4.6	DoS	Ping scan
v4.7	DoS	Port scan
v4.8	Ping scan	Port scan

In Table 4.14, the precision scores of the results of the tests are given. In order to easily see for what attack types 75% of the flows was removed in each test, the precision scores corresponding to these attack types are accentuated in bold and italic. In Appendix A, the confusion matrix of each of the tests is given for more detailed results.

Generally speaking, these results are very promising. In almost all of the tests, the prediction scores are almost as good as or even better than the scores of experiments 1[5]

[5] In experiment 1 two RFC models were used and tested on the *internal week 1* and *internal week 2* datasets

Table 4.14: Precision scores of all tests of federated learning trained MLP model in Experiment 7 (all values in percentages)

Attack type	v4.1	v4.2	v4.3	v4.4
Normal	96.42	98.08	97.05	96.18
BruteForce	81.04	***19.76***	***93.43***	***80.10***
DoS	99.89	100.0	***100.0***	99.93
PingScan	92.26	***67.00***	95.66	***95.76***
PortScan	85.85	94.52	90.46	90.77

Attack type	v4.5	v4.6	v4.7	v4.8
Normal	97.21	93.58	94.31	94.46
BruteForce	***68.36***	93.44	92.59	94.29
DoS	99.93	***99.88***	***99.88***	99.93
PingScan	94.54	***95.30***	94.81	***91.91***
PortScan	***92.38***	87.22	***86.75***	***89.62***

and 4[6] where no flows of any attack type are removed before training the classification models. However, there are two exceptions: the prediction scores in tests v4.4 and v4.5 for the brute force attack deviate significantly from the results in experiments 1 and 4 where no flows of any attack type are removed. This is also what was noted in experiment 6 and just like in experiment 6 it is worth having a look at the confusion matrices presented in Appendix A. Here too, when the MLP models in tests v4.4 and v4.5 are wrong about brute force flows, respectively 91.5% and 94.5% of the flows are given the label “port scan”.

In any case, the fact that the results in these experiments, where 75% of the flows of an attack type were removed in each of the clients’ local training dataset, are competitive with results where no flows of any attack type are removed, is very promising in favor of federated learning when thinking about the question being answered in this thesis: “Can classification models learn from each other using federated learning in the context of network intrusion detection?”

[6] In experiment 4, an MLP model was trained and tested with a balanced dataset containing data from both *internal week 1* and *internal week 2* to obtain as much training data as possible

5
Conclusions

The goal of this thesis was to find an answer to the question: “Can two network intrusion detection systems learn from each other using federated learning if the network flow classification models behind each of the IDSs have a local training dataset in which flows of one attack type are not available while the other local training dataset does have flows of that attack type available?” If asked to answer this question only with a “yes” or a “no”, the answer is “yes”. However, the in depth answer is more nuanced and will be discussed in this chapter. In the previous chapter, all experiments and their results leading to the answer to the aforementioned question are discussed.

The experiments started with using the classic machine learning model “random forest classifier”. The first experiment showed that such an RFC model is very suitable to classify the CIDDS-001 network flows with respect to the classification label “attack type”. The first experiment was followed by the so-called “base experiment”. In this experiment, two RFC models were trained with their own training dataset. The CIDDS-001 attribute “attack type” has five possible values: “normal”, “brute force”, “DoS”, “ping scan” and “port scan”. In the base experiment, all “brute force” network flows were removed from the training dataset of the first RFC model while all “ping scan” network flows were removed from the training dataset of the second RFC model. Without surprise, once the models are trained and tested with a test set that contains flows of all attack types, the models cannot correctly predict the flows of which the attack type was not available in their training dataset.

The base experiment serves as a motivation for why federated learning may be useful in the context of network intrusion detection. The base experiment shows that if an organization has two geographically separated networks where each network has its own IDS in which the classification model is trained with a local training dataset, it is possible that one IDS can detect attacks that the other cannot and vice versa. This is unfortunate since the organization would benefit from both IDSs being able to detect all attacks. Sharing the local training dataset could be the solution, but may be disallowed due to certain policies (privacy/security policies) since the training datasets may contain sensitive data, as the training datasets consist of captured network traffic. On top of that, sending the training datasets over the network and having to retrain the models can be too costly. However, the parameters of the classification models of each client can be exchanged and may be aggregated using federated averaging.

For the federated learning experiments 5, 6 and 7 (cfr. infra), the framework TensorFlow Federated is used. Unfortunately, TensorFlow Federated does not support RFC models to be used in federated learning. Therefore, a new classification model was introduced in experiment 3. The model that was used is a multi-layer perceptron neural network of which the configuration is based on the MLP presented in N. Oliveira et al.'s paper [36]. The same experiment that N. Oliveira et al. executed in their paper (which, to be clear, does not make use of federated learning) was executed in experiment 3. Due to great imbalance in the training set, the prediction scores of the attack types "brute force" and "ping scan" are very poor while the prediction scores of "normal", "DoS" and "port scan" are very good (cfr. results in Table 4.6).

Since the results of the third experiment that introduces the MLP model is subject to the negative effects of training a model with imbalanced data, the MLP model is used in the forth experiment in which it is trained with a balanced dataset. Now, the results are good for all attack types and they are similar to the results of the RFC models used in the first experiments.

Knowing all of this, it was time to start experimenting with federated learning. The first experiment that uses federated learning, i.e. experiment 5, tries to identify whether or not an MLP model that is trained using federated learning can reach competitive prediction scores when compared to the RFC and MLP models that are trained without federated learning. In experiment 5, two federated learning clients are considered by creating two local training datasets. These two local training datasets are balanced out so that no negative effect due to imbalanced training datasets is experienced. Then, the global MLP model is trained with federated learning and when tested, the conclusion is that the MLP model trained using federated learning is indeed competitive with the RFC and MLP models that were trained without federated learning. Thus, the loss of averaging multiple weights of the local MLP models is minimal.

Now that it is known that federated learning is capable of classifying the attack flows, the sixth experiment repeats the base experiment, but in a federated learning setup. This means that there are two clients and each of the clients has a local dataset in which flows from an attack type are removed. The first client has a local training dataset from which all “brute force” flows are removed while the second client has a local training dataset from which all “ping scan” flows are removed. Then, TensorFlow Federated’s federated learning implementation, with federated averaging as aggregation algorithm, is used to see whether the global MLP model does learn to recognize flows from the attack types “brute force” and “ping scan”, although only one of the clients contains flows from these attack types. The results show, as seen in Table 4.10, that the global MLP model indeed starts to learn how to classify flows of the attack types “brute force” and “ping scan”. However, there is a big difference in the score of “brute force” and the score of “ping scan”. The scores of the other attack types are comparable with the results in experiment 4 (MLP model trained with a balanced dataset in which no flows of any attack type were removed). Compared to the base experiment, the first client is now able to predict 20% of the “brute force” flows as opposed to 0% without federated learning, but also loses in capability of correctly predicting “ping scan” flows. The score drops from about 95% (average between the two tests executed in experiment 2) to 67%. The second client improves its score on “ping scan” from 0% to 67% but also loses in capability of predicting the attack type “brute force”: the score drops from 98% to 20%. The conclusion for the sixth experiment is that federated learning allows for their clients to start detecting attacks they previously could not detect. But, federated learning is no silver bullet. With the advantage of federated learning, also comes a disadvantage. Namely, that clients can also lose the ability to correctly predict some attacks due to the fact that the weights of the local MLP models of both clients are averaged.

Finally, there is a seventh experiment that also uses federated learning and provides additional tests. A first shortcoming of the sixth experiment is that only flows of the attack types “brute force” and “ping scan” are removed from the local datasets of respectively client 1 and client 2. What is the result if other combinations of attack types are removed? A second remark on the sixth experiment is that removing all flows of certain attack types is quite abrupt. Although such scenarios are not impossible, the usability of federated learning is not only limited to scenarios where certain datasets have no flows of a certain attack type while others do. Therefore, some additional tests are provided in the seventh experiment.

In the seventh experiment, instead of removing all flows of an attack type, a small portion of flows of the attack type is left in the local training dataset of a client. The experiment starts with creating more or less balanced datasets for each client that can be used as a baseline dataset, cfr. Table 4.12. Then, six tests are executed in which every time, for each of the baseline datasets, 75% of the available flows of one of the attack types is removed. In each test, a different combination of attack types is chosen. This way, each of the combinations of attack types is tested. It turns out that the results of these tests are impressive, as can be seen in Table 4.14. Almost all scores are well above 90%, also for the attack types from which 75% of the flows in the balanced dataset is removed. But, like in experiment 6, “brute force” deviates from this in two occasions. The first test where brute force has a lower score is the test where the combination is “brute force” and “ping scan”, this is somewhat to be expected since it is the same combination as used in experiment 6. The second test that has the same fate for the score of “brute force” is the test where the combination “brute force” and “port scan” is used. In both tests where the scores for “brute force” deviate from the rest, the biggest group of the samples is predicted to be “port scan”. This is also to be expected because it was already observed in experiment 6. But it has to be noted that the deviated score in this seventh experiment is much better than seen in experiment 6. Here, the scores are respectively 80% and 68% which is a big difference with the 20% as seen in experiment 6.

So, after experiments 6 and 7, the conclusion about federated learning is that it is certainly valuable to be used in the context of network intrusion detection. Federated learning enables multiple network intrusion detection classification models to learn from each other by sharing and aggregating the weights of the models, rather than sharing the local datasets.

Even though the conclusion about federated learning is made and thus the question of this thesis is answered, there is still some work that can be done in the future to give even more insight in the usefulness of federated learning in network intrusion detection. First, in the experiments 6 and 7, either all flows of an attack type were removed or 75%

was removed. The results of experiment 6 when all flows are removed are less useful in real life scenarios as opposed to training without federated learning while the results in experiment 7 show that if there is an imbalance in a local training dataset, but still 25% of the flows of an attack type is available (as opposed to the balanced scenario for that attack type), the results become generally very good. It may be useful to do research about how much imbalance federated learning can deal with. Another part of federate learning that requires further research is the number of clients. In the experiments of this thesis, only two clients are used. Performing experiments with more than two clients will increase insights in the usability of federated learning in network intrusion detection. For example: How does federated learning perform when there are three clients, where one attack type is removed from the three clients by 0% (i.e. nothing is removed), 75% and 100%. A lot of such variations can be thought of that give even more insights in the usability of federated learning in network intrusion detection systems.

Bibliography

[1] M. Ring, S. Wunderlich, D. Grüdl, D. Landes, and A. Hotho, “Flow-based benchmark data sets for intrusion detection,” in *Proceedings of the 16th European Conference on Cyber Warfare and Security (ECCWS).* ACPI, 2017, pp. 361–369.

[2] A. Verma and V. Ranga, “On evaluation of network intrusion detection systems: Statistical analysis of CIDDS-001 dataset using machine learning techniques,” *Pertanika Journal of Science and Technology*, vol. 26, no. 3, pp. 1307–1332, 2018.

[3] B. Li, Y. Wu, J. Song, R. Lu, T. Li, and L. Zhao, “DeepFed: Federated Deep Learning for Intrusion Detection in Industrial Cyber-Physical Systems,” *IEEE Transactions on Industrial Informatics*, vol. 17, no. 8, pp. 5615–5624, 2021.

[4] StatisticsTimes, “Projected GDP Ranking,” 2021. [Online]. Available: https://statisticstimes.com/economy/projected-world-gdp-ranking.php

[5] C. MacKechnie, “Information Technology & Its Role in the Modern Organization,” 2019. [Online]. Available: https://smallbusiness.chron.com/information-technology-its-role-modern-organization-1800.html

[6] B. Whitaker, “SolarWinds: How Russian spies hacked the Justice, State, Treasury, Energy and Commerce Departments,” 2021. [Online]. Available: https://www.cbsnews.com/news/solarwinds-hack-russia-cyberattack-60-minutes-2021-02-14/

[7] L. Constantin, “SolarWinds attack explained: And why it was so hard to detect,” 2020. [Online]. Available: https://www.csoonline.com/article/3601508/solarwinds-supply-chain-attack-explained-why-organizations-were-not-prepared.html

[8] M. Brennan, “”Hack everybody you can”: What to know about the massive Microsoft Exchange breach,” 2021. [Online]. Available: https://www.cbsnews.com/news/microsoft-exchange-server-hack-what-to-know/

[9] S. Morgan, “2021 Report: Cyberwarfare in the C-suite,” Cybercrime Magazine, Tech. Rep., 2021. [Online]. Available: https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/

[10] J. Lewis, “Economic impact of cybercrime : no slowing down,” *Centre for Strategic and International Studies*, no. February, pp. 1–28, 2018. [Online]. Available: https://www.csis.org/analysis/economic-impact-cybercrime

[11] L. Zhanna Malekos, Smith; Eugenia and J. A. Lewis, “The Hidden Costs of Cybercrime,” pp. 1–38, 2020. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-hidden-costs-of-cybercrime.pdf

[12] S. Roshan, Y. Miche, A. Akusok, and A. Lendasse, “Adaptive and online network intrusion detection system using clustering and Extreme Learning Machines,” *Journal of the Franklin Institute*, vol. 355, no. 4, pp. 1752–1779, 2018. [Online]. Available: https://doi.org/10.1016/j.jfranklin.2017.06.006

[13] AT&T, “Intrusion Detection Systems (IDS) explained,” 2021. [Online]. Available: https://cybersecurity.att.com/solutions/intrusion-detection-system/ids-explained

[14] E. De Poorter, “Security Goals,” in *Network and Computer Security*, 2020, ch. Chapter 2.

[15] S. A. Rahman, H. Tout, C. Talhi, and A. Mourad, “Internet of Things intrusion Detection: Centralized, On-Device, or Federated Learning?” *IEEE Network*, vol. 34, no. 6, pp. 310–317, 2020.

[16] A. Muallem, S. Shetty, L. Hong, and J. W. Pan, “TDDEHT: Threat Detection Using Distributed Ensembles of Hoeffding Trees on Streaming Cyber Datasets,” *Proceedings - IEEE Military Communications Conference MILCOM*, vol. 2019-Octob, pp. 219–224, 2019.

[17] T. H. Hai and N. T. Khiem, “Architecture for IDS Log Processing using Spark Streaming,” in *Proc. of the 2nd International Conference on Electrical, Communication and Computer Engineering (ICECCE)*, no. June, 2020, pp. 3–7.

[18] K. Alrawashdeh and C. Purdy, “Fast hardware assisted online learning using unsupervised deep learning structure for anomaly detection,” *2018 International Conference on Information and Computer Technologies, ICICT 2018*, pp. 128–134, 2018.

[19] G. Li, Y. Shen, P. Zhao, X. Lu, J. Liu, Y. Liu, and S. C. Hoi, "Detecting cyberattacks in industrial control systems using online learning algorithms," *Neurocomputing*, vol. 364, pp. 338–348, 2019. [Online]. Available: https://doi.org/10.1016/j.neucom.2019.07.031

[20] H. Huang, R. S. Khalid, W. Liu, and H. Yu, "Work-in-progress: A fast online sequential learning accelerator for IoT network intrusion detection," *2017 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2017*, no. 1, 2017.

[21] K. Alrawashdeh and C. Purdy, "Fast Activation Function Approach for Deep Learning Based Online Anomaly Intrusion Detection," *Proceedings - 4th IEEE International Conference on Big Data Security on Cloud, BigDataSecurity 2018, 4th IEEE International Conference on High Performance and Smart Computing, HPSC 2018 and 3rd IEEE International Conference on Intelligent Data and Securit*, pp. 5–13, 2018.

[22] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," *Proceedings - IEEE Symposium on Security and Privacy*, pp. 305–316, 2010.

[23] J. Johnson, "Anomaly Detection with Machine Learning: An Introduction," 2020. [Online]. Available: https://www.bmc.com/blogs/machine-learning-anomaly-detection/

[24] Fortinet, "DMZ." [Online]. Available: https://www.fortinet.com/resources/cyberglossary/what-is-dmz

[25] Paessler, "All-in-one NetFlow Analyzer PRTG," 2021. [Online]. Available: https://www.paessler.com/netflow_monitoring

[26] M. Ring, S. Wunderlich, D. Gruedl, D. Landes, and A. Hotho, "Technical Report CIDDS-001 data set," in *Proceedings of the 16th European Conference on Cyber Warfare and Security (ECCWS), to appear.* ACPI, 2017, vol. 001, pp. 1–13. [Online]. Available: https://www.hs-coburg.de/fileadmin/hscoburg/Forschung/WISENT_cidds_Technical_Report.pdf

[27] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology*, vol. 10, no. 2, pp. 1–19, 2019.

[28] H. Brendan McMahan, E. Moore, D. Ramage, S. Hampson, and B. Agüera y Arcas, “Communication-efficient learning of deep networks from decentralized data,” *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017*, vol. 54, 2017.

[29] L. Zhao, Q. Wang, Q. Zou, Y. Zhang, and Y. Chen, “Privacy-Preserving Collaborative Deep Learning with Unreliable Participants,” *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1486–1500, 2020.

[30] Python, “Python Release Python 3.8.10.” [Online]. Available: https://www.python.org/downloads/release/python-3810/

[31] P. D. Team, “Pandas version 1.2.4 documentation,” 2021. [Online]. Available: https://pandas.pydata.org/pandas-docs/version/1.2.4/

[32] Scikit-learn, “Release Highlights for scikit-learn 0.24 – scikit-learn 0.24.2.” [Online]. Available: https://scikit-learn.org/stable/auto_examples/release_highlights/plot_release_highlights_0_24_0.html

[33] TensorFlow, “TensorFlow.” [Online]. Available: https://www.tensorflow.org/

[34] A. Verma and V. Ranga, “Statistical analysis of CIDDS-001 dataset for Network Intrusion Detection Systems using Distance-based Machine Learning,” *Procedia Computer Science*, vol. 125, no. December, pp. 709–716, 2018. [Online]. Available: https://doi.org/10.1016/j.procs.2017.12.091

[35] B. A. Tama and K.-H. Rhee, “Attack classification analysis of IoT network via deep learning approach,” *Res. Briefs Inf. Commun. Technol. Evol.*, vol. 3, no. 15, pp. 1–9, 2017.

[36] N. Oliveira, I. Praça, E. Maia, and O. Sousa, “Intelligent cyber attack detection and classification for network-based intrusion detection systems,” *Applied Sciences (Switzerland)*, vol. 11, no. 4, pp. 1–21, 2021.

[37] J. Brownlee, “How to Scale Data With Outliers for Machine Learning,” 2020. [Online]. Available: https://machinelearningmastery.com/robust-scaler-transforms-for-machine-learning/

[38] T. Yiu, “Understanding Random Forest,” 2019. [Online]. Available: https://towardsdatascience.com/understanding-random-forest-58381e0602d2

[39] J. Leonel, “Perceptrons,” 2018. [Online]. Available: https://medium.com/\spacefactor\@m{}jorgesleonel/perceptrons-a22fb29facc4

[40] MLK, “Dummies guide to Cost Functions in Machine Learning,” 2019. [Online]. Available: https://machinelearningknowledge.ai/cost-functions-in-machine-learning/#Why_Cross_Entropy_and_Not_MAEMSE_in_Classification

[41] M. Nielsen, “The backpropagation algorithm,” in *Neural Networks and Deep Learning*, 2019, ch. How the ba. [Online]. Available: http://neuralnetworksanddeeplearning.com/

[42] J. Leonel, “Backpropagation,” 2018. [Online]. Available: https://medium.com/@jorgesleonel/backpropagation-cc81e9c772fd

[43] A. V. Srinivasan, “Stochastic Gradient Descent — Clearly Explained !!” 2019. [Online]. Available: https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31

[44] J. Brownlee, “A Gentle Introduction to Dropout for Regularizing Deep Neural Networks,” 2019. [Online]. Available: https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/

[45] Wikipedia, “Softmax function,” 2021. [Online]. Available: https://en.wikipedia.org/wiki/Softmax_function

[46] A. ARROW, “Feather File Format,” 2021. [Online]. Available: https://arrow.apache.org/docs/python/feather.html

[47] Scikit-learn, “RandomForestClassifier,” 2020. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[48] TensorFlow, “TensorFlow Federated: Machine Learning on Decentralized Data.” [Online]. Available: https://www.tensorflow.org/federated

[49] E. I. Polat, “Federated Learning: A Simple Implementation of FedAvg (Federated Averaging) with PyTorch,” 2020. [Online]. Available: https://towardsdatascience.com/federated-learning-a-simple-implementation-of-fedavg-federated-averaging-with-pytorch-90187c9c9577

[50] TensorFlowFederated, “Federated Learning for Text Generation,” 2021. [Online]. Available: https://www.tensorflow.org/federated/tutorials/federated_learning_for_text_generation

[51] ——, “Federated Learning for Image Classification,” 2021. [Online]. Available: https://www.tensorflow.org/federated/tutorials/federated_learning_for_image_classification

[52] ——, “tff.learning.build_federated_averaging_process,” 2021. [Online]. Available: https://www.tensorflow.org/federated/api_docs/python/tff/learning/build_federated_averaging_process

[53] ——, “tff.templates.IterativeProcess,” 2021. [Online]. Available: https://www.tensorflow.org/federated/api_docs/python/tff/templates/IterativeProcess

[54] ——, “tff.learning.framework.ServerState,” 2021. [Online]. Available: https://www.tensorflow.org/federated/api_docs/python/tff/learning/framework/ServerState

[55] Numpy, “Numpy,” 2021. [Online]. Available: https://numpy.org/

A

Confusion matrices of the experiments

Before the confusion matrices are presented, a quick note on how they should be interpreted. Each row represents the true label of the samples and the columns represent what label the classification model has given to the samples.

For example, in Table A.1, the value 326 in the first row and first column of the confusion matrix tells that the model labeled 326 samples as “Normal” while their actual label was “Normal”. On the other hand, the value 7 in the first row and second column of the confusion matrix tells that the model labeled 7 samples as “Brute force” while their actual label was “Normal”.

A.1 Experiment 1

A.1.1 Testing rfc_week1 on the test set from internal week 1

Table A.1: Confusion matrix of experiment 1 where rfc_week1 is tested on the test set from internal week 1

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	331	2	0	0	0
Brute force	19	309	0	0	0
DoS	0	0	331	0	0
Ping scan	20	1	0	285	1
Port scan	7	0	0	3	317

A.1.2 Testing rfc_week2 on the test set from internal week 2

Table A.2: Confusion matrix of experiment 1 where rfc_week2 is tested on test set from internal week 2

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	555	7	0	0	0
Brute force	0	495	0	0	0
DoS	0	0	575	0	0
Ping scan	36	15	0	520	0
Port scan	11	12	0	23	482

A.1.3 Testing rfc_week1 on the full dataset internal week 2

Table A.3: Confusion matrix of experiment 1 where rfc_week1 is tested on the full dataset internal week 2

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	8491794	11596	3350	619	7970
Brute force	259	2896	0	0	211
DoS	652	11	1706236	0	1
Ping scan	242	1	0	2481	7
Port scan	3686	30	0	2380	76311

A.1.4 Testing rfc_week2 on the full dataset internal week 1

Table A.4: Confusion matrix of experiment 1 where rfc_week2 is tested on the full dataset internal week 1

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	6936118	61454	3794	659	8872
Brute force	58	1567	0	0	1
DoS	232	9	1251880	0	6
Ping scan	88	25	0	3244	2
Port scan	2032	1229	0	4815	175435

A.2 Experiment 2

A.2.1 Testing rfc_week1 on the test set from internal week 1

Table A.5: Confusion matrix of experiment 2 where rfc_week1 is tested on the test set of week 1 where all of the brute force flows are removed

	Normal	DoS	Ping scan	Port scan
Normal	2717	0	0	0
DoS	1	2704	0	0
Ping scan	77	0	2576	2
Port scan	49	0	65	2557

A.2.2 Testing rfc_week2 on the test set from internal week 2

Table A.6: Confusion matrix of experiment 2 where rfc_week2 is tested on the test set of week 2 where all of the ping scan flows are removed

	Normal	Brute force	DoS	Ping scan
Normal	2681	36	1	3
Brute force	3	2693	0	0
DoS	0	0	2667	0
Port scan	44	68	0	2575

A.2.3 Testing rfc_week1 on the full dataset internal week 2

Table A.7: Confusion matrix of experiment 2 where rfc_week1 is tested on the full dataset internal week 2

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	8506966	0	3015	771	4577
Brute force	3155	0	0	0	211
DoS	1349	0	1705550	0	1
Ping scan	243	0	0	2483	5
Port scan	3696	0	0	2377	76334

A.2.4 Testing rfc_week2 on the full dataset internal week 1

Table A.8: Confusion matrix of experiment 2 where rfc_week2 is tested on the full dataset internal week 1

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	6933654	61464	6448	0	9331
Brute force	61	1565	0	0	0
DoS	344	9	1251771	0	3
Ping scan	88	25	0	0	3246
Port scan	2016	1250	0	0	180245

A.3 Experiment 3

Table A.9: Confusion matrix of experiment 3

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	8477675	0	30307	0	7347
Brute force	1858	208	0	0	1300
DoS	1292	0	1705601	0	7
Ping scan	244	0	0	5	2482
Port scan	8025	46	0	0	74336

A.4 Experiment 4

Table A.10: Confusion matrix of experiment 4

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	1093	86	16	11	4
Brute force	13	989	0	5	11
DoS	0	0	1158	0	0
Ping scan	28	32	0	1170	2
Port scan	12	120	0	40	1027

A.5 Experiment 5

Table A.11: Confusion matrix of experiment 5

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	863	23	5	2	2
Brute force	105	667	0	6	45
DoS	1	0	905	0	0
Ping scan	38	27	0	810	3
Port scan	11	75	0	35	734

A.6 Experiment 6

Table A.12: Confusion matrix of experiment 6

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	1328	2	7	1	16
Brute force	67	134	0	0	477
DoS	0	0	1330	0	0
Ping scan	11	0	0	473	222
Port scan	33	1	0	38	1241

A.7 Experiment 7

A.7.1 Test *v4.3*

Table A.13: Confusion matrix of experiment 7, test *4.3*

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	1285	34	0	3	2
Brute force	9	697	0	2	38
DoS	0	0	840	0	0
Ping scan	32	16	0	1147	4
Port scan	22	57	0	54	1261

A.7.2 Test *v4.4*

Table A.14: Confusion matrix of experiment 7, test *4.4*

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	1234	40	5	1	3
Brute force	9	616	0	4	140
DoS	1	0	1359	0	0
Ping scan	22	10	0	746	1
Port scan	28	47	0	55	1278

A.7.3 Test *v4.5*

Table A.15: Confusion matrix of experiment 7, test *4.5*

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	1287	32	0	4	1
Brute force	10	510	0	3	223
DoS	1	0	1353	0	0
Ping scan	41	24	0	1159	2
Port scan	9	28	0	28	788

A.7.4 Test *v4.6*

Table A.16: Confusion matrix of experiment 7, test *4.6*

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	1195	69	10	0	3
Brute force	21	955	0	0	46
DoS	1	0	859	0	0
Ping scan	28	9	0	750	0
Port scan	33	89	0	56	1215

A.7.5 Test *v4.7*

Table A.17: Confusion matrix of experiment 7, test *4.7*

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	1243	57	6	10	2
Brute force	29	925	0	8	37
DoS	1	0	853	0	0
Ping scan	37	25	0	1170	2
Port scan	15	64	0	32	727

A.7.6 Test *v4.8*

Table A.18: Confusion matrix of experiment 7, test *4.8*

	Normal	Brute force	DoS	Ping scan	Port scan
Normal	1245	65	6	0	2
Brute force	16	942	0	4	37
DoS	1	0	1333	0	0
Ping scan	34	25	0	693	2
Port scan	18	40	0	29	751

B

Distributions of datasets used throughout the experiments

This Appendix gives an overview of the histograms of the distributions with respect to the classification label “AttackType” of all datasets that are used throughout the seven experiments in this thesis.

B.1 Experiment 1

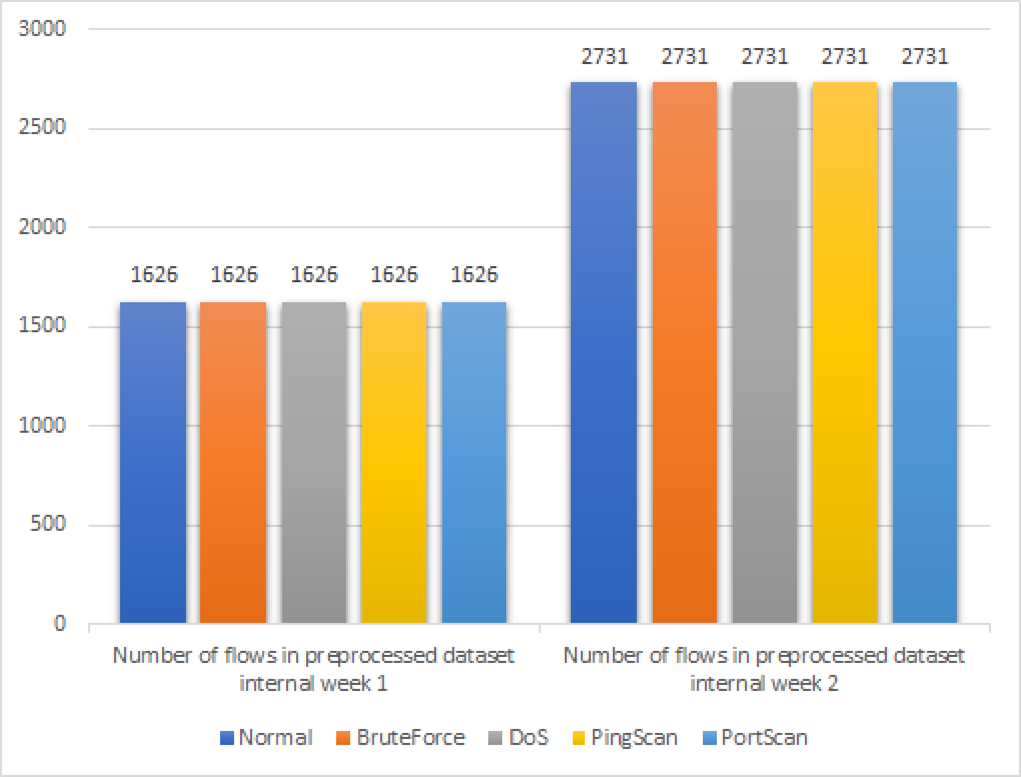


Figure B.1: Histogram of the distribution of flows with respect to the attribute "attack type" for the datasets used in Experiment 1

B.2 Experiment 2

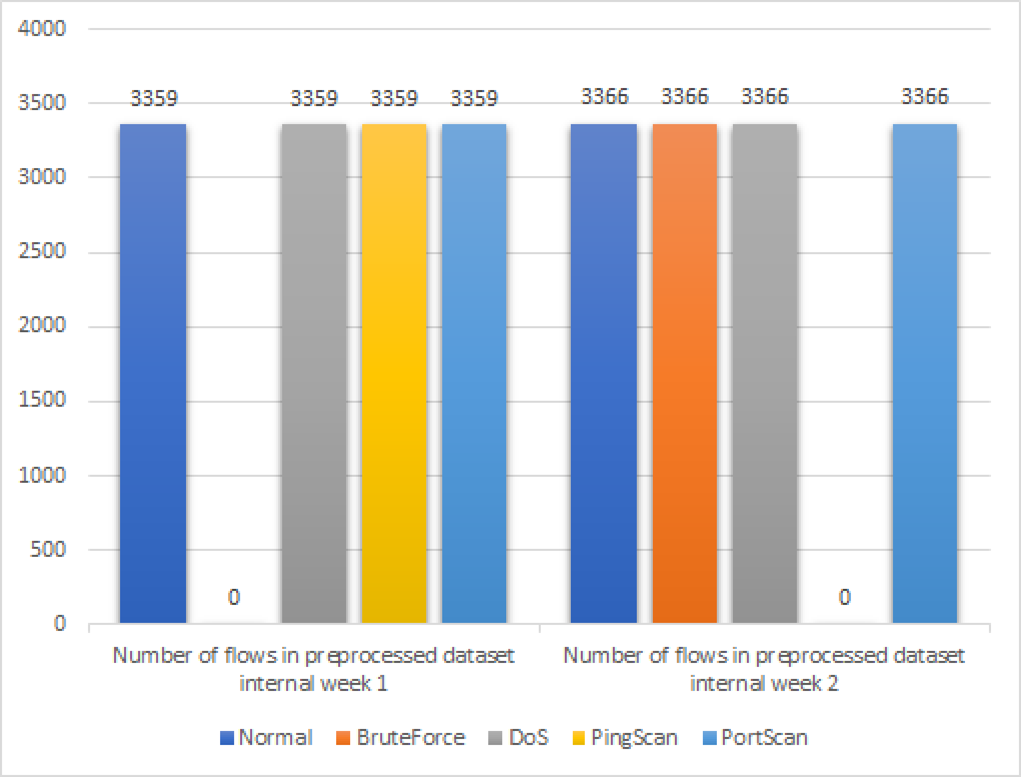


Figure B.2: Histogram of the distribution of flows with respect to the attribute “attack type” for the datasets used in Experiment 2

B.3 Experiment 3

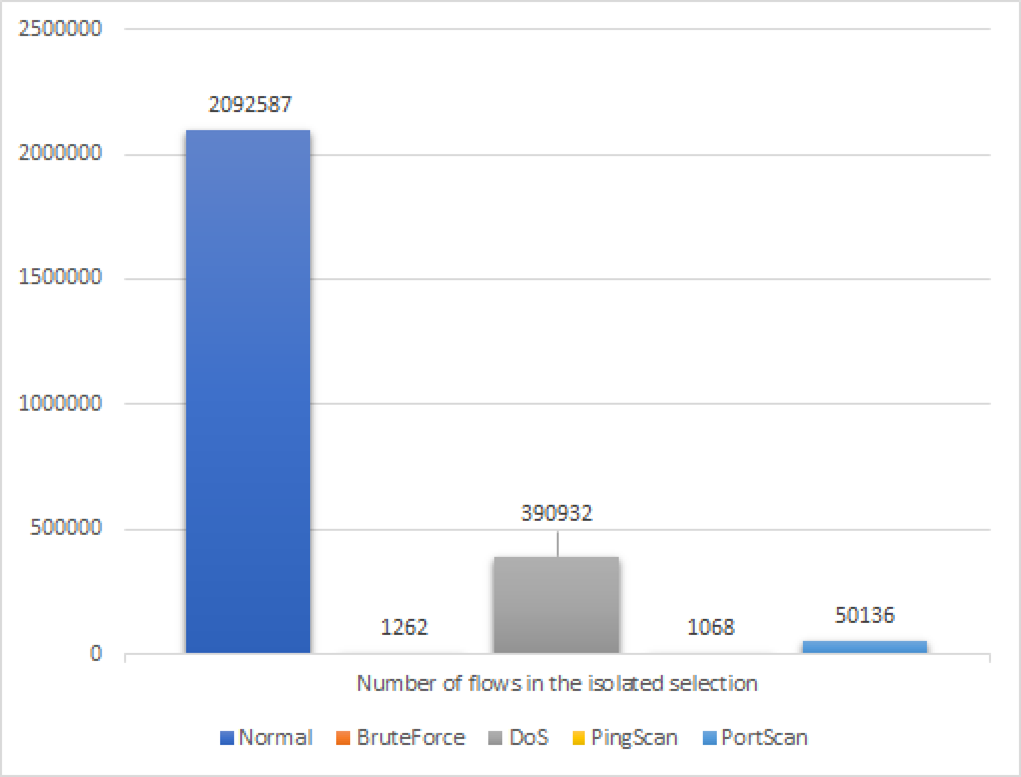


Figure B.3: Histogram of the distribution of flows with respect to the attribute “attack type” for the datasets used in Experiment 3

B.4 Experiment 4

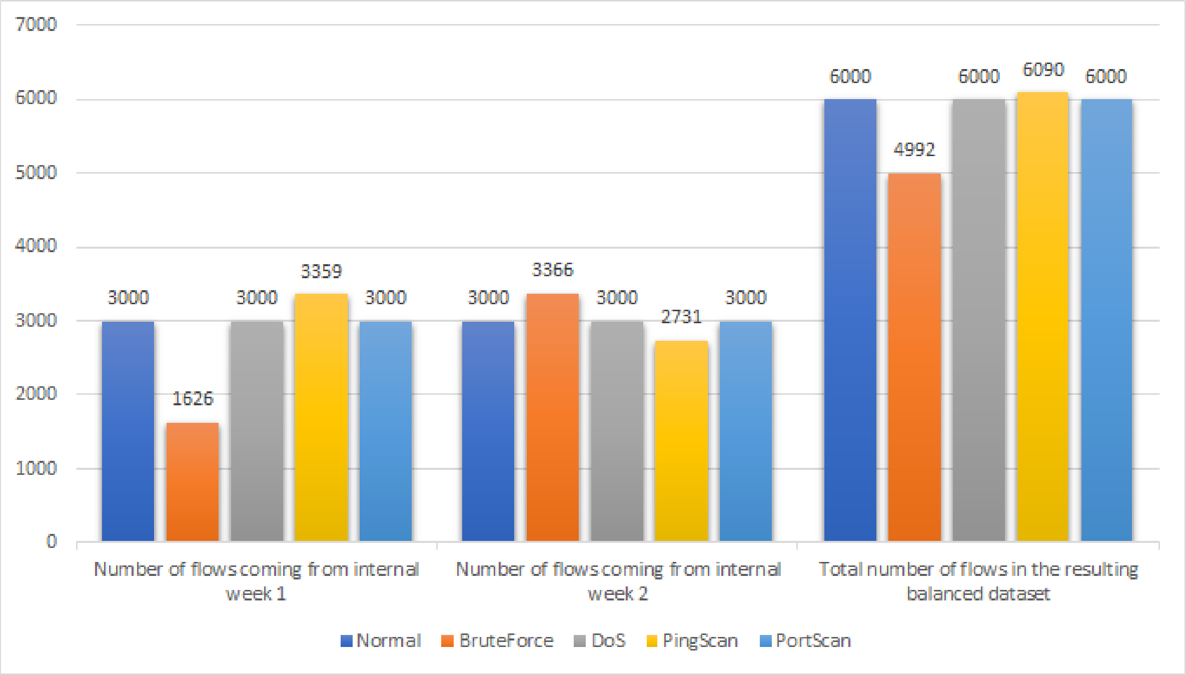


Figure B.4: Histogram of the distribution of flows with respect to the attribute "attack type" for the datasets used in Experiment 4

B.5 Experiment 5

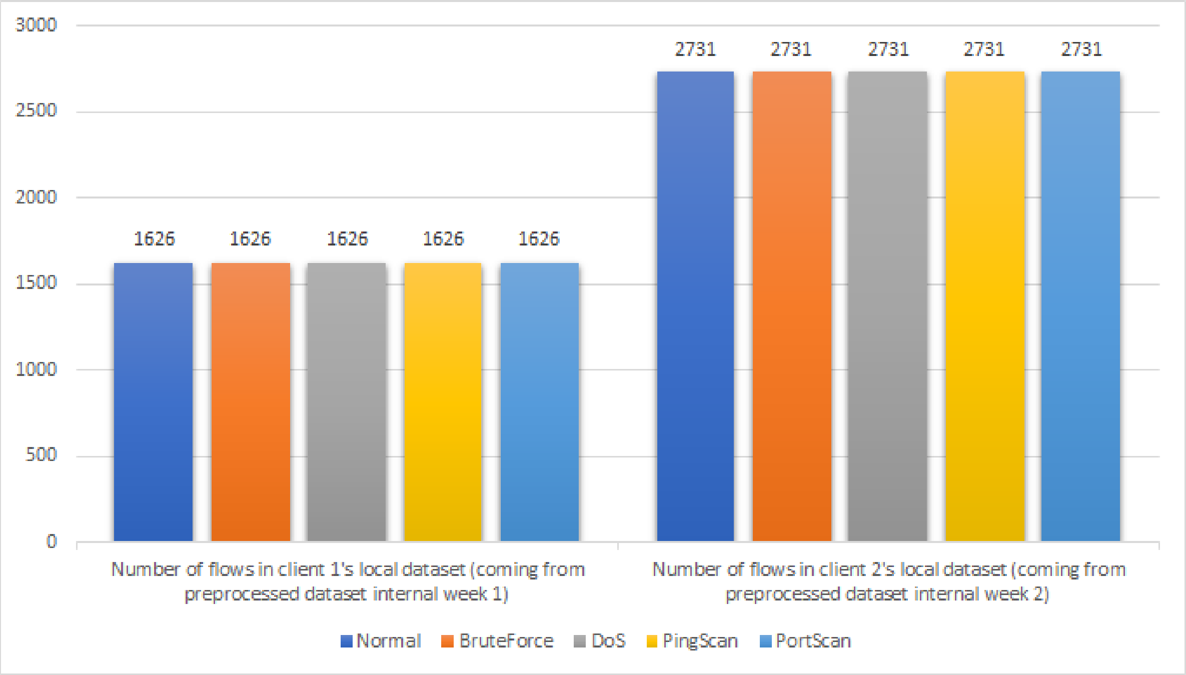


Figure B.5: Histogram of the distribution of flows with respect to the attribute “attack type” for the datasets used in Experiment 5

B.6 Experiment 6

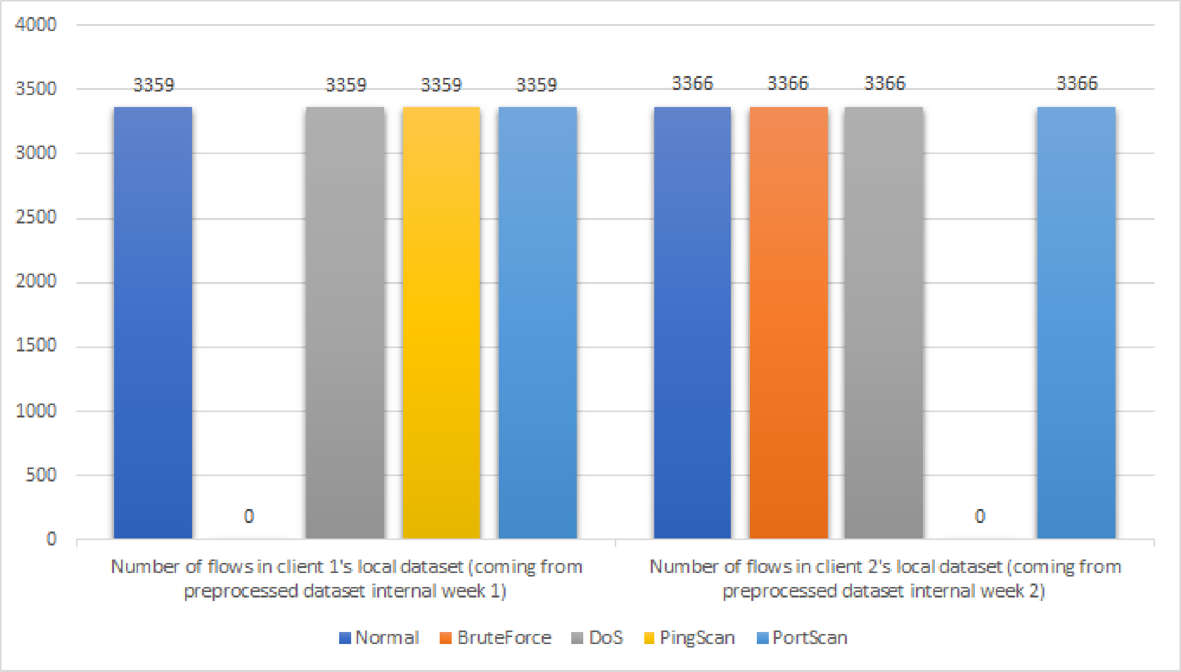


Figure B.6: Histogram of the distribution of flows with respect to the attribute "attack type" for the datasets used in Experiment 6

B.7 Experiment 7

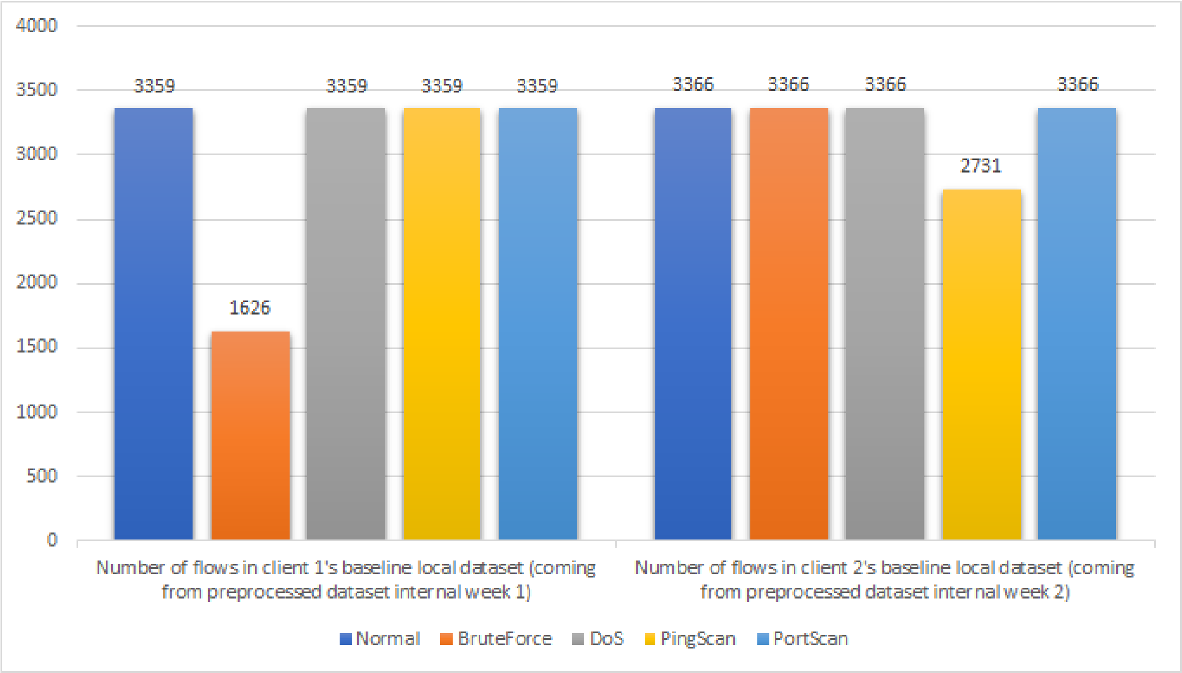


Figure B.7: Histogram of the distribution of flows with respect to the attribute “attack type” for the datasets used in Experiment 7