Deep learning for Internet of Things (IoT) network security

Mustafizur Rahman Shahid

▶ To cite this version:

Mustafizur Rahman Shahid. Deep learning for Internet of Things (IoT) network security. Artificial Intelligence [cs.AI]. Institut Polytechnique de Paris, 2021. English. NNT : 2021IPPAS003 . tel-03193266

HAL Id: tel-03193266
https://theses.hal.science/tel-03193266

Submitted on 8 Apr 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

NNT : 2021IPPAS003

Thèse de doctorat

Deep Learning for Internet of Things (IoT) Network Security

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom SudParis

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 22 Mars 2021, par

MUSTAFIZUR RAHMAN SHAHID

Composition du Jury :

Thomas Clausen Professeur, École Polytechnique - IP Paris	Président
Sébastien Tixeuil Professeur, Sorbonne Université	Rapporteur
Eric Totel Professeur, IMT Atlantique	Rapporteur
Youki Kadobayashi Professeur, Nara Institute of Science and Technology	Examinateur
Cristel Pelsser Professeure, Université de Strasbourg	Examinatrice
Urko Zurutuza Maître de conférences, Mondragon University	Examinateur
Hervé Debar Professeur, Télécom SudParis - IP Paris	Directeur de thèse
Gregory Blanc Maître de conférences, Télécom SudParis - IP Paris	Co-encadrant de thèse
Zonghua Zhang Professeur, IMT Lille Douai	Invité

Deep Learning for Internet of Things (IoT) Network Security

Mustafizur Rahman SHAHID

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Télécom SudParis
Institut Polytechnique de Paris
France
March, 2021

Abstract

The growing Internet of Things (IoT) introduces new security challenges for network activity monitoring. Most IoT devices are vulnerable because of a lack of security awareness from device manufacturers and end users. As a consequence, they have become prime targets for malware developers who want to turn them into bots and use them to perform large scale attacks.

Contrary to general-purpose devices, an IoT device is designed to perform very specific tasks. Hence, its networking behavior is very stable and predictable making it well suited for data analysis techniques. Therefore, the first part of this thesis focuses on leveraging recent advances in the field of deep learning to develop network monitoring tools for the IoT. Two types of network monitoring tools are explored: IoT device type recognition systems and IoT Network Intrusion Detection Systems (NIDS). For IoT device type recognition, supervised machine learning algorithms are trained to perform network traffic classification and determine what IoT device the traffic belongs to. The IoT NIDS consists of a set of autoencoders, each trained for a different IoT device type. The autoencoders learn the legitimate networking behavior profile and detect any deviation from it. Experiments using network traffic data produced by a smart home show that the proposed models achieve high performance.

Despite yielding promising results, training and testing machine learning based network monitoring systems requires tremendous amount of IoT network traffic data. But, very few IoT network traffic datasets are publicly available. Physically operating thousands of real IoT devices can be very costly and can rise privacy concerns. In the second part of this thesis, we propose to leverage Generative Adversarial Networks (GAN) to generate synthetic bidirectional flows that look like they were produced by a real IoT device. Generated bidirectional flows consist of a sequence of individual packet sizes along with a duration value. Hence, in addition to generating packet-level features which are the sizes of individual packets, our developed

generator implicitly learns to comply with flow-level characteristics, such as the ordering of the packets, the total number of packets and bytes in a bidirectional flow or the total duration of the flow. Experimental results using data produced by a smart speaker show that our method allows us to generate high quality and realistic looking synthetic bidirectional flows.

Résumé

L'internet des objets (IoT) introduit de nouveaux défis de sécurité pour la surveillance des réseaux. La plupart des appareils IoT sont vulnérables en raison d'un manque de sensibilisation à la sécurité des fabricants d'appareils et des consommateurs. En conséquence, ces appareils sont devenus des cibles privilégiées pour les développeurs de malware qui veulent les transformer en bots (appareils infectés ou "zombies" contrôlés par un acteur malveillant) pour ensuite pouvoir les utiliser pour mener des attaques à grandes échelles.

Contrairement à un ordinateur de bureau, un objet IoT est conçu pour accomplir des tâches très spécifiques. Par conséquent, son comportement réseau est très stable et prévisible, ce qui le rend bien adapté aux techniques d'analyse de données. Ainsi, la première partie de cette thèse tire profit des algorithmes de *deep learning* pour développer des outils de surveillance des réseaux IoT. Deux types d'outils de surveillance réseau sont explorés: Les systèmes de reconnaissance de type d'objets IoT et les systèmes de détection d'intrusion réseau IoT. Des données réseau produites par une maison connectée sont utilisées pour évaluer la performance des solutions développées. Pour développer le système de reconnaissance de types d'objets IoT, nous avons d'abord défini un ensemble de caractéristiques (*features*) appropriées pour décrire les flux réseaux bidirectionnels. Il s'agit de la taille des N premiers paquets envoyés et reçus ainsi que le temps interarrivé entre ces paquets. Nous avons procédé à la visualisation des données à l'aide de l'algorithme t-SNE (*t-Distributed Stochastic Neighbor Embedding*) pour mettre en évidence la capacité des *features* sélectionnées à bien distinguer les flux bidirectionnels produits par les différents type d'objets IoT. Nous avons ensuite entraîné et testé différents algorithmes d'apprentissage supervisés pour classer les flux bidirectionnels en fonction du type d'objet IoT auquel ils appartiennent. L'algorithme *Random Forest* a atteint une exactitude globale de 99,9%. Pour le système de détection d'intrusion réseau IoT, les *features* utilisés pour décrire les flux réseaux bidirectionnels sont

des statistiques (moyenne, écart type, maximum, minimum, etc) sur la taille des N premiers paquets envoyés et reçus, ainsi que des statistiques sur le temps interarrivé entre ces paquets. Le système de détection d'intrusion consiste en un ensemble d'*autoencoders*, chacun étant entraîné pour un type d'objet IoT différent. Les *autoencoders* sont des réseaux de neurones non supervisés qui apprennent le profil du comportement réseau légitime et détectent tout écart par rapport à celui-ci. Les résultats expérimentaux montrent que notre méthode permet d'obtenir un taux de vrais positifs élevé pour un faible taux de faux positifs.

Bien que permettant d'obtenir des résultats prometteurs, l'entraînement et l'évaluation des modèles de *deep learning* nécessitent une quantité énorme de données réseau IoT. Or, très peu de jeux de données de trafic réseau IoT sont accessibles au public. Le déploiement physique de milliers d'objets IoT réels peut être très coûteux et peut poser problème quant au respect de la vie privée. Ainsi, dans la deuxième partie de cette thèse, nous proposons d'exploiter des *GAN (Generative Adversarial Networks)* pour générer des flux bidirectionnels qui ressemblent à ceux produits par un véritable objet IoT. Un flux bidirectionnel est représenté par la séquence des tailles des paquets ainsi que de la durée (*duration*) du flux. Par conséquent, en plus de générer des caractéristiques au niveau des paquets, tel que la taille de chaque paquet, notre générateur apprend implicitement à se conformer aux caractéristiques au niveau du flux, comme le nombre total de paquets et d'octets dans un flux ou sa durée totale. Les séquences de tailles de paquets ont été modélisées comme étant des séquences de données catégorielles. Ainsi, pour surmonter le problème de l'utilisation des *GAN* pour la génération de séquences de données catégorielles, nous avons décidé de combiner *GAN* (plus précisément un *Wasserstein GAN* ou *WGAN*) et *autoencoder*. Tout d'abord, l'*autoencoder* est entraîné pour apprendre à convertir des séquences de données catégorielles, à savoir la séquence des tailles des paquets, en un vecteur latent dans un espace continu. Ensuite, un *WGAN* est entraîné dans l'espace latent continue pour apprendre à générer des vecteurs latents qui pourront être décodés en séquences réalistes, grâce à la partie décodeur de l'*autoencodeur*. Pour chaque séquence de tailles de paquets générée, nous avons également déterminé sa durée totale. La durée étant une variable contenant du bruit aléatoire (en raison de la congestion du réseau par exemple), sa valeur a été déterminée à l'aide d'un *Mixture Density Network (MDN)*, un type de réseau de neurones capable de produire des distributions de probabilité permettant de modéliser l'incertitude. Des résultats expérimentaux utilisant des données produites par un haut-parleur intelligent montrent que notre méthode permet de générer des flux bidirectionnels synthétiques réalistes et de haute qualité.

Acknowledgements

I would like to thank my PhD supervisor Prof. Hervé Debar, as well as co-supervisors Prof. Gregory Blanc and Prof. Zonghua Zhang. They guided and encouraged me throughout this long journey. Without their help, the goal of this project could not have been realized.

I would also like to express my deepest gratitude to *Institut Mines-Télécom's Futur & Ruptures* program, the *Fondation Mines-Télécom* and the *Carnot Télécom & Société numérique* for providing the necessary funding for the successful completion of this project.

I would also like to thank all the people who supported me throughout this project and made life easier. They are so many that I will certainly miss most of them if I had to mention them name by name. They are colleagues, lab members, technical personnel, support staff, family or friends. Without their support this project could not have reached its goal.

Contents

Abstract i

Résumé iii

Acknowledgements v

List of Figures ix

List of Tables xii

1 Introduction 1

1.1 IoT Ecosystem Characteristics 1
1.2 Vulnerabilities in the IoT 5
1.3 IoT Botnets 8
1.4 IoT Security Solutions 11
1.5 Objectives of the Thesis 15
1.5.1 Context 15
1.5.2 Limitations of Existing Works 16
1.5.3 Contributions 17

2 State of the Art 19

2.1 Deep Learning 19
2.1.1 A Sub-field of Machine Learning 20
2.1.2 Neural Network Training 22
2.1.3 Metrics for Model Assessment 25
2.1.4 Neural Network Architectures 28
2.1.5 Deep Learning Applications in Cybersecurity 31
2.2 IoT Network Traffic Classification 33

2.2.1 IoT Device Type Identification . . . 33
2.2.2 IoT Device State Determination . . . 37
2.2.3 Summary . . . 41
2.3 IoT Network Intrusion Detection . . . 43
2.3.1 Supervised NIDS . . . 43
2.3.2 Unsupervised NIDS/ Anomaly Detection . . . 46
2.3.3 Summary . . . 48
2.4 IoT Network Traffic Generation . . . 50
2.4.1 Flow-level Network Traffic Generation . . . 50
2.4.2 Network Packets Generation . . . 54
2.4.3 Summary . . . 56

3 IoT Network Traffic Monitoring 58
3.1 IoT Device Recognition through Network Traffic Classification . . . 59
3.1.1 Overview . . . 59
3.1.2 Features Description . . . 60
3.1.3 Smart Home Dataset Description . . . 61
3.1.4 Experimental Results - Network Traffic Visualization . . . 63
3.1.5 Experimental Results - Classification . . . 64
3.1.6 Discussion . . . 67
3.1.7 Summary . . . 68
3.2 IoT Network Anomaly Detection . . . 68
3.2.1 Overview . . . 68
3.2.2 Features Description . . . 69
3.2.3 Sparse Autoencoder for Anomaly Detection . . . 70
3.2.4 Individual Autoencoders vs Set of Autoencoders . . . 73
3.2.5 Dataset Description . . . 74
3.2.6 Evaluation Methodology . . . 75
3.2.7 Experimental Results: Performance of Individual Autoencoders 76
3.2.8 Experimental Results: Performance of the Set of Autoencoders 77
3.2.9 Discussion . . . 80
3.2.10 Summary . . . 82
3.3 General Conclusion . . . 82

4 IoT Network Traffic Generation 84
4.1 Generating Sequences of Packet Sizes . . . 86
4.1.1 Modeling Sequences of Packet Sizes . . . 86
4.1.2 Generative Models for Sequences of Categorical Data Generation 89
4.1.3 Evaluation Methodology . . . 93
4.1.4 Smart Speaker Dataset . . . 93
4.1.5 Experimental Results - Simplified Packet Ordering . . . 94
4.1.6 Experimental Results - Realistic Packet Ordering . . . 100
4.1.7 Discussion . . . 103

4.1.8 Summary 104
4.2 Determining the Duration of a Generated Bidirectional Flow 104
4.2.1 Duration as a Random Variable 105
4.2.2 Mixture Density Networks 105
4.2.3 Evaluation Methodology 107
4.2.4 Duration Dataset 108
4.2.5 Experimental Results 108
4.2.6 Summary 110
4.3 General Conclusion 111

5 Conclusion 112

List of Publications 116

Bibliography 117

List of Figures

1.1 IoT ecosystem characteristics . . . 2
1.2 Attack example resulting from interdependence between devices by W. Zhou et al. [Zho+18b] . . . 3
1.3 Overview of Mirai botnet communication and basic components . . . 9
1.4 IoT network architecture . . . 14

2.1 Detailed representation of how the output of a neuron is computed and its corresponding simplified representation . . . 22
2.2 Example of an artificial neural network architecture . . . 22
2.3 K-fold cross-validation . . . 23
2.4 Neural Network Training . . . 24
2.5 I. Goodfellow et al. [GBC16]: Typical relationship between a model's capacity and the training and generalization errors . . . 25
2.6 Confusion matrix . . . 26
2.7 ROC curve example [Sci20] . . . 28
2.8 Recurrent Neural Network . . . 29
2.9 Autoencoder . . . 30
2.10 Variational Autoencoder . . . 30
2.11 Generative Adversarial Networks . . . 31
2.12 Y. Meidan et al. [Mei+17a]: Overview of proposed method for IoT white listing . . . 34
2.13 A. Sivanathan et al. [Siv+18]: Architecture of the multi-stage classifier 35
2.14 S. Marchal et al. [Mar+19]: Overview of device type identification . 36
2.15 A. Acar et al. [Aca+18]: Overview of the multi-stage privacy attack 38
2.16 N. Apthorpe et al. [ARF17]: Network traffic send/receive rates of selected IP streams from 4 commercial IoT devices during controlled experiments . . . 40

2.17 R. Doshi et al. [DAF18]: IoT DDoS detection pipeline 44
2.18 H. H. Pajouh et al. [Paj+16]: Two-tier classification module 45
2.19 Y. Mirsky et al. [Mir+18]: An illustration of Kitsune's Architecture 47
2.20 T. Luo et al. [LN18]: Architecture of a WSN that uses autoencoders for anomaly detection 48
2.21 M. Rigaki et al. [RG18]: Network experiments setup. The GAN is implemented independently and communicates with the malware through a web service. The malware gets the parameters and modifies its traffic in real time. The C2 channel should be maintained and should be operational. The IPS blocks all the traffic that does not look like Facebook chat 51
2.22 Q. Yan et al. [Yan+19]: Overview of DoS-WGAN architecture 53
2.23 Z. Lin et al. [Lin+19]: CGAN training mechanism 55
2.24 A. Cheng [Che19]: Conversion and one-to-multi mapping process . . 56

3.1 Network traffic classification pipeline 60
3.2 Experimental smart home network 62
3.3 Dataset visualization using t-SNE 64
3.4 Overall accuracy achieved by Random Forest classifier for different values of N 67
3.5 Sparse Autoencoder 71
3.6 Proposed anomalous communications detection architecture using a set of sparse autoencoders (SAE) when the type of device producing the network traffic is unknown 73
3.7 ROC curves of the different sparse autoencoders in the case of N=10 76
3.8 AUC of the different sparse autoencoders and for different values of N 77
3.9 False positive rate (FPR) and True Positive Rate (TPR) of the individual sparse autoencoders (each trained for a specific device) for different threshold values. 78
3.10 False positive rate (FPR) and True Positive Rate (TPR) of the set of sparse autoencoders (SAE) for different threshold values. The performance of the set of SAE is also compared to other machine learning models. 79

4.1 Motivation for IoT network traffic generation: Scenario (a): realistic network traffic generation for NIDS performance evaluation; Scenario (b): data augmentation to train machine learning based NIDS; Scenario (c): mimicry attack generation for data exfiltration purpose. . 85
4.2 Bidirectional flow generation pipeline: the sequence of packet sizes generation module is followed by the duration determination module 86

4.3 Word by word text generation: When generating the next word of the sequence, the generator actually provides a probability distribution over the vocabulary. The actual sequence is constructed by picking the next word from this probability distribution. 90
4.4 Combining an autoencoder with a GAN to generate sequences of categorical values 92
4.5 Cumulative distribution function of the total number of packets per bidirectional flows produced by the Google Home Mini (a partial view is presented for better clarity). 90% of the flows contain 42 packets or less. 94
4.6 Architecture of the autoencoder used for the experiment 95
4.7 Architecture of the WGAN used for the experiment 96
4.8 Simplified packet ordering - Comparison of the distribution of packet sizes (a, b, c), the number of packets per bidirectional flow (d, e, f), the number of bytes per bidirectional flows (g, h, i) for different models (autoencoder/WGAN-GP, autoencoder/WGAN-C, VAE) . . 97
4.9 Realistic packet ordering - Comparison of the distribution of packet sizes (a, b, c), the number of packets per bidirectional flow (d, e, f), the number of bytes per bidirectional flows (g, h, i) for different models (autoencoder/WGAN-GP, autoencoder/WGAN-C, VAE) . . 102
4.10 Duration distribution for the bidirectional flow F_1 106
4.11 The mixture density network can represent general conditional probability densities $p(t|x)$ by considering a parametric mixture model for the distribution of t whose parameters are determined by the outputs of a neural network that takes x as its input vector [Bis06] 106
4.12 MDN architecture 109
4.13 Simplified packet ordering - Comparison of real duration values with durations generated by the MDN trained using sequences of packet sizes 110
4.14 Realistic packet ordering - Comparison of real duration values with durations generated by the MDN trained using sequences of *(size, direction)* tuples 110

List of Tables

1.1 Top user names and passwords used in IoT attacks according to Symantec [Sym20] 5

1.2 A. Costin et al. [CZ18]: delays between IoT malware sample discovery, capture and analysis, and public release of the corresponding IDS rules 7

2.1 Examples of shallow learning algorithms 21

2.2 Summary of works on IoT network traffic classification for device type or device state determination 41

2.3 Limitations of existing works on IoT network traffic classification for device type or state determination 42

2.4 Summary of works on intrusion detection in IoT networks 49

2.5 Limitations of existing works on intrusion detection in IoT networks 50

2.6 Summary of works on network traffic generation using generative deep learning models 57

2.7 Limitations of works on network traffic generation using generative deep learning models 57

3.1 Functionalities provided by each device and explored during the network traffic collection phase 62

3.2 Total number of bidirectional flows per device 63

3.3 Best hyperparameter values for the different classifiers 65

3.4 Overall performance on the test set of the different classifiers 65

3.5 Precision on the test set of the different classifiers and for specific devices 65

3.6 Recall on the test set of the different classifiers and for specific devices 66

3.7 F1 score on the test set of the different classifiers and for specific devices 66

3.8 Features used to describe bidirectional TCP flows 70

3.9 Total number of bidirectional flows per device 75

4.1 Simplified packet ordering - Earth mover's distance (10^{-4}) between the real and generated traffic histograms of Figure 4.8. WGAN-C based model achieves the smallest distance. 98
4.2 Simplified packet ordering - TPR and FPR on the test set achieved by the trained anomaly detectors 99
4.3 Simplified packet ordering - FNR when the anomaly detectors are fed with synthetic flows ($FNR_{synthetic}$) compared to the FNR and TNR on the test set 100
4.4 Realistic packet ordering - Earth mover's distance (10^{-4}) between the real and generated traffic histograms of Figure 4.9 WGAN-C based model achieves the smallest distance. The EMD obtained under the simplified ordering assumption are shown in brackets. 103
4.5 Realistic packet ordering - TPR and FPR on the test set achieved by the trained anomaly detectors 103
4.6 Realistic packet ordering - FNR when the anomaly detectors are fed with synthetic flows ($FNR_{synthetic}$) compared to the FNR and TNR on the test set 103

1

Introduction

"Once you learn to read, you will be forever free."

– Frederick Douglass

The total number of Internet of Things (IoT) devices is expected to reach 75 billion by 2030 [Lou]. The IoT will encompass all aspects of our life, covering a wide range of applications, such as home automation, smart transportation, smart agriculture, wearable devices or e-Health. The vulnerable nature of IoT devices make them easy targets for botnet developers. The raise of IoT malware introduces new security challenges for network administrators. In this section, we first start by describing the inherent characteristics of IoT ecosystems that make them very challenging to secure. Then, we present the most common types of vulnerabilities encountered in IoT ecosystems and how IoT botnets operates. We then describe a number of solutions proposed to secure the IoT along with their limitations. Finally, we present the objective of this thesis putting special emphasis on the limitations of existing works and how our contributions are going to overcome them.

1.1 IoT Ecosystem Characteristics

IoT systems are designed for diverse purposes (smart home, smart transportation, e-health, etc.), different from traditional computer systems such as personal computers or smartphones. They differ in many aspects from traditional systems in that their inherent characteristics, such as their heterogeneity, pervasiveness, mobility and resource constraints, make them very challenging to secure. Those specific characteristics are extensively described in [Zho+18b; Zha+14; Cha+19a; Ala+17; Fru+17; Mah+15] and are presented in Figure 1.1. Below, we propose to describe

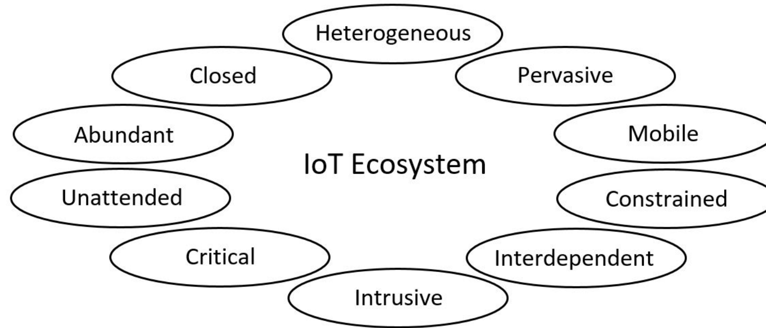


Figure 1.1: IoT ecosystem characteristics

each of them:

- **Heterogeneous/Diverse**: the IoT ecosystem is very diverse in many ways. It consists of a wide variety of application fields, communication protocols and hardware platforms. The fields of application of the IoT are endless. They include intelligent transportation systems, smart homes, smart cities, smart grids, connected health, wearables, smart industries and many more. Moreover, within each field of application, the IoT is used for a number of different use cases. For example, a smart home consists of devices as diverse as refrigerators, thermostats, bulbs, plugs or motion sensors. The IoT is also heterogeneous in terms of the communication protocols it employs. Communication protocols used by IoT devices can be as diverse as Bluetooth, WiFi, ZigBee or Z-Wave. Heterogeneity of the IoT also refers to the diversity of hardware platforms used: x86, ARM, MIPS, etc. The diversity of the IoT ecosystem makes it difficult to define general best practice rules for programmers or network administrators. It is also not possible to develop a one-size-fits-all security solution as the security requirements might vary depending on the application, communication protocol or hardware platform used.

- **Pervasive**: it describes the fact that IoT devices are becoming part of every aspect of our lives. In the literature, this characteristic is also sometimes referred to as the ubiquitous nature of the IoT. In the near future, IoT devices will be everywhere around us. They will become essential to our daily life and we will rely more and more on them. Devices will no longer need human intervention to function properly, such as a bulb connected to a motion sensor that turn on or off automatically. Most people will not even realize how dependent on IoT they are until an attack occurs. Despite its ubiquitous nature, most people usually ignore the security implications of the IoT.

- **Mobile**: this refers to the fact that IoT devices move from one network to another. For example, a smart car that moves from one place to another might connect to a number of different networks along the way. Once connected to a new network, a smart device might need to communicate with other

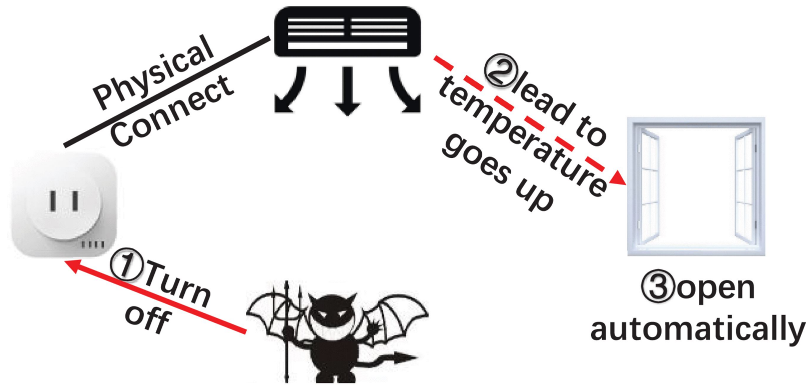


Figure 1.2: Attack example resulting from interdependence between devices by W. Zhou et al. [Zho+18b]

devices present in the same network. For example, a smart car might exchange information with the other cars present nearby or with roadside units. This ability of IoT devices to hop from one network to another make them targets of choice for malware. First, it makes them vulnerable as it increases the likelihood of them being connected to a compromised network and getting infected by a malware. Then, as the device moves from one network to another, it may facilitate the propagation of the malware.

- **Constrained**: IoT devices are resource-constrained. Some smart devices, such as sensors, agricultural devices or medical implants, need to be lightweight and able to work for a long period of time without the need for charging. Hence, those devices have very limited memory and computation capabilities. They do not have a Memory Management Unit (MMU), and proper memory safety measures, like address space layout randomization (ASLR), are not implemented. Those devices cannot implement complex resource-intensive cryptographic operations either. Hence, advanced encryption or authentication algorithms cannot be used to protect them. The use of an antivirus is also not possible as it requires a fair-amount of computational power.

- **Interdependent**: it describes the situation of a device controlling the state of another device. IoT devices can communicate with each other to self-adapt to their environment without resorting to human intervention. For example, if a thermometer detects a room temperature that exceeds a certain threshold, the air conditioner is turned on or the windows are opened as illustrated in Figure 1.2. An attacker can bypass the defense mechanisms of a targeted device by, first, compromising a poorly configured device that has interdependent relationships with the targeted device. Hence, a highly protected device can become vulnerable if it has a single interdependent relationship with a vulnerable device.

- **Intrusive**: some IoT devices might collect sensitive personal information rais-

ing privacy concerns. For example, medical devices or wearables might monitor information about your heart rate, your blood pressure, your location or your daily physical activities. In a smart home, motion sensors or security cameras also collect sensitive personal information. Most of the time, the data collected by those devices are made available to the user through a smartphone application. Hence, the sensitive personal data can be leaked if the device, the application or the communication channel between the device and the application are not sufficiently secured.

- **Critical**: some sensitive IoT applications are highly critical, such as medical applications, autonomous vehicles or industrial control infrastructures deployed in nuclear power plants. In such cases, device availability might be very important and any compromise of the device can lead to severe physical and human damages. For example, the possibility for criminals to take control of cars to cause deliberate accidents will be a serious security issue in the near future [Ind20].

- **Unattended**: some devices such as smart meters, medical implants or industrial sensors need to operate for a long period of time without the possibility of physical access. Authors in [Zho+18b] refer to this characteristic as the long time unattended status of an IoT device. As those devices are not physically accessible, it is very difficult to monitor their states and detect any compromise.

- **Abundant**: the data generated by IoT devices are abundant. As IoT devices are proliferating rapidly, they will soon generate huge amounts of data. In [IDC20], it is estimated that there will be 41.6 billion connected IoT devices, generating 79.4 zettabytes (ZB) of data by 2025. Most of these data will be generated by video surveillance applications. Because IoT devices are not as much protected as laptops or smartphones, a malware can rapidly infect a significant number of them. The malware can then leverage the abundant data generation capability of the compromised IoT network to perform large scale Distributed Denial of Service attacks (DDoS).

- **Closed**: in [Fru+17], the authors point out the fact that most IoT devices are "closed" because they do not provide any way to install additional software once the device has been manufactured at the factory. Hence, the customers cannot easily install an antivirus or any additional security software on the device.

Characteristics such as heterogeneity, pervasiveness, mobility, resource-scarcity, interdependency, intrusiveness, criticality, abundance, autonomy (the fact that IoT devices can be left unattended) and closed nature make IoT systems particularly challenging to secure. Those characteristics coupled with the fact that IoT devices are full of vulnerabilities make an IoT environment a target of choice for an attacker.

Table 1.1: Top user names and passwords used in IoT attacks according to Symantec [Sym20]

user names	percent	passwords	percent
root	38.1	123456	24.6
admin	22.8	[BLANK]	17.0
enable	4.5	system	4.3
shell	4.2	sh	4.0
sh	1.9	shell	1.9
[BLANK]	1.7	admin	1.3
system	1.1	1234	1.0
>/var/tmp/.ptmx && cd /var/tmp/	0.9	password	1.0
>/var/.ptmx && cd /var/	0.9	enable	1.0

1.2 Vulnerabilities in the IoT

In addition to the previously mentioned inherent characteristics, a general lack of security awareness, from both the manufacturers and the consumers, contributes to the vulnerability of IoT devices. Since manufacturers require a short time to market, they see security as an unnecessary additional cost. On the other hand, customers are completely unaware of the security issues related to IoT devices and usually ignore simple security best practices, such as changing the default password of a device. Here, we present the most common types of security vulnerabilities encountered in IoT systems:

- **Weak credentials** [Wan+17; BI17; KKS17]: weak, guessable, or hardcoded passwords have been reported [OWA20] as being the most common vulnerability in the IoT. Most IoT devices are shipped with weak default passwords. The end users rarely change the default user name and password, and when they do, the new credentials are often easily guessable for lack of password security literacy. Moreover, most manufacturers do not enforce the use of a strong password. Sometimes, the password is hardcoded and there is no possibility for the user to change it. The use of weak credentials make IoT devices vulnerable to brute-force attacks. A malware, such as GoScanSSH performs brute-force attacks using dictionaries containing more than 7,000 unique credential pairs [CZ18]. Table 1.1 shows the top user names and passwords used in IoT attacks in 2018 according to Symantec's Internet Security Threat Report [Sym20].

- **Backdoors** [Nes+19; Zha+14; KKS17; Sac+17]: very common among IoT devices, they are deliberately inserted by vendors for management and testing purposes. This includes unnecessary open ports running services such as Telnet, SSH or FTP. Examining a device before deployment is not sufficient, as backdoors can be inserted during system upgrades or when the device receives patches for security updates. Those backdoors can be discovered through port scanning. An attacker can leverage those backdoors to get active remote ad-

min capabilities and get full control of the device. The infected device might then be used as a pivot to launch attacks against other devices. Moreover, devices with ports running vulnerable services are susceptible to buffer overflow or DDoS attacks that can render them unavailable.

- **Software vulnerabilities** [Nes+19; Zha+14]: IoT devices often contain a plethora of software vulnerabilities such as buffer overflow or authentication bypass. Software vulnerabilities, especially 0-day vulnerabilities, are exploited by malware to take full control of a device. Because of the heterogeneity of the IoT (diverse hardware platforms, protocols and applications), it is very difficult to define best practice rules for programmers [Zha+14; Nes+19]. Moreover, because of the wide variety of architectures involved (different from personal computers that run predominantly on x86 architecture), most IoT devices use a Linux-based OS [Coz+18]. However, smart devices such as fridges, bulbs or plugs do not really need all the functionality provided by a Linux OS. The use of a Linux OS increases the attack surface and makes the devices vulnerable to legacy attacks found in the desktop world [KKS17]. Because of their long term unattended nature, lots of IoT devices often use legacy OS or outdated components that are no longer maintained. According to Palo Alto [Alt20], 83% of medical imaging devices run on unsupported operating systems. This is mainly due to Windows 7 reaching end of life.

- **Poor software update policy** [CZ18; Nes+19; KKS17]: because of a poor software update policy, most discovered vulnerabilities are not even properly patched. In [CZ18], the authors also point out the significant delay between the first malware sample discovery and the public release of a security solution. As shown in Table 1.2, once a new IoT malware has been identified, it can take from tens to hundreds of days for a first security solution to be publicly available. Whereas the majority of malware spread the most within the first hours or few days of their existence. In [Nes+19], the authors report the fact that many manufacturers do not have an automated patch-update mechanism (mechanism that ensures that as soon as a patch-update is available, it is sent automatically to all the devices without requiring the end user to ask for it). They also report cases of lack of integrity guarantee of the available security updates, making them susceptible to malicious modifications. For some devices, the software update process is neither signed nor protected by end-to-end encryption [KK16; BW20; OWA20].

- **Insecure interfaces** [KS18; BI17]: Web and mobile application interfaces used to manage and configure IoT devices are often insecure and poorly designed. Most of the time, those interfaces are vulnerable to attacks such as SQL injection or Cross-site Scripting (XSS) that can lead to sensitive information leakage. In [BW20], the authors reported to have found 10 security issues in 15 web portals used to control IoT devices, including serious issues

Table 1.2: A. Costin et al. [CZ18]: delays between IoT malware sample discovery, capture and analysis, and public release of the corresponding IDS rules

	Mean (days)	Median (days)
Delay between the first seen in the wild sample of malware family and corresponding IDS rule first public release	675	166
Delay between the first submitted for analysis sample of malware family and corresponding IDS rule first public release	241	63
Delay between the first technical analysis of malware family and corresponding IDS rule first public release	32	25

that can lead to unauthorized access to the backend systems. Sometimes, the interface may not offer the possibility to change the default password. Robust password recovery mechanism can also be absent [BI17; OWA20]. Interfaces might not implement account lockout mechanism making them vulnerable to brute-force attacks. They might also be susceptible to account enumeration attacks.

- **Lack of proper encryption** [Nes+19; Ala+17; BW20; BI17; RS16]: most IoT devices fail to implement a proper encryption mechanism to protect the data transmitted over the local network or to the Internet. This allows the data to be viewed while traveling over the network rising serious privacy issues. The problem is exacerbated because most IoT devices communicate through wireless networks making them even more vulnerable to eavesdropping attacks. In [RS16], the authors pointed out that, during the installation phase of certain smart bulbs, the WiFi passwords are transmitted unencrypted. In [BW20], the researchers report that 19% of the IoT device control mobile apps they tested were not using SSL connections to connect to the cloud. Moreover, the resource limitation of smart devices reduce robustness and effectiveness of cryptographic algorithms making it possible for an attacker to circumvent the deployed encryption system [Nes+19].

- **Insufficient authentication and authorization mechanism** [Nes+19; Fru+17; KS18; Ala+17; BI17; BW20]: because of constrained resources, smart devices often do not implement sufficient authentication mechanism. In [BW20], the authors report that, at the time of writing, the Belkin WeMo connected switch did not require the user to authenticate to connect to it. Hence, an attacker located in the same network as the device can send any command to it. They also reported that the LightwaveRF smart hub communicated with a TFTP server on the Internet without requiring any authentication, allowing an intruder to perform Man-in-the-Middle (MitM) attacks. IoT devices also do not provide any granular role-based access control mechanism making it possible for the collected data to be shared with unauthorized entities [Ala+17].

- **Poor physical security** [Nes+19; Fru+17; KS18; BI17]: because most IoT devices are deployed in the nature in an unattended way, it is easier for an attacker to get physical access to them. IoT devices rarely implement physical hardening measures making them vulnerable to attackers with physical access. Most devices often provide debugging or USB ports that can be used to extract sensitive information from the device or modify the memory and the configuration settings of the device. This includes unveiling employed cryptographic schemes, issuing a new device pairing request, configuring a new password, or installing a forged SSL certificate [BW20; Nes+19]. A skilled attacker can even read the firmware of a device to understand how the device works, find vulnerabilities in it, or forge a malicious version of the firmware. Second-hand IoT device markets are also great places for attackers to sell intentionally compromised devices.

IoT devices are subject to a tremendous number of vulnerabilities: the use of weak passwords, the presence of backdoors and software vulnerabilities, an inefficient software update policy, insecure web and mobile interfaces, lack of proper encryption and authorization mechanisms, or a poor physical security. Most of those vulnerabilities are the result of a lack of security awareness from both the manufacturers and the customers.

1.3 IoT Botnets

As it is established that IoT devices are particularly vulnerable, they have become low-hanging fruits for attackers who want to turn them into bots. Bots are compromised devices that can be used to launch large scale and coordinated attacks.

Extensive study has been conducted to characterize IoT Botnets [Kol+17; Ang17; KKS17; Ant+17; Pa+15; Wan+17; Jer17; CZ18]. Most studies focus on Mirai or Bashlite as most IoT malware are highly inspired from them and present similar behavior. As illustrated in Figure 1.3, the Mirai botnet is composed of four types of components: the bots, the command and control server (C&C), the loader and the report server. A bot is a compromised device that is waiting to receive commands from the botmaster (the attacker controlling the botnet) in order to scan for vulnerable devices or launch DDoS attacks against designated targets. The C&C provides a centralized management interface for the botmaster to control the botnet and launch attacks. The loader contains malware executables for different architectures (ARM, MIPS, x86, ...) and is used to infect new devices. The report server contains a database with details about all devices being part of the botnet.

The behavior of a Mirai-like malware can be split into two phases: the propagation phase and the attack (or monetization) phase (some work further split the propagation phase into a reconnaissance phase and an infection phase [Pa+15]):

- *Propagation phase*: first, a bot scans the network searching for vulnerable devices running the Telnet service on TCP ports 23 or 2323. Once a weakly

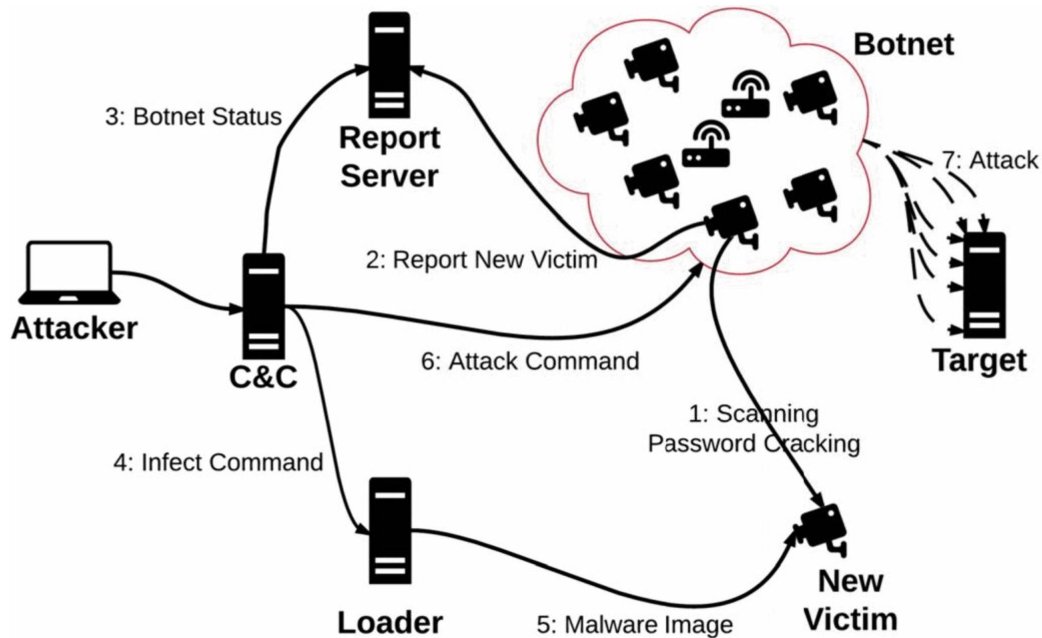


Figure 1.3: Overview of Mirai botnet communication and basic components from [KKS17]

configured device has been identified, brute forcing is used to find out the correct credentials of the device. There are 62 possible username/password pairs hardcoded in Mirai. Upon successful first login, the IP address of the victim, the associated credentials and information about various characteristics of the device are sent to the report server. The C&C is used by the botmaster to regularly check the status of the report server. And when a new vulnerable device is available, the C&C issues an "infection" command to the loader with all the necessary information about the device to infect. The loader then logs into the targeted device, determines its underlying environment and instructs it to download and execute the architecture-specific malicious binary. To hide its presence, Mirai deletes the downloaded binary file. Any other processes bound to TCP ports 22 or 23, as well as processes associated with competing malware are killed. The newly infected device is now waiting for attack commands from the C&C while scanning the network for new vulnerable devices.

- *Attack phase*: to launch an attack, the botmaster sends an attack command, through the C&C, to designated bots, specifying the targeted server and the type of attack to be carried out. Mirai can launch various attack types including UDP flood, SYN flood, DNS water torture or GRE IP flood. Upon reception of the attack command, the selected bots will start to launch the specified attack against the designated target. Mirai was mainly involved in large scale DDoS attacks like the one that took place in October 2016 against the servers of Dyn, a large DNS provider, and brought down large parts of the Internet [Gua20].

Mirai primarily infected cameras, DVRs and home routers [Ang17]. At its peak, Mirai infected more than 600,000 devices worldwide [Ant+17]. The most affected countries were Brazil, Vietnam, China and Columbia. Since Mirai, IoT malware have considerably evolved and are getting more and more sophisticated. Below we list a few other notable IoT botnets that operate differently than Mirai:

- *Hajime* [EP16; Kol+17] (2016) infects devices that have their Telnet port open, similarly to Mirai. But instead of relying on a centralized architecture, it is built on a P2P network. Bots act as both a command server and a client which receives commands. Contrary to centralized botnets, P2P botnets are more resilient as they avoid any single point of failure. It uses BitTorrent's DHT protocol for peer discovery and uTorrent Transport Protocol (uTP) for data exchange. Messages are encrypted and signed using an RC4 cryptographic scheme. Hajime has never been involved in any attack so far.

- *Reaper* [Rad20b] (2017) is an IoT botnet that, instead of performing credentials brute-forcing, leverages HTTP-based exploits of known IoT vulnerabilities to infect devices.

- *BrickerBot* [Rad20a; Kol+17] (2017) targets Linux/BusyBox-based IoT devices and performs permanent denial-of-service (PDoS) attacks against them. It does not try to upload any binary to infect a device. Rather, it performs a set of commands to render the device unusable. It uses various methods such as altering the firmware of a device, erasing all files from its memory or reconfiguring its network parameters.

- *Persirai* [Mic20; Kol+17] (2017) is built upon the open-sourced Mirai code and targets over 1,000 IP Camera models. IP Cameras often use Universal Plug and Play (UPnP), a network protocol that allows devices to open a port on the router and act like a server exposing them to the Internet. Persirai gets access to the IP Camera's web interface through TCP port 81 and performs a command injection to force the IP Camera to download and execute a malicious shell script. Persirai does not use brute forcing to get the credentials, rather it exploits a 0-day vulnerability to directly obtain the password file.

- *HideNSeek* [Bit20; Ava20; Har19] (2018) is an IoT malware that uses a home-made P2P protocol for communications, and relies on both known IoT vulnerabilities exploitation and credentials brute forcing to infect devices. New versions of HideNSeek achieve persistency (are capable of surviving a device reboot).

- *BlackIoT* is described by S. Soltan et al. [SMP18] as a possible new type of botnets in the near future made of vulnerable high wattage IoT devices (such as air-conditioners and heaters) that can attack power grids. They define a new type of attack called Manipulation of demand via IoT (MadIoT). Through

simulations, they show that such attacks can result in local outages or large blackouts in the power grid.

Because of their vulnerable nature, IoT devices are prime targets for botnet developers. IoT botnets are rapidly evolving and are getting more and more sophisticated. They are primarily used to perform large-scale DDoS attacks. But with more and more smart appliances with the ability to connect to the Internet, other types of threats are emerging such as MadIoT attacks against power grids. A number of solutions have been proposed to secure the IoT ecosystems and are described in the next Section.

1.4 IoT Security Solutions

In this Section we briefly describe a number of solutions that have been proposed to secure the IoT. We also point out their limitations. User level solutions, software level solutions, cryptographic solutions and network level solutions are discussed.

User Level Solutions

Given that a large number of IoT botnets infect devices by performing brute-force attacks, one simple solution is for the consumer to set strong credentials. Manufacturers can force the user to set a strong password during the setup phase of a device. However, such a solution assumes that both the end user and the manufacturer are aware of the security issues related to the IoT and are willing to fundamentally change their habits. Moreover, enforcing the use of strong credentials will not prevent a botnet from infecting devices by exploiting software vulnerabilities or backdoors inserted by the manufacturer for management and testing purposes.

Software Level Solutions

Because IoT devices are closed in nature, no additional software such as antivirus can be installed by the end users. Therefore, they must be secure-by-design. Vendors must evolve their security concept from traditional "add-on-security" (security can be added later) that is appropriate for laptops or smartphones, to "built-in-security" [Fru+17]. For example, instead of using a Linux OS that increases the attack surface, lightweight OSs like RIOT OS or Google's Brillo are much safer options for IoT devices [KKS17]. Those lightweight OSs provide enough functionalities for a smart device to perform its specific task. Although reducing the attack surface, lightweight OSs are limited in that they can still contain software vulnerabilities. To overcome this shortcoming, cybersecurity-oriented distributions such as RIOT-fp [20c] are emerging. However, strong software level security do not prevent the use of weak credentials or the insertion of backdoors by manufacturers.

Cryptographic Solutions

As IoT devices have very limited memory and computation capabilities, widely used cryptographic algorithms, such as RSA, DES, 3DES and AES are not appropriate. Designing resource efficient lightweight cryptographic scheme for the IoT is an active research area [Sin+17; CGP18; SNB18].

Cryptographic algorithms are used for three main security purposes: confidentiality, authentication and integrity. Symmetric algorithms (like AES) are used to ensure confidentiality while asymmetric algorithms (like RSA) are mainly used for authentication purposes or to exchange symmetric keys. Hash functions are often used to ensure data integrity. Hence studies focus on the development of three types of lightweight cryptographic algorithms: symmetric algorithms, asymmetric algorithms and hash functions.

A. Bogdanov et al. [Bog+07] proposed PRESENT, a lightweight symmetric block cipher. It offers high level of security with a 64-bit block size and an 80-bit key. Hardware implementation of PRESENT requires 1570 GE (Gate Equivalence) which is competitive when compared to the 3400 GE required by AES-128 or 2309 GE required by DES. G. Leander et al. [Lea+07] proposed DESL, a lightweight variant of classical DES algorithms. They replaced the eight original S-boxes in classical DES by a single but cryptographically stronger S-box repeated eight times. Hardware implementation of DESL requires 1848 GE.

Asymmetric lightweight algorithms are mostly based on Elliptic Curve Cryptography (ECC). ECC is based on the difficulty to solve the elliptic curve discrete logarithm problem. It provides the same level of security as RSA but with a smaller key size. As a consequence, it has a fast processing speed and requires less memory making it well suited for IoT devices. Several works have proposed ECC based cryptographic schemes for authentication in an IoT environment [KS15; Kum+18; HZ14].

J. Guo et al. [GPP11] proposed PHOTON, a lightweight hash-function based on the AES design strategy and with performance very close to the theoretical optimum (in terms of the minimal internal state memory size). Another example of lightweight hash functions is QUARK (JP. Aumasson et al.) [Aum+10].

To secure the IoT ecosystem, many research works focus on designing secure lightweight cryptographic algorithms. It is obvious that encrypting the transmitted data will protect against eavesdropping attacks; authenticating devices can prevent MitM attacks; and data integrity verification will prevent from malicious data tampering. However not all malicious activities can be countered by the use of cryptographic means. For example, a poorly configured IoT device that uses weak credentials can be infected by a botnet through brute-force attacks despite its communication being secured with cryptographic means. Network traffic encryption can even make the work of intrusion detection systems harder as they are no longer able to access certain headers or payloads.

Network Level Solutions

At the network level, traditional tools to enforce security are firewalls and Network Intrusion Detection Systems (NIDS).

A firewall is software or hardware used to enforce a network security policy. It consists of a set of network security rules used to monitor and control incoming and outgoing network traffic. It is used to partition the network into safe and unsafe zones. There exists two type of firewalls: host-based firewalls and network-based firewalls. A host-based firewall runs on a device and filters the network traffic sent or received by the device. A network-based firewall runs on a network hardware like a router and filters the traffic between two or more networks. A major limitation of host-based firewalls is that they require a lot of computational power from the host to analyze, monitor and filter the network traffic. Because IoT devices are resource-constrained in nature, the use of a host-based firewall is not a viable solution. A network-based firewall can be used to secure a private IoT network by filtering its communication with the Internet. It will prevent the smart devices of the private network from being infected by a botnet located in the external network. However, IoT devices are characterized by their heterogeneity and mobility limiting the efficiency of a network-based firewall. Indeed, the heterogeneity of network protocols for example (Bluetooth, WiFi, ZigBee or Z-Wave, etc.) induces the deployment of multiple firewalls each dedicated to a particular protocol. Because of its mobile nature, an IoT device can also connect to an unsecure external network, and get infected by a malware. Moreover, network-based firewalls are ideal when the communications of all the devices in the network are centralized, which is not necessarily the case with IoT devices, as interdependent devices might communicate with each other directly. Some devices like security cameras run web services on specific ports to provide web interfaces that can be accessed from the Internet (if the user is not at home for example). Hence, the firewall must be configured to allow connections coming from the Internet to connect to the security cameras. In such case, the firewall is of no use to block malware like Persirai that exploit vulnerabilities present in the exposed Web interface of a device.

NIDSs are hardware or software based systems that inspect the network traffic looking for malicious activities. They are usually placed at strategic points within a network so that they can monitor traffic to and from all the devices on the network. Once an attack is identified, or an abnormal behavior is detected, an alert is sent to the administrator. When the NIDS has a response capability, it is referred to as a Network Intrusion Prevention System (NIPS). An NIPS differs from an NIDS in that it is capable of altering flows of network traffic to block an ongoing attack for example. An NIDS can be either signature-based or anomaly-based. In signature-based NIDSs, a specific pattern (sequence of TCP flags, sequence of bytes, sequence of instructions, TTL value, etc.), also called a signature, is indicative of a specific attack. Each attack type, such as SYN flood or TCP half-open port scan, has its specific signature that needs to be stored in the database of the NIDS. The main limitation

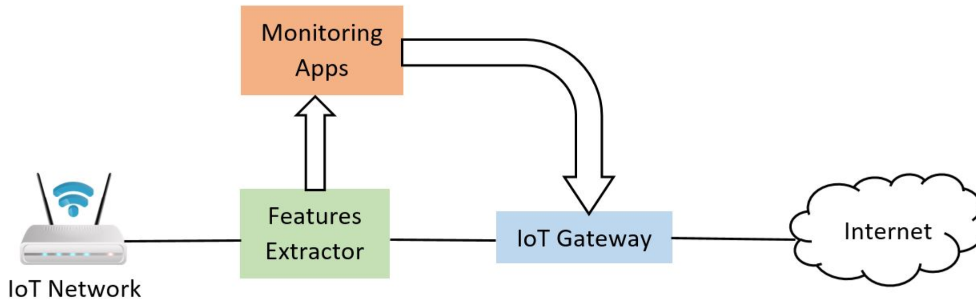


Figure 1.4: IoT network architecture

of signature-based NIDSs is that they can only detect attacks for which a signature exists. Hence, they completely fail to detect new attacks such as attacks exploiting zero-day vulnerabilities. An anomaly-based NIDS learns the profile of legitimate networking behavior. When the network traffic does not comply with the learnt legitimate behavior profile, an alert is triggered. Contrary to signature-based NIDSs, anomaly-based NIDSs can detect previously unknown attacks. However, they are susceptible to higher false positive rates because legitimate traffic can sometimes deviate from the learnt profile, triggering false alarms.

As IoT devices are intended to perform very specific tasks, tasks that remain the same over time, it makes their behavior very stable and predictable. For example, a smart plug can only be switched on or off. However, it is not supposed to send emails or click on ads. The network traffic produced by IoT devices being very predictable, it is well suited for machine learning techniques. Therefore, developing machine learning based NIDSs and network monitoring tools seems to be a promising way to secure IoT networks. In contrast, applying machine learning techniques for intrusion detection in the case of general purpose devices, such as desktop computers, laptops or smartphones, has often proved difficult due to the great variability and diversity of the generated network traffic [SP10]. Recent advances in the field of machine learning, especially the emergence of efficient deep learning models can be leveraged to develop new IoT network monitoring solutions. Deep learning algorithms however require huge amount of data to be trained on. Hence, one needs to find new cost-efficient ways to generate IoT network traffic data.

In this thesis, we attempt to leverage recent advances in the field of machine learning to secure IoT networks and seek to answer the following questions:

- How can deep learning help to monitor IoT networks?
- How can deep learning help to overcome the lack of IoT network traffic data?

1.5 Objectives of the Thesis

1.5.1 Context

Figure 1.4 shows the main components of a machine learning based network security solution for the IoT. The IoT devices connect to the local network through a wireless access point. The IoT gateway connects the local IoT network to the Internet. By IoT gateway, we refer to a router in a traditional network or an SDN switch in an SDN-based network. The IoT gateway can be used to filter the network traffic based on predefined rules. The features extractor operates between the wireless access point and the IoT gateway. Its role is to extract specific features from the raw network traffic and send them to the monitoring applications. Features are used to describe the raw network traffic. Examples of features include the number of packets per flow, the number of bytes per flow, the duration of the flow, and so on. The monitoring applications are IoT device identification tools, IDS or IPS. They consist of trained machine learning models that take as input the features provided by the features extractor. Depending on the output of the models, the monitoring applications can push new filtering rules to the IoT gateway. In a traditional network, the features extractor is a device with enough memory and computational resources. The monitoring applications can run on a server within the local network or even on the same device as the features extractor. In an SDN environment, the IoT gateway will be an SDN switch and the monitoring applications will be hosted by controller applications. Thanks to protocols, such as OpenFlow, communications between the controller and the IoT gateway can be easily supported. As for the features extractor, it can be integrated into the SDN switch by programming the packet forwarding plane using languages such as P4. This thesis focuses primarily on the development of monitoring applications, specifically for IoT device type recognition and IoT network intrusion detection. The study of the other components of the architecture such as the features extractor are let as future work.

Determining the type of an IoT device connected to the network allows to specify device-specific filtering rules that prevent the device from doing anything else than what it is expected to do. It also allows to prevent certain types of device, considered to be vulnerable, from accessing the network. Given the great diversity of IoT devices, it is difficult to come up with a specific network signature for each existing IoT device type. As a consequence, most existing works on IoT device fingerprinting [Mei+17b; Mei+17a; Mie+17; Bez+18] take advantage of machine learning algorithms to learn patterns in network traffic data that can help to determine the type of devices connected to the network. Indeed, contrary to general purpose devices like personal computers, an IoT device is intended to perform very specific tasks that remain the same over time, making its networking behavior very stable and predictable. Hence, data analysis techniques are well suited to model the networking behavior of IoT devices. For this same reason, most existing works on IoT NIDS differentiate malicious activities from legitimate ones using machine

learning algorithms [Mei+18; Mir+18; Ngu+19; DAF18].

However, developing IoT network monitoring tools such as NIDSs requires tremendous amount of IoT network traffic data. The data is primarily used to train and test the developed models. However, very few IoT network traffic datasets are publicly available. Physically deploying thousands of real IoT devices to produce network traffic can be very costly. Moreover, companies or institutions that operate IoT devices are reluctant to share network traffic data because of privacy concerns. Therefore, generating synthetic IoT network traffic data is an alternative. Advances in the field of generative deep learning has prompted a recent interest in leveraging models such as Generative Adversarial Networks (GANs) to generate real-looking synthetic network traffic. Existing works attempt to generate both legitimate and malicious network traffic data [RG18; Rin+19; Lin+19; Che19].

1.5.2 Limitations of Existing Works

Most existing works on IoT device type recognition through network traffic analysis have limitations. Some of them are not *delay-free*, meaning that the proposed approach require the user to wait for a long period of time before being able to determine the type of a device. Other proposed methods are not *phase-independent*. That is, they focus on a specific phase of a device life cycle (like the setup phase) or a specific packet type. Hence, the device type can only be determined under some specific conditions. Some works are limited in that they fail to be *non-intrusive*. They require to look at application level data and hence cannot be used when network traffic is encrypted.

As for IoT NIDSs, a number of them have been proposed in the literature but have some major limitations. Some of them are limited in that they are based on supervised machine learning algorithms, and hence cannot detect new attacks not seen during the training phase. Some works are limited because the type of the device that is generating the network traffic needs to be known beforehand. As a consequence, the developed model must be placed inside the local network because local information like the MAC address or the local port numbers of a device are often needed to determine the type of a device. Other works use local network information to derive features. In case the local IoT network is connected to the Internet through a NAT (which is often the case), local information like MAC addresses or ports used by the device are no longer available beyond the NAT.

Existing works on network traffic generation, either focus on flow-level features or packet-level features generation, but not both at the same time. The resulting traffic is incomplete since a flow and the individual packets composing it are closely related. For example, the number of bytes exchanged for the duration of a flow usually amounts to the sum of the sizes of each packet that composes the flow. Traffic generation based only on flow-level features will fail to fool network monitoring tools that perform packet-level analysis, while traffic generation based only on packet-level features will fail to fool tools that perform flow-level analysis. Moreover, all

existing works on network traffic generation uses non IoT data, often collected from networks composed exclusively of general purpose devices such as PCs, laptops or smartphones.

1.5.3 Contributions

We make the following contributions in this thesis:

- We propose to leverage machine learning to develop IoT network monitoring tools that overcome the limitations of existing works. Two types of monitoring tools will be explored: IoT device type recognition system and IoT network intrusion detection system (NIDS). For the device type recognition system, we will first determine an appropriate set of features. Those features can be extracted from any type of network communication and do not require to look at application layer data, allowing the model to be *phase-independent* and *non-intrusive*. Moreover, they can be determined by observing only a few number of network packets making the model *delay-free*. To get an insight of the selected set of features and assess its representational power, data visualization will be performed. Then, supervised machine learning algorithm will be used to perform network traffic classification and determine the type of the IoT device the traffic belongs to. As for the IoT NIDS, it must be able to detect new types of attack and not require prior knowledge about the type of the device that is generating the network traffic. To this purpose, we will take advantage of autoencoders, an unsupervised neural network to learn the legitimate networking behavior profile and detect any deviation from it. During the training phase, one different autoencoder will be trained for each IoT device type. During the testing phase, we will explore both possibilities: either it is possible to determine the type of the device that is generating the network traffic or it is not (for example when the NIDS is deployed outside the local network).

- We propose to leverage generative deep learning to generate synthetic bidirectional flows that look like they were produced by a real IoT device. To overcome the shortcomings of existing works, we aim at generating both packet-level and flow-level features at the same time. To this purpose, we propose to generate sequences of packet sizes representing bidirectional flows along with duration values. Hence, in addition to generating packet-level features which are the sizes of individual packets, our developed generator implicitly learns to comply with flow-level characteristics, such as the ordering of the packets, the total number of packets and bytes in a bidirectional flow or the total duration of the flow. As a sequence of packet sizes is a sequence of categorical data, our problem is similar to word by word text generation. Inspired by the solutions proposed in the field of Natural Language Processing, we propose to combine Generative Adversarial Networks (GAN) with autoencoders

to generate sequences of packet sizes. As for the duration, its noisy nature is modeled using Mixture Density Networks (MDN), a type of neural networks that output probability distributions allowing to model uncertainty.

The rest of the thesis is organized as follows: Chapter 2 first introduces deep learning and the related concepts that are essential to understand the rest of this thesis. Then, it describes existing works in IoT network traffic classification, IoT NIDSs and network traffic generation, putting special emphasis on their limitations. Chapter 3 describes our proposed IoT network monitoring systems, namely an IoT device type recognition system and an IoT network anomaly detection system. In Chapter 4 is presented our proposed approach for synthetic IoT network traffic data generation. Chapter 5 concludes with possible future work.

2

State of the Art

"Books serve to show a man that those original thoughts of his aren't very new after all."

– Abraham Lincoln

Before going deeper into the description of our contributions, we first need to define key machine learning concepts and have a look at existing solutions. In this chapter, we start by describing what deep learning is and how it can be used to secure IoT networks. We define deep learning concepts and algorithms that will be extensively used in the rest of this thesis. Then, we present a literature review of existing works on IoT network traffic classification, IoT network intrusion detection and deep learning based network traffic generation.

2.1 Deep Learning

In this section, we introduce basic deep learning concepts. We first define what deep learning is and its relation to machine learning. Then, we describe different types of neural network architectures. We also present how deep learning models are trained. Different metrics commonly used to assess the performance of a model are also described. Finally, we end by giving examples of deep learning application in the field of cybersecurity. Note that this section does not exhaustively cover all deep learning concepts but focuses mainly on concepts and methodology that will be important to understand the rest of this thesis.

2.1.1 A Sub-field of Machine Learning

Machine Learning

In classical computer programming, to perform a specific task, the input data is processed through hard-coded rules defined by a programmer. An example of a hard-coded rule for a spam filter can be: if the email contains the word "sale" or "discount", then flag it as a spam. But is it possible to let the computer learn those rules from the data? This is the question that machine learning [Alp20; Gér19; Bis06] tries to answer. Machine Learning is the science of making computers learn from data. A formal definition of learning is given by Tom Mitchell in 1997 [Mit97]:

"A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."

Instead of hard-coded rules, the idea is to let the learning algorithms find the underlying statistical patterns in the input data that can be used to perform a specific task. For example, if a learning algorithm is fed with a sufficient number of spam and non-spam emails, it will find statistical patterns, like emails containing the words "sale" or "discount" being more likely spams.

Depending on the type of supervision they get during training, machine learning algorithms can be categorized into four categories [Gér19]: *supervised learning, unsupervised learning, semi-supervised learning* or *reinforcement learning*:

- in supervised learning, each instance in the training set includes its desired solution also called its *label* or *target value*. Supervised learning is used for tasks such as classification or regression. For example, during training, a spam detector classifier is fed with emails along with their labels (spam or not). After training, the classifier can be fed with new emails to determine if they are spams or not. In regression tasks, the model does not attempt to predict a class but a number like predicting the price of a house. The main drawback of supervised learning is that labeled data is not always available and labelling huge amount of data can be a daunting task.

- in unsupervised learning, the training set is unlabeled and the model tries to learn without supervision. Unsupervised learning tasks include clustering, dimensionality reduction, anomaly detection or data generation. Clustering aims at grouping instances in a way that similar instances end up in the same group, while dissimilar instances are assigned to different groups. Dimensionality reduction is often used for data visualization purposes. During training, an anomaly detection model is fed only with normal instances and it learns the profile of the legitimate or expected behavior. After training, it detects whether or not a new instance comply with the learned legitimate profile. For data generation, models are trained to learn to generate new data that follows the same statistical patterns as the data of the training set.

Table 2.1: Examples of shallow learning algorithms

supervised learning	unsupervised learning
Linear regression, Support Vector Machine (SVM), Decision Tree, Random Forest, k-Nearest Neigbors (kNN)	k-means, Isolation Forest, One-Class SVM, Elliptic Envelope, Principal Component Analysis (PCA)

- in semi-supervised learning, algorithms are able to deal with partially labeled data. Usually a small amount of labeled data and a huge amount of unlabeled data. Semi-supervised learning algorithms are often combinations of unsupervised and supervised learning algorithms.

- reinforcement learning is very different from the other categories. In reinforcement learning, an agent learns by interacting with its environment. It observe the environment, decides to take an action and get a reward in return. The aim of the agent is to find the best strategy to maximize the rewards it get over time.

Deep Learning vs Shallow Learning

Machine learning algorithms can be divided into *shallow learning* and *deep learning* algorithms [GBC16; LBH15; Fra17]. Shallow learning models can only learn one or two layers of representation of the input data [Fra17]. Hence, they have very limited representational capacity and cannot learn complex dependencies between features of the input data. This is why feature engineering is an important step when using shallow learning algorithms. Feature engineering is the process of finding features that contains meaningful information about the representation of the input. For example, instead of feeding the algorithm with raw pixel values of an image, it is better to provide high level features such as the number of circles, the number of vertical and horizontal lines in the image, etc. Table 2.1 shows examples of supervised and unsupervised shallow learning algorithms.

Deep learning is a sub-field of machine learning that focuses on learning successive layers of increasingly meaningful representations of the input data. The learning is performed using *artificial neural networks (ANN)* composed of multiple *layers*. A layer consists of *neurons*, also called *units*. Figure 2.1 shows how the output of a neuron is calculated when the input is a 2-dimensional vector. Let (x_1, x_2) be the input vector, the *output* of the neuron is given by:

$$output = f(x_1 w_1 + x_2 w_2 + b)$$

where w_1 and w_2 are the connections *weights*, and b a *bias term*. f is an *activation function*. Its role is to introduce non-linearity and make the model capable of learning complex non-linear function. When schematically representing neural networks, a simplified representation is preferred as shown in Figure 2.1. All the connection weights of all the units of a layer can be represented in a matrix. Hence,

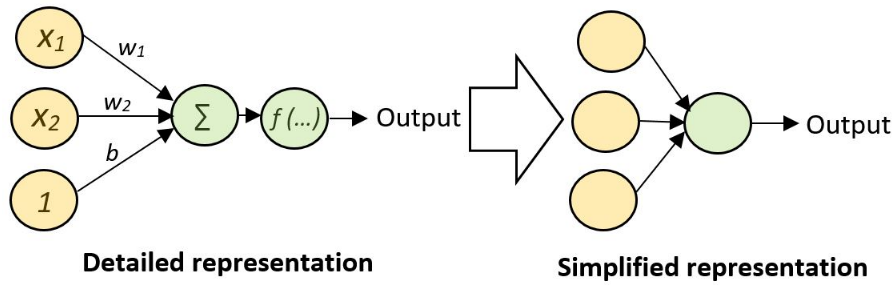


Figure 2.1: Detailed representation of how the output of a neuron is computed and its corresponding simplified representation

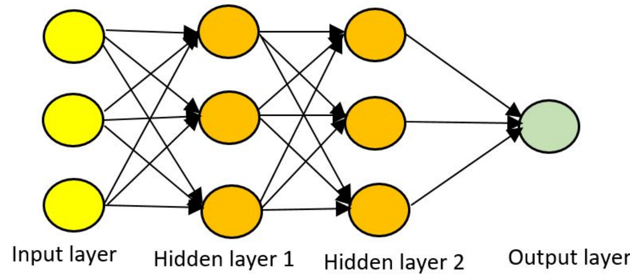


Figure 2.2: Example of an artificial neural network architecture

a layer consists basically of a matrix multiplication followed by an activation function. The depth of a model refers to the number of layers of the model. As shown in Figure 2.2, an ANN typically consists of an input layer, multiple hidden layers and an output layer. Feature engineering is no longer required as successive layers allow deep neural networks to learn very complex dependencies between the input features by themselves. In fact, a neural network can approximate any function of the input data provided that it is sufficiently large.

2.1.2 Neural Network Training

Dataset Partitioning

To train and evaluate a model it is important to split the dataset into three sets: the *training set*, the *validation set* and the *test set* [Gér19; Fra17]. The training set is used to learn the weights, also called the parameters, of the model (see Section 2.1.1). The validation set is used to fine-tune the hyperparameters of the model. The hyperparameters of a neural network can be the number of layers, the number of units per layer or the total number of training iterations. Multiple models, each with a different configuration of the hyperparameters are trained and then evaluated on the validation set to find out the hyperparameters configuration that yields the best result. The test set is only used to assess the performance of the final model. It is used to assess how well the model performs on previously unseen data.

However, when the dataset is small, splitting it might drastically reduce the

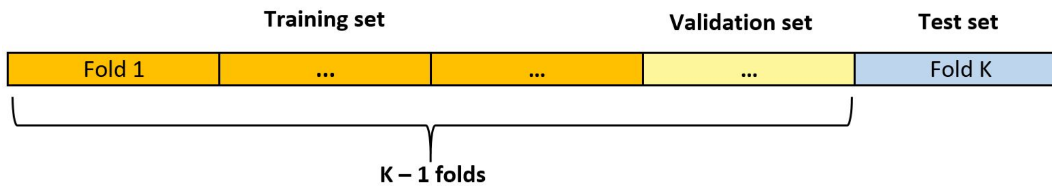


Figure 2.3: K-fold cross-validation

number of samples available either for training or testing. An advanced technique to assess the performance of a model is to use *k-fold cross-validation*. The idea is to partition the dataset into k folds as illustrated in Figure 2.3. The model is learnt using $k-1$ folds (that contains the training and the validation set) and assessed using the remaining folds as the test set. The process is repeated k times with each of the k folds used exactly once as the test set. The performances measured for each fold used as the test set can be averaged to obtain the overall performance of the model. Note that a nested k-fold cross validation can be performed to further split the $k-1$ folds used for learning the model into a training set and a validation set.

Gradient Descent

As explained in Section 2.1.1, each layer of a neural network is parameterized by a weights matrix. The purpose of training is to find the value of the weights such that the neural network will correctly map the input instances to their corresponding targets (the actual value that we expect the model to predict). A *loss function* (also called *cost function*) is defined to measure for each training instance the error between the output predicted by the model and the actual target we expect. It measures how well the model is doing in predicting the target value for a specific input instance. The gradient of the loss function is then used to slightly update the weights of the model in the direction that minimizes the loss. This is called a *gradient descent* step. The magnitude of each gradient descent step is controlled through the *learning rate* which is a hyperparameter of the model. If the learning rate is too small, it will take a long time for the algorithm to converge. While, if it is too big, the algorithm will diverge.

Instead of computing the loss and the corresponding gradient over the whole training set, it is usually computed over a small number of random training instances called a *batch* of training instances. One *epoch* is completed when the learning algorithm has worked through all the training instances once. The batch size as well as the total number of training epochs are hyperparameters of the model.

The loss function minimization through gradient descent steps is performed by an *optimizer* as shown in Figure 2.4. A number of different optimizers have been proposed such as RMSProp [TH12] or Adam [KB14]. The optimizer takes advantage of the *backpropagation* algorithm [GBC16; Gér19]: it first uses the model to make predictions (forward pass) on a batch of training instances, it measures the error

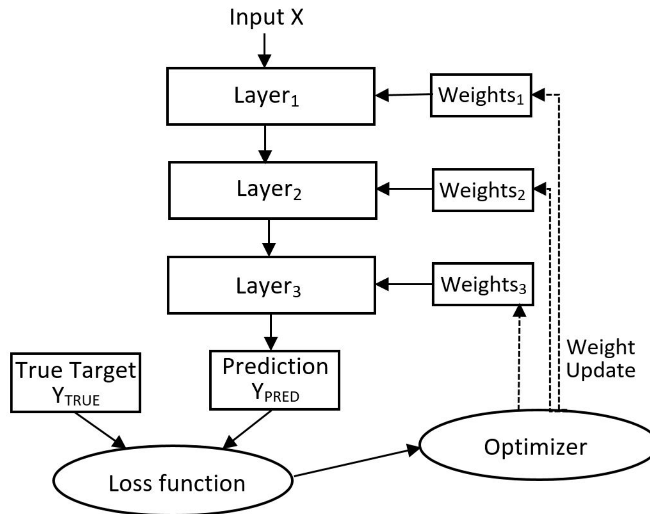


Figure 2.4: Neural Network Training

between the predictions and the actual targets through the loss function, then it goes through each layer in reverse to measure the contribution to the error of each weight of each layer (backward pass), and finally updates the weights to reduce the error. At the beginning of the training process, the weights are randomly initialized, then they are iteratively updated so that after a number of gradient descent steps, they converge toward an optimal solution.

Overfitting and Underfitting

The training of a model is an optimization problem: the parameters are updated so that the model better fits the training set. However, performing well on the training set does not imply that the model will generalize well on new unseen data (test set). Hence, a model training is not only an optimization problem but also a generalization problem: make the training error small (optimization) and make the difference between the training error and the test error small (generalization). To avoid information leak, the test set is only used at the very end of a project to assess the performance of the final model. The generalization error is rather monitored during the training phase using the validation set. The validation set plays the role of new unseen data. The ability of a model to learn complex patterns is called its *capacity*. The capacity of a model can be increased by adding more neurons or layers, or even training the model for a longer period of time (giving it more time to learn). Figure 2.5 illustrates the relationship between a model's capacity and the training and generalization errors. As the capacity of a model increases, both training and validation (also called the generalization error) error decrease, since the model is learning patterns that are common to both sets. After a certain capacity is reached, the validation error starts increasing while the training error keeps decreasing. The model is starting to learn patterns that are specific to the

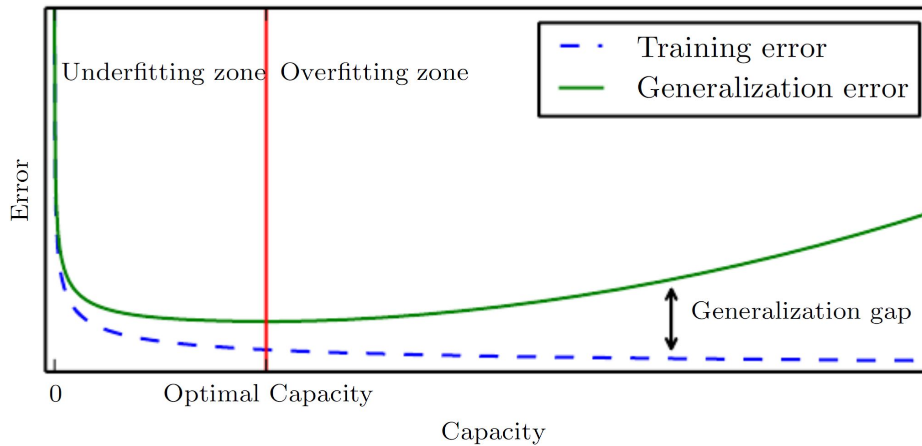


Figure 2.5: I. Goodfellow et al. [GBC16]: Typical relationship between a model's capacity and the training and generalization errors

training set, such as random noises, but are irrelevant when it comes to new data. The model is said to *overfit* the training set: it performs well on the training set but generalizes poorly on new data. One solution to reduce *overfitting* is to get more training examples. Another solution is to perform model *regularization*. Regularization consists of adding constraints to the model so as to reduce its capacity to learn complex function. When it comes to neural networks, common regularization techniques include early stopping [GBC16; Gér19](stop model training after a fixed number of epochs), dropout [Sri+14; Hin+12] or weight regularization. The opposite of overfitting is called *underfitting*. A model is said to *underfit* the data if it performs poorly on both the training set and the validation set. Both the training error and the generalization error are high. The model is not able to capture any relevant pattern present in the training set. Solutions to avoid underfitting include increasing the capacity of the model (increase the number of layers or neurons), remove constraints from the model, or add new features.

2.1.3 Metrics for Model Assessment

In this subsection, we present the most common metrics used to measure the performance of a machine learning model.

Regression Task

For a regression task the model attempts to predict a numerical value as close as possible to the true target value. The most common metric used to assess the performance of a regression task is the Mean Squared Error (MSE). Let m be the total number of samples in the test set used to compute the MSE. Let $\hat{y}_i$ and y_i be respectively the prediction and the target value for the i_{th} instance in the test set. The MSE is given by:

$$MSE = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

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

Figure 2.6: Confusion matrix

For classification tasks, we will first introduce metrics for binary classification then show how they can be generalized for multiclass classification.

Binary Classification/ Anomaly Detection

A binary classification problem is composed of a positive class and a negative class. In the specific case of a detection problem, the positive class corresponds to what we want to detect. For example in an attack detection problem, the positive class contains instances that correspond to attacks while the negative class contains instances that can be labeled as 'no attack' or legitimate. Note that all the metrics used for binary classification problem can also be used to assess anomaly detection model. Indeed, performance evaluation is performed on the test set, and for an anomaly detection problem, the test set usually contains both positive and negative class instances. The main difference between a binary classification model and an anomaly detection model is the data used during the training phase. A binary classifier is trained using both negative and positive instances while an anomaly detector is trained on negative instances only.

The performance on the test set of a binary classifier can be summarized using a confusion matrix as shown in Figure 2.6. A confusion matrix shows the number of true positives (TP), the number of true negatives (TN), the number of false positives (FP) and the number of false negatives (FN).

The number of TP, TN, FP and FN are used to compute different performance metrics such as the accuracy, the precision, the recall and the False Positive Rate (FPR).

The accuracy is the ratio of correctly predicted classes and is given by:

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

The precision is the proportion of instances predicted as positive that are actually positive and is given by:

$$precision = \frac{TP}{TP+FP}$$

The recall, also called the True Positive Rate (TPR) or attack detection rate (in intrusion detection problems), is the proportion of actual positive instances that are correctly predicted as positive and is given by:

$$recall = \frac{TP}{TP+FN}$$

The FPR is the proportion of actual negative instances that are incorrectly predicted as positive and is given by:

$$FPR = \frac{FP}{FP+TN}$$

To compare two models, it is often convenient to combine the precision and the recall into a single metric called the F_1 score. It is given by:

$$F_1 = \frac{2 \times precision \times recall}{precision + recall}$$

Another method to assess the performance of a binary classifier or an anomaly detector is to plot a Receiver Operating Characteristic (ROC) curve and compute the Area Under the Curve (AUC). To predict the class of an instance some model computes a score. If the score is greater than a certain threshold, the instance is assigned to the positive class. Otherwise, it is assigned to the negative class. The ROC curve plots the TPR against the FPR for various detection threshold values as shown in Figure 2.7. The AUC represents a measure of the separability between negative and positive classes. It can be interpreted as follows: consider the situation in which all the instances are already correctly classified, then we randomly pick one instance from the negative class and one instance from the positive class; we use the model to predict the score for both instances and decide that the one with the highest score belongs to the positive class. The AUC is the percentage of randomly drawn pairs for which we will be right. The closer the AUC is to 1 the better the model separates the two classes. An AUC close to 0.5 indicates that the model is performing no better than random guessing.

Similar to ROC curve is the Precision-Recall (PR) curve which plots precision versus recall. The area under the PR curve is referred to as the average precision (AP).

Multiclass Classification

For multi-class classification problems, macro-averaging, weighted-averaging or micro-averaging can be performed to assess the performance of the classifier [Shm20].

In the macro-average method, the performance is obtained by averaging over the individual performances obtained for each class. Let C be the total number of classes, and $precision_i$ be the precision obtained for class i. Then, the macro-average of precision is given by:

$$macro_avg_precision = \frac{1}{C} \sum_{i=1}^{C} precision_i$$

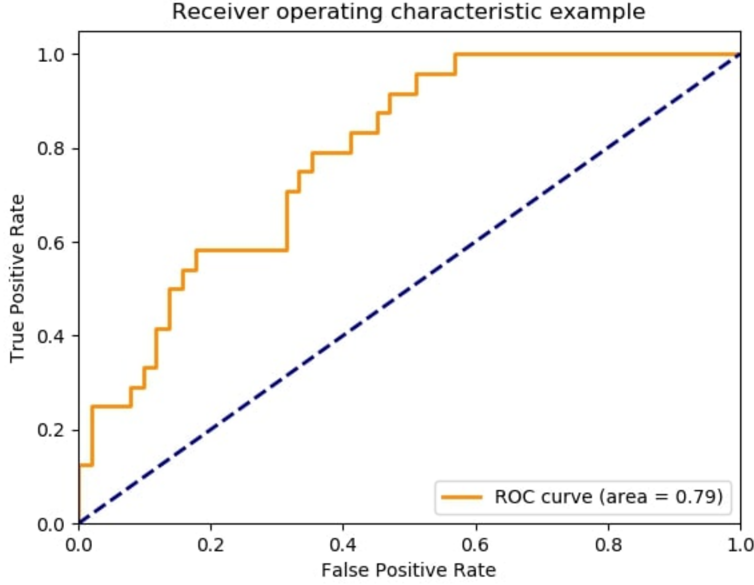


Figure 2.7: ROC curve example [Sci20]

In the weighted-average method, class imbalance is taken into account and the contribution of each class is weighted with the number of instances in that class. Let C be the total number of classes. Let N be the total number of instances and N_i be the number of instances belonging to class i. The weighted-average of precision is given by:

$$weighted_avg_precision = \frac{1}{N}\sum_{i=1}^{C} N_i \times precision_i$$

In the micro-average method, the number of TP, FP, FN and TN obtained for each class are summed up to obtain the overall number of TP, FP, FN and TN. Let C be the total number of classes. Let TP_i, TN_i, FP_i and FN_i be the number of TP, TN, FP, and FN respectively for class i. And Let TP_o, TN_o, FP_o and FN_o be the model's overall number of TP, TN, FP, and FN respectively, computed as follows:

$$TP_o = \sum_{i=1}^{C} TP_i$$
$$TN_o = \sum_{i=1}^{C} TN_i$$
$$FP_o = \sum_{i=1}^{C} FP_i$$
$$FN_o = \sum_{i=1}^{C} FN_i$$

The micro-average of precision is then given by:

$$micro_avg_precision = \frac{TP_o}{TP_o+FP_o} = \frac{\sum_{i=1}^{C} TP_i}{\sum_{i=1}^{C} TP_i+FP_i}$$

2.1.4 Neural Network Architectures

Multiple ANN architectures have been proposed in the literature. We propose to briefly describe some of them, especially the ones that will be extensively used in this thesis:

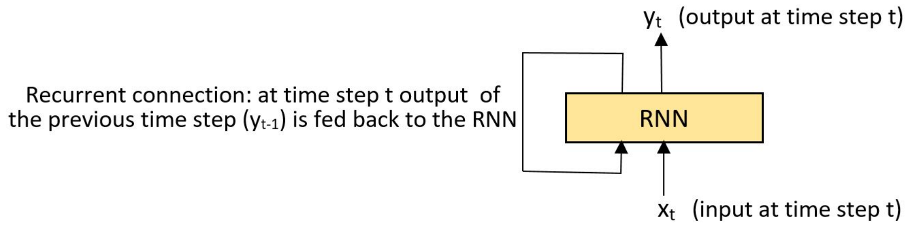


Figure 2.8: Recurrent Neural Network

- *Feedforward Neural Networks (FNN)* are a category of neural networks in which the information flows only in the forward direction from the input, though the hidden layers, to the output. There are no feedback connections in which the output of a layer is fed back into itself. *Fully connected Neural Networks (FCNN)*, also called *Multilayers Perceptron (MLP)*, are a type of FNN composed exclusively of densely connected layers as illustrated in Figure 2.2. A densely connected layer (also called fully-connected layer) is a layer in which each unit is connected to all the units of the previous layer.

- *Convolutional Neural Networks (CNN)* [GBC16; LeC+98] are a type of FNN specialized in the processing of grid-like data such as an image (2D grid) or a time series (1D grid). A CNN uses convolution in place of general matrix multiplication in at least one of its layers. A convolution layer is based on a shared-weights architecture and translation-invariance characteristics, making it a good choice to process image-like data. CNN are extensively used in Computer Vison, whether it is for image classification, object detection or image segmentation.

- *Recurrent Neural Networks (RNN)* [GBC16; Fra17] are specialized in the processing of sequential data such as time series or texts. As illustrated in Figure 2.8, at each time step, a recurrent layer receives the input that corresponds to that time step as well as its own output from the previous time step. As a consequence, the output at a given time step is a function of all the inputs from the previous time steps. Hence, the network is said to have a memory. Widely used variants of RNN that provide better long-term memory include Long Short-Term Memory (LSTMs) [HS97] and Gated Recurrent Units (GRU) [Cho+14] networks.

- *Autoencoders* [Gér19; GBC16] are unsupervised neural networks that learn to copy their inputs to their outputs under some constraints. For example, in vanilla autoencoders, the constraint is the limited size of the hidden layer that contains less neurons than the total number of input neurons. An autoencoder is also said to be composed of an *encoder* and a *decoder*. As shown in Figure 2.9, the encoder compresses the input to obtain a latent representation of it. The role of the decoder is to reconstruct the original input from its compressed latent representation. The difference between the input and

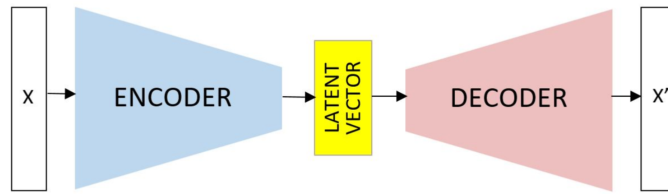


Figure 2.9: Autoencoder

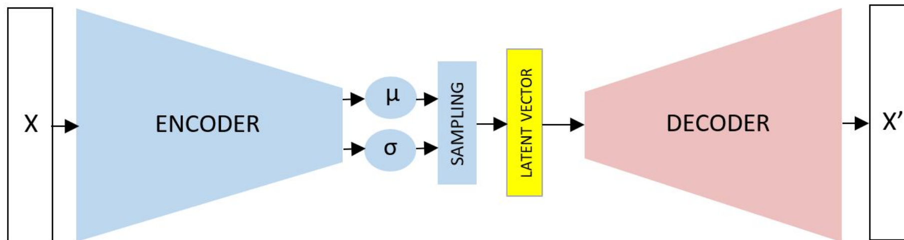


Figure 2.10: Variational Autoencoder

the output is called the *reconstruction error*. An autoencoder is trained to minimize the reconstruction error on a given dataset. Autoencoders can be used for anomaly detection. Indeed, once trained an autoencoder is good at compressing and reconstructing instances that are similar to the data used during training. However it is very bad at reconstructing new instances that were not seen during the training phase.

- *Variational Autoencoders (VAE)* [KW13; Gér19] are special types of autoencoders composed of an encoder that maps the input to a multivariate normal distribution around a point in the latent space. Actually, as illustrated in Figure 2.10, the encoder of a VAE maps each input instance to a mean vector μ and a variance vector σ^2 in the latent space. The mean and variance vectors are then used as the parameters of the multivariate normal distribution from which the latent vector is sampled. This introduces stochasticity in the process, as a same input might result in different latent vectors. A VAE can be used as a generative model. In fact, the loss function of a VAE contains an additional term which is the Kullback-Leibler (KL) divergence. The role of the KL loss is to force the mean and variance vectors generated by the encoder to be close to the parameters of a standard normal distribution (zero mean and unit variance). Hence, once trained using the original dataset, one only needs to sample a latent vector from the standard normal distribution and feed it to the decoder of the VAE to generate new instances.

- *Generative Adversarial Networks (GAN)* proposed by I. Goodfellow et al. [Goo+14] are composed of two competing neural networks, a discriminator and a generator (see Figure 2.11). The role of the generator is to generate observations as similar as possible to the instances present in the training set in order to fool

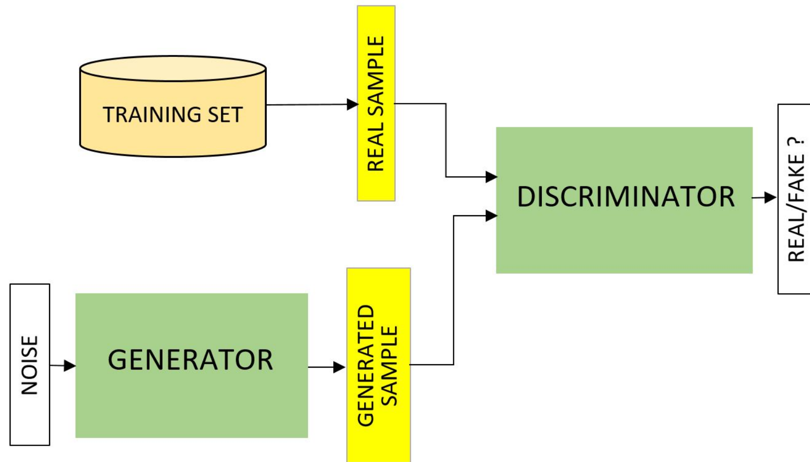


Figure 2.11: Generative Adversarial Networks

the discriminator. The role of the discriminator is to predict whether a given instance comes from the training set or has been generated by the generator. The GAN is trained by alternatively training the generator and the discriminator. After a sufficient number of training iterations the generator is good enough to generate instances that the discriminator cannot discriminate from real instances anymore.

2.1.5 Deep Learning Applications in Cybersecurity

Machine learning and especially deep learning models have been applied in different sub-fields of cybersecurity such as malware classification [Ye+17], vulnerability discovery [GS17] or network intrusion detection [BG15].

Malware Classification

B. Kolosnjaji et al. [Kol+16] leverage deep learning for malware system call sequences classification. They perform dynamic malware analysis: the malware is executed in a controlled environment to extract the sequence of system calls it performs. As system calls are categorical data, they are represented using one-hot encoded vectors. Malware samples are labeled using services provided by VirusTotal [20h]. They train a neural network composed of a recurrent layer stacked on top of a convolution layer. They report a classification precision of 85.6% and a recall of 89.4%. R. Pascanu et al. [Pas+15] propose to detect malware using a special type of RNN called Echo State Network (ESN). They also use sequences of system calls as features. The RNN is trained in an unsupervised fashion. It is followed by a FNN that is trained to classify malicious and benign sequences of system calls. They report a TPR of 98.3% and a FPR of 0.1%. J. Saxe et al. [SB15] propose to use deep neural networks to detect malicious binaries in Windows systems. To this purpose, they extract features from static benign and malicious binaries. Features include PE (Portable Executable) file's metadata and the list of DLL (Dynamic Link

Library) imports. They train a FFN to distinguish malicious binaries from benign ones. They reported a TPR of 95% and an FPR of 0.1%.

Vulnerability Discovery

Z. Li et al. [Li+18] describe a framework for vulnerability detection in C/C++ programs at source code level using deep learning. Vulnerable pieces of codes along with their patched version are obtained from different vulnerability databases such as the National Vulnerability Database (NVD). The code is tokenized before being fed to a bidirectional LSTM classifier. F. Yamaguchi et al. [YLR11] present a vulnerability extrapolation method using dominant API usage patterns. First, the source code is parsed into individual functions. Then API symbols for each function are extracted. Each function is embedded in a vector space using the extracted API symbols. Principal Component Analysis is then applied to infer the descriptive directions in the vector space. Finally, each function is expressed using dominant API usage patterns. Knowing the API usage pattern of a vulnerable function help us to identify other functions that share a similar API usage and possibly contain similar vulnerability.

Network Intrusion Detection

In the field of network intrusion detection, the KDD99 and the NSL-KDD datasets have been widely used to train and test machine learning based NIDS [BG15; 20e]. Those datasets describe TCP/IP communications using 41 features, including basic features such as the duration, the total number of bytes received and sent, and more advanced content-level features such as the number of incorrect logins or the number of operations performed as root. They contain two basic classes: normal and attack network traffic. The network attacks are further divided into four different categories: DoS, R2L (Remote to Local), U2R (User to Root) and probes. A. Javaid et al. [Jav+16] train and test a deep learning based NIDS on the NSL-KDD dataset. They propose a neural network architecture composed of two stages: an unsupervised pretrained component followed by a FCNN that plays the role of a classifier. Unsupervised pretraining of the lower layers is used to learn useful representation of the input data. They report an F_1 score of 98.84% for the binary classification problem (normal or attack) and an F_1 score of 75.76% for the 5-class classification problem (normal or one of the four attack types). Other notable works include RNN based NIDS by C. Yin et al. [Yin+17] and the NIDS proposed by N. Shone et al. [Sho+18] that consists of stacked autoencoders followed by a Random Forest classifier.

More recently, with the rapid development of IoT networks, works taking advantage of data analysis techniques for IoT network traffic monitoring are emerging. Those works specifically focus on IoT network traffic classification and IoT network intrusion detection and will be thoroughly described in the following sections.

2.2 IoT Network Traffic Classification

In this section, we present works that focus on IoT network traffic classification. Existing works on IoT network traffic classification can be divided into two categories depending on their purpose: IoT device type identification and IoT device state determination. In IoT device type identification (or fingerprinting), network traffic classification is performed to identify the IoT device type that is generating a given network traffic. In IoT device state determination, the purpose of traffic classification is to determine what action a device is currently performing (and hence guess what the user is doing) just by analyzing its network traffic.

2.2.1 IoT Device Type Identification

Y. Meidan et al. [Mei+17a; Mei+17b] propose to identify IoT device type to prevent unauthorized devices from connecting to the network (see Figure 2.12). To this purpose, a white list of trustworthy devices is created. A Random Forest algorithm is trained to learn to identify IoT device types from the white list based on network traffic data. The dataset consists of full TCP sessions from SYN to FIN, identified by source and destination IP addresses and ports. The TCP sessions are described using 274 features extracted using a tool presented in [Bek+15], along with, 60 new features designed by the authors. The authors do not provide lot of information about the used features. Only the top features, the ones found to be the most influential in identifying device type, are described. Most of them are statistics (minimum, average, quartiles, etc.) on the TTL (time-to-live) value of the packets in the TCP session. For experimental evaluation, 9 types of IoT devices are used: baby-monitor, motion-sensor, refrigerator, security camera, smoke detector, socket, thermostat, TV, watch. For empirical evaluation of the method, 9 experiments were performed. In each experiment, one IoT device type is left out of the white list, to represent an unauthorized device. For each experiment a multiclass Random Forest classifier is trained using TCP sessions data from the 8 devices that compose the white list. When fed with the feature vector describing a TCP session, the classifier outputs a probability distribution over all possible device classes. A threshold tr is used, so that if the probability is higher than tr for a given class the session is classified as belonging to that device class. Otherwise, if the probability is less than tr for every device class, the session is classified as "unknown". Performance evaluation shows that, on average, 94% of unknown IoT device types were detected and 97% of the white-listed devices were correctly classified by their actual types. The authors propose to improve the performance by performing majority voting over several consecutive TCP sessions. Based on 20 consecutive TCP sessions and majority voting rule, on average, 96% of the unknown IoT devices were detected and 99% of the white-listed devices were correctly classified.

M. Miettinen et al. [Mie+17] present IoT SENTINEL, a system capable of fingerprinting the type of an IoT device thanks to the network traffic it generates

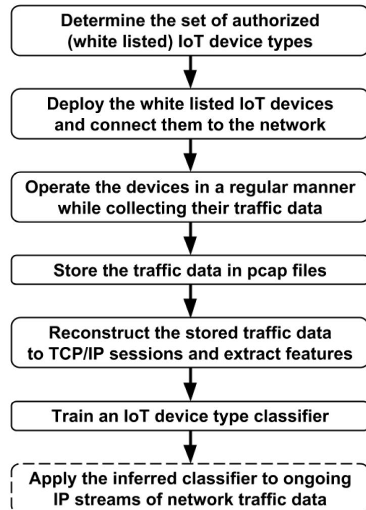


Figure 2.12: Y. Meidan et al. [Mei+17a]: Overview of proposed method for IoT white listing

during the setup phase. The aim is to constrain the communications of vulnerable devices. Once a new device, identified by a new MAC address, is being connected to the network, the network packets sent during the setup phase are collected. The end of the setup phase is identified by a decrease in the number of packets sent. Each network packet is described using a total of 23 features obtained after one-hot encoding of the categorical variables. Examples of categorical features are the link layer protocol, the network layer protocol, the transport layer protocol or the application layer protocol. Other features are integers, such as the number of different destination IP addresses or the packet size. The final fingerprint is a feature vector of size 23 $\times$ number of packets sent during the setup phase. One Random Forest classifier per device is trained. If a device fingerprint matches several device types, it is compared to a subset of fingerprints from each device type it got a match for. The dissimilarity score is computed using the Damerau-Levenshtein edit distance. The lowest dissimilarity score gives the final predicted device type. The authors report an overall accuracy of 81.5% for their method.

B. Bezawada et al. [Bez+18] describe IoTSense, a system to fingerprint IoT device type from the analysis of a sequence of packets. A network packet is described using a total of 20 features, after one-hot encoding of categorical variables. They reuse 17 of the binary features used in IoTSENTINEL [Mie+17], such as the different protocols used from network to application layers. The 3 other features are the Shannon entropy of the payload, the TCP payload length and the TCP window size. A fingerprint is represented using 5 session packets, which leads to a final feature vector of size $5 \times 20 = 100$. They tested 4 different classifier, namely, kNN, Decision trees, Gradient boosting and Majority voting. The Gradient boosting classifier achieved the best results with a mean accuracy of 99%.

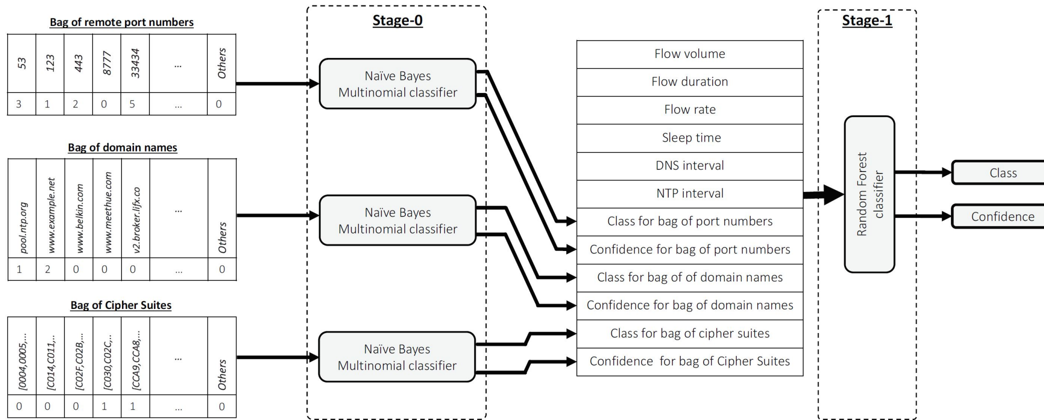


Figure 2.13: A. Sivanathan et al. [Siv+18]: Architecture of the multi-stage classifier

A. Sivanathan et al. [Siv+18] present a method to perform IoT network traffic classification for device type identification using a multi-stage machine learning classifier. The network traffic is described using 9 different statistical attributes. This includes 3 nominal attributes, namely the set of domain names requested by the device, the set of remote port numbers the device communicates with and the set of TLS/SSL cipher suites supported by the device. The remaining attributes are numerical and include the total number of bytes exchanged, the mean flow duration, the average flow rate, the device sleep time (time interval over which the device has no active flow), the DNS interval (average time interval between two consecutive DNS request) and the NTP interval. Those statistical attributes are derived from one hour of active network traffic data. That is, if a device does not spend much time online, the duration one have to wait to collect one hour of active network traffic data might last longer. As described in Figure 2.13, the first stage of the machine learning process is to convert the 3 nominal attributes into numerical attributes. To this purpose, each of the 3 nominal attributes are fed to a Naïve Bayes classifier that outputs a tentative class and a confidence interval for each of the attributes. The 6 newly obtained numerical attributes are then combined with the other numerical attributes leading to a final feature vector of size 12. The final stage consists of a Random Forest classifier. The authors report an accuracy of 99% in identifying the type of IoT device.

S. Marchal et al. [Mar+19] propose AuDI, an autonomous distributed system trained to learn to identify the type of IoT devices connected to the network based on the temporal periodicity of IoT network communications. It can be used as a first step before performing anomaly detection. First, AuDI is used to identify the type of device connected to the network. Then the appropriate legitimate behavior profile (the one that corresponds to the identified device) can be used to monitor the device. The network traffic is described using flows. A flow is defined as a sequence of network packets originating from the same MAC address and having the same communication protocol (e.g., NTP, ARP, RTSP, etc). As shown in Figure 2.14,

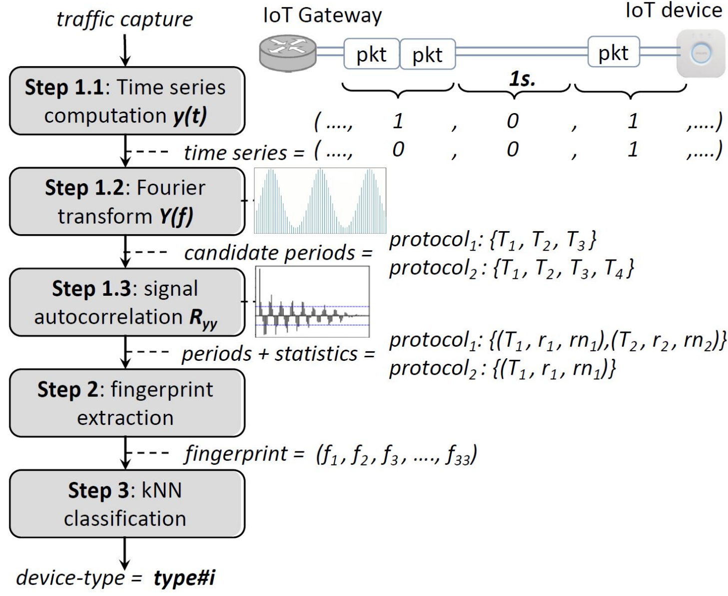


Figure 2.14: S. Marchal et al. [Mar+19]: Overview of device type identification

the flow is then discretized into a binary time series sampled at one value per second, indicating whether the flow contained one or more packets during the 1-second period (value 1) or not (value 0). Discrete Fourier Transform and discrete autocorrelation are computed to derive the features. A total of 33 features are obtained. Those features mainly describe periodic aspect of the network communications. For experimental purposes, a 30-minute window is used to capture the flows. That is, 30 minutes is required for identifying a device type. A kNN classifier is trained to identify the devices. The authors reported an overall accuracy of 98.2%.

L. Bai et al. [Bai+18] propose to combine an LSTM network with a CNN to better classify IoT network traffic. They do so by converting streams of network packets into time series data: streams of packets generated by a device (devices are uniquely identified by their MAC addresses) are first sub-divided into segments of fixed time interval. That way, each obtained segment contains a subset of the initial stream of packets. The obtained sequences of segments correspond to a time-series. Each segment is described by features such as the total number of packets it contains, the protocol of the packets and different statistics on the size of the packets (mean, standard deviation, minimum, maximum, etc.). The LSTM-CNN model achieves an accuracy of 74.8% in identifying IoT device type.

V. Thangavelu et al. [Tha+18] present DEFT, a distributed IoT fingerprinting solution. It is composed of a controller and multiple gateways. The controller is located in the ISP network and its role is to train and maintain classifiers. The gateways are located closer to the IoT networks, inside homes or enterprises, and their role is to classify the network traffic. Network traffic data is described using traffic sessions. A traffic session is defined as an aggregation of connections from

and to a particular device for a fixed interval of time. A traffic session is described through 111 features, including features, such as the number of DNS queries, statistics on the size of the packets and the duration of the connections. For experimental evaluation, a time interval of 15 minutes is used to define the traffic sessions. Four machine learning algorithms are tested: kNN, Gaussian and Bernoulli Naïve Bayes, and Random Forest. Random Forest performs the best with an accuracy of 98%.

F. Le et al. [Le+19] propose to identify IoT device type by analyzing DNS requests. The DNS names that are queried by each device are collected during a specified time interval. Then the authors propose to model the set of DNS names as a set of words representing a document. As a first step, the DNS names help to determine the vendor of a device. If a device queries DNS names from multiple vendors, majority voting is used to determine the vendor that manufactured the device. Next, the type of device is determined by applying TF-IDF (term frequency–inverse document frequency) [JM09] method to the document representation of the DNS names. TF-IDF is a natural language processing method that can be used to find the most similar document in a corpus to an input document. The document representation of the DNS names of the training set are sorted by vendor and device type in a database. Once a new device is connected to the network, the DNS names it queries are collected and compared to the training set database to determine its vendor and type. The experimental testbed consists of 91 IoT devices. A 24h time interval is used to collect the DNS names queried by each device. The method achieves 99% accuracy. When the time interval is reduced, the accuracy decreases. When collecting DNS names for only 1 hour, the method is no longer able to differentiate IoT devices by type.

N. Ammar et al. [ANT19] propose to identify the type of an IoT device newly connected to the network by collecting data exchanged during its setup phase. The features used are textual data such as the manufacturer name derived from the MAC address, device name obtained from DHCP information, model information and manufacturer friendly name extracted from UPnP messages, device local name and service names obtained from mDNS records, device OS and model extracted from HTTP headers. The textual features are used to create a Bag of Words (BoW) vector representation. The BoW representations of each device is stored in a database. To determine the type of a new device, its BoW representation is extracted and is compared to all the inputs in the database to find out the most similar one.

2.2.2 IoT Device State Determination

A. Acar et al. [Aca+18] propose to leverage machine learning to profile the wireless network traffic of a smart home in order to determine the states of smart devices and the actions they are performing. This rises serious privacy concerns as one can determine what is going on inside a smart home just by passively sniffing the

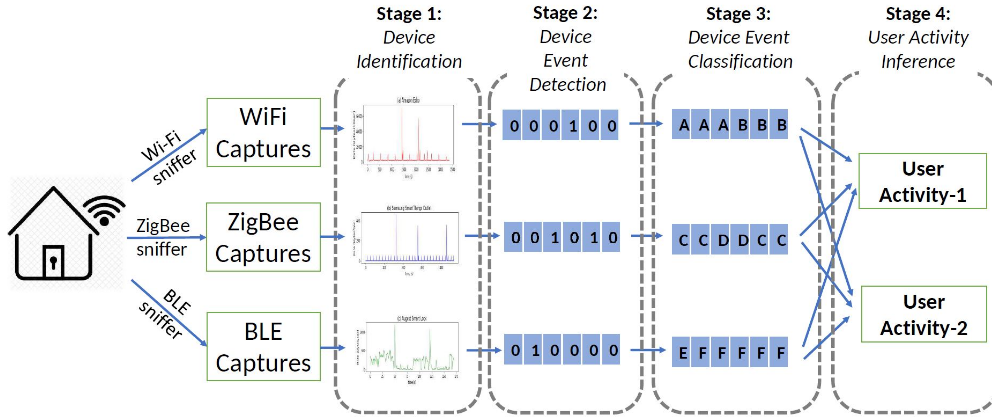


Figure 2.15: A. Acar et al. [Aca+18]: Overview of the multi-stage privacy attack

network even if the network traffic is encrypted. The authors refer to their novel attack as multi-stage privacy attack, which is achieved by passively observing the wireless network traffic of the smart home devices. Figure 2.15 shows the 4 different stages of the attack. Three wireless protocols are targeted: WiFi, ZigBee, and Bluetooth Low Energy (BLE). In the first stage of the attack (stage 1), the attacker's goal is to determine the type of each smart home device associated with specific physical addresses (e.g. MAC address, NwkAddr). This is done by performing multi-class classification based on the traffic profile of each device. As works on device type identification already exist, like the ones presented earlier [Mei+17a; Mie+17; Bez+18], the authors propose to reuse them. At stage 2, the aim of the attacker is to detect for each individual device whether a state transition is occurring or not. To this purpose the network traffic is observed for a given time interval and a feature vector is extracted. The feature vector consists of three variables: mean packet length, mean inter-arrival time and median absolute deviation of packet size. For each device type, Random Forest and kNN classifiers are trained to predict its state transitions. Experiments show an average F_1 score of 91% for this stage. At stage 3, after detecting transitions between device states, the aim is to determine the specific state to which the device transitioned. To this purpose, the network trace of a device is split into segments corresponding to different device states (e.g., ON, OFF). Then multi-class classification is performed with classes representing possible devices states. For this latter classification, 191 time-series features are extracted using a tool called *tsfresh* [Chr+18]. The following classification algorithms are tested: XGBoost, Adaboost, Random Forest, SVM with RBF kernel, kNN, Logistic Regression, Naïve Bayes, and Decision Tree. An F_1 score of 94% is achieved for this stage. Finally, at stage 4, using the state information of the devices, the attacker can guess what activity the user inside the house is doing. This is done by using the states of the different devices and a Hidden Markov Model. Depending on the activity, the F_1 score varies between 91% and 100% for this stage.

N. Apthorpe et al. [ARF17] present a strategy for a passive network observer

to be able to infer consumers behavior from rates of IoT device traffic, even when the traffic is encrypted. First, the observer needs to separate packet streams for each device. DNS queries associated with each packet stream allow the observer to determine the type and vendor of the IoT device that generated the stream. Then, simply plotting send/receive rates of the streams (bytes per second) can reveal potentially private user interactions with the device. Four consumer devices where analyzed in the study: Sense sleep monitor, Nest security camera, WeMo switch, Amazon Echo (see Figure 2.16). The analysis of the network traffic rate of the Sense sleep monitor allows the observer to determine the time at which the user is sleeping. For the Nest security camera, it is possible to determine if the user is actively viewing the video feed through the web or mobile application. The traffic rate analysis make it also possible to determine when a WeMo switch is turned on or off. It also makes it possible to know the time of the day a user is interacting with an Amazon Echo assistant.

B. Copos et al. [Cop+16] propose to infer if anybody is home by analyzing network traffic data. They specifically attempt to determine when a device is transitioning from one mode to another. For example, the Nest thermostat offers 2 modes: Home and Auto-away mode. To this purpose, they identify all the unique connections defined by a destination IP address and a specific number of bytes sent. Then an N by N correlation matrix is defined where N is the total number of unique connections. The idea is to detect connections that are correlated and often occur simultaneously during mode transition. Experimental results show that, based only on network traffic originating from the device, it is possible to determine when the thermostat transitions from the Home to the Auto Away mode and vice versa with an accuracy of 88% and 67% respectively.

D. Caputo et al. [Cap+20] show that it is possible for an adversary to infer the state of a smart speaker, such a Google Home Mini, just by analyzing its generated network traffic. The authors consider 3 states of a Google Home Mini: the microphone is off (*mic_off*), the microphone is on but does not receive any stimuli or noise from its environment (*mic_on*), the microphone is on and is surrounded of humans speaking to each other (and not to the device) and background noise (*mic_on_noise*). For the experiment, the traffic is collected for 3 days for each of the states. The features are network traffic statistics collected over specific time windows. They include statistics such as the number of TCP, UDP and ICMP packets, the number of different IP addresses and ports, or the IAT between packets. Decision Tree and Adaptive Boosting (AdaBoost) algorithms are considered for training. For the first experiment, the classifiers are trained on network traffic data from 2 states: *mic_off* and *mic_on*. The aim is to determine whether it is possible to determine if the microphone of the device is on or off just by analyzing its network traffic data. Both classifiers achieve a reasonable accuracy of about 80% when the network traffic is collected over a time window of at least 500 seconds or when more than 500 packets are collected. For the second experiment, the classifiers are trained

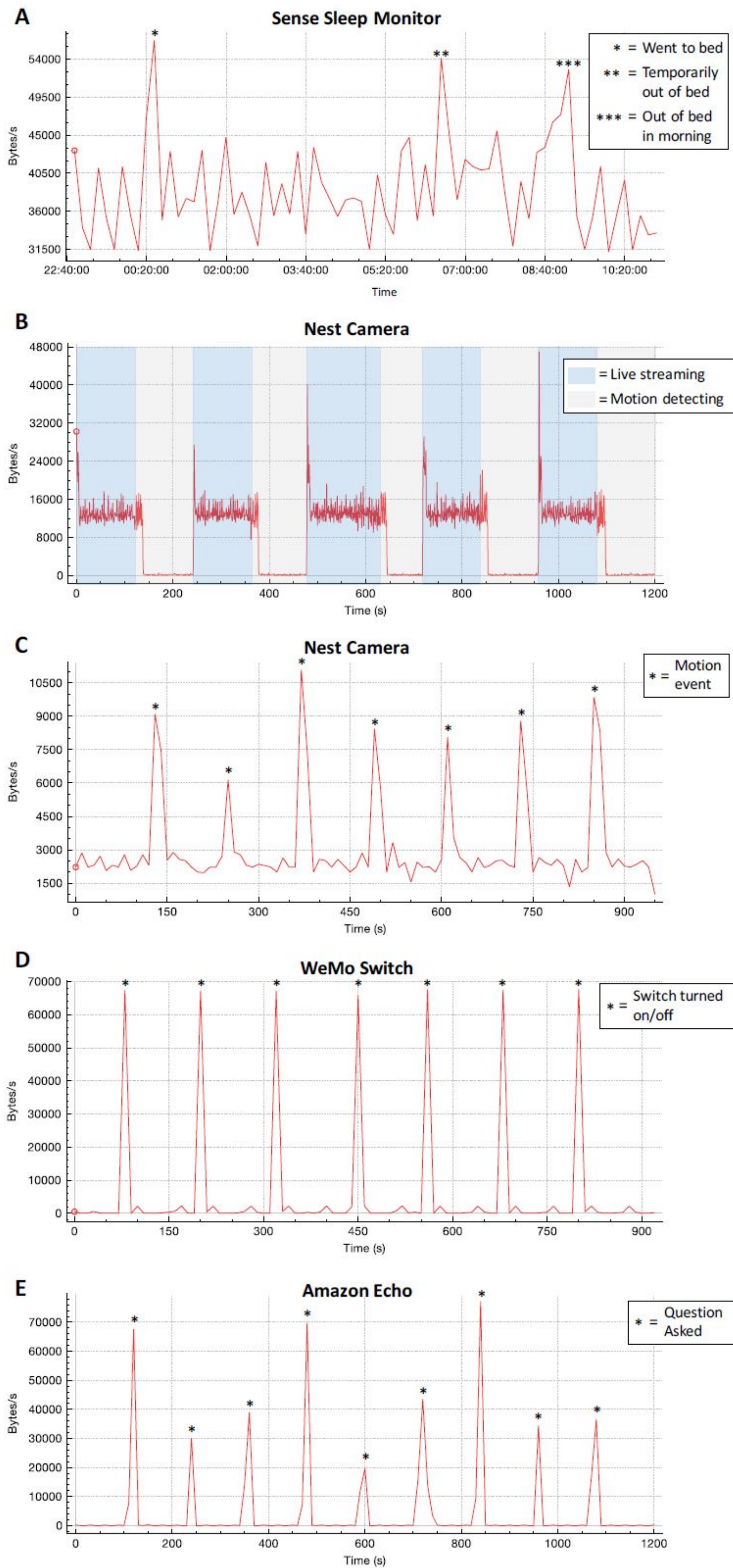


Figure 2.16: N. Apthorpe et al. [ARF17]: Network traffic send/receive rates of selected IP streams from 4 commercial IoT devices during controlled experiments

Table 2.2: Summary of works on IoT network traffic classification for device type or device state determination

Reference	Device type/state detection?	Algorithms used	Example of features used
Y. Meidan et al. [Mei+17a; Mei+17b]	type	Random Forest	statistics on TTL values
M. Miettinen et al. [Mie+17]	type	Random Forest	link, network, transport and application layers protocols, number of different destination IP addresses, packet size
B. Bezawada et al. [Bez+18]	type	k-NN, Decision trees, Gradient boosting and Majority voting	Shanon entropy of the payload, the TCP payload length and the TCP window size
A. Sivanathan et al. [Siv+18]	type	NaÏve Bayes, Random Forest	set of domain names, set of remote port numbers, set of supported TLS/SSL cipher suites, total number of bytes exchanged, mean flow duration, mean flow rate, sleep time, DNS and NTP intervals
S. Marchal et al. [Mar+19]	type	k-NN	discrete Fourier Transform and discrete autocorrelation derived from a binary time series representing the presence (1) or absence (0) of a packet
L. Bai et al. [Bai+18]	type	LSTM-CNN	total number of packets, protocol, statistics on the size of the packets
V. Thangavelu et al. [Tha+18]	type	k-NN, Gaussian and Bernoulli Naïve Bayes, and Random Forests	number of DNS queries, statistics on size of the packets and the duration of the flows
F. Le et al. [Le+19]	type	TF-IDF	DNS names
A. Acar et al. [Aca+18]	state	k-NN, Random Forest, XGBoost, AdaBoost, Hidden Markov Model	mean packet length, mean inter-arrival time, median absolute deviation of packet size
N. Apthorpe et al. [ARF17]	state	Visual statistical analysis	DNS queries, send/receive rates (bytes per second)
B. Copos et al. [Cop+16]	state	Correlation matrix	NA
D. Caputo et al. [Cap+20]	state	Decision Tree, AdaBoost	number of TCP, UDP and ICMP packets, number of different IP

on the network traffic data to distinguish between *mic_on* and *mic_on_noise*. The aim is to determine whether or not it is possible to determine if there is noise in the device's surroundings just by analyzing its network traffic. The classifiers achieve a reasonable accuracy of more than 80% when the network traffic is collected over a time window of at least 15 seconds or when more than 20 packets are collected.

2.2.3 Summary

Table 2.2 summarizes the different works on IoT device type identification and IoT device state determination through network traffic analysis. It presents the algorithms and features used in the different works.

As presented in Table 2.3, the existing works lack at least one of the following characteristics: be delay-free, phase-independent or non-intrusive. Works that do not require to wait for an unspecified amount of time or use a timeout of less than 15 minutes are said to be delay-free. Approaches that can be used at any moment of a device's life cycle (provided that the device is generating network traffic) are said to be phase-independent. Solutions that do not look at application data and

Table 2.3: Limitations of existing works on IoT network traffic classification for device type or state determination

Reference	Delay-free	Phase-independent	Non-intrusive
Y. Meidan et al. [Mei+17a; Mei+17b]	No	Yes	Yes
M. Miettinen et al. [Mie+17]	Yes	No	Yes
B. Bezawada et al. [Bez+18]	No	Yes	No
A. Sivanathan et al. [Siv+18]	No	Yes	No
S. Marchal et al. [Mar+19]	No	Yes	Yes
L. Bai et al. [Bai+18]	No	Yes	Yes
V. Thangavelu et al. [Tha+18]	No	Yes	Yes
F. Le et al. [Le+19]	No	No	No
A. Acar et al. [Aca+18]	NA	No	No
N. Apthorpe et al. [ARF17]	Yes	No	No
B. Copos et al. [Cop+16]	No	No	No
D. Caputo et al. [Cap+20]	Yes	No	No

only use network level features are said to be non-intrusive.

Some works on device type identification extract features from full communication sessions [Mei+17a; Mei+17b] or even over multiple sessions [Bez+18]. Moreover no timeout is used, that is, if a session lasts for hours, one has to wait until the end of the session to be able to determine the type of an IoT device. However, only the first few packets exchanged at the beginning of a session might already contain enough information to determine the type of a device. In other works on device type determination, data is collected for a fixed time interval like 30 minutes [Mar+19], 1 hour [Siv+18] or even 24 hours [Le+19]. However, it is not sure if it is really necessary to wait for such long fixed time intervals if the data collected within the first few seconds already provide enough information to determine the type of a device. When the first few packets already contain enough information to determine the type of an IoT device, waiting for the end of a session or a fixed time interval introduce unnecessary latency. Works determining the state of an IoT device try to minimize unnecessary latency as their purpose is to determine the state of a device at measurement time and not with hours of delay.

Some works on device type determination are also limited because they focus on a specific phase of a device life cycle or a specific packet type. Hence, a device type can be identified only under some conditions. For example, in [Mie+17], the type of an IoT device is determined at its setup phase. Their proposed method cannot be used if we have missed the setup phase of a device or if we want to determine the type of devices that are already connected to the network. The method presented in [Le+19] exclusively relies on DNS queries data, hence we can only use the approach after we have collected enough DNS queries data. Works on device state determination are also subjected to this type of limitation as only network traffics that occur during a state transition are of importance.

Works can also be separated based on their privacy-compliance level or intrusiveness. Obviously, works on device state determination are by essence not meant to be

privacy-compliant. Network-based device type determination methods can be categorized into two categories based on their intrusiveness: approaches that only use network level features such as packet sizes or the number of bytes, and approaches that also include application level features such as DNS queries or the entropy of the payload. Methods looking at application level data cannot be used if the network traffic is encrypted.

In Chapter 3 of this thesis, we propose an IoT device type recognition system that is delay-free, phase-independent and non-intrusive.

2.3 IoT Network Intrusion Detection

Machine learning based IoT NIDSs can be categorized as being supervised or unsupervised [BG15]. In supervised learning (also referred to as misuse detection), the algorithm is provided with a labeled dataset and the attack classe(s) is/are learned during the training phase, that is, data describing attacks are necessary. Conversely, in unsupervised learning, which includes the special case of anomaly detection, models are trained using only data describing legitimate behaviors. Therefore, there is no need for data labelling. During the training phase, an anomaly detection model learns the legitimate behavior profile. During the testing phase, the model is applied to new data to detect any deviation from the learnt legitimate behavior. Anomaly detection models allow the detection of previously unseen attacks.

2.3.1 Supervised NIDS

R. Doshi et al. [DAF18] propose to leverage classification algorithms for DDoS attack detection in consumer IoT networks. The detection pipeline is shown in Figure 2.17. First, the network traffic is captured and processed to records the source and destination IP addresses, source and destination ports, packet size and timestamps of each network packet. Source IP addresses are then used to separate packets by device. The stream of packets from each device is also divided into non-overlapping time windows. Stateless features, such as the packet size, IAT and the protocols used, as well as, stateful features, such as the total bandwidth and the total number of different destination IP addresses, are extracted. For the experiment, 3 IoT devices were used to generate legitimate traffic. DDoS traffic was generated by simulating TCP SYN flood, UDP flood and HTTP GET flood attacks. Five different supervised machine learning algorithms were tested: kNN, SVM with linear kernel, Decision Tree, Random Forest and FCNN. The authors report an F_1 score between 0.927 and 0.999 depending on the algorithms.

A. A. Diro et al. [DC18] propose a distributed deep learning based attack detection scheme for the IoT. The fog nodes (gateways, routers, hubs, etc.), located at the edge of the distributed network are responsible for training and storing the attack detection models. A coordinating master node ensures parameters sharing between the models trained at each fog node. They propose to use a FCNN composed of 3

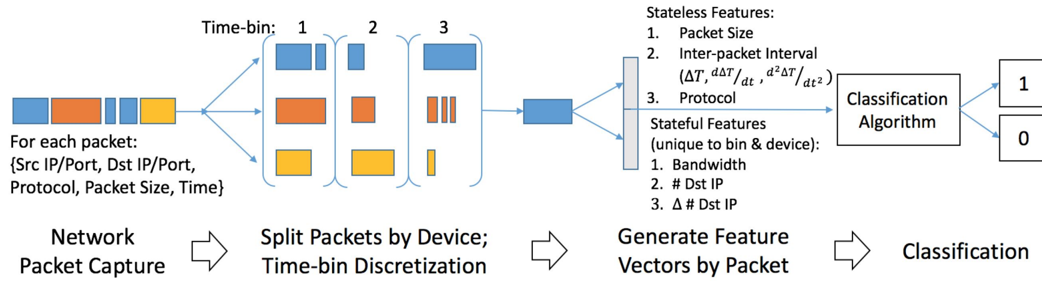


Figure 2.17: R. Doshi et al. [DAF18]: IoT DDoS detection pipeline

hidden layers (with a number of neurons per layer of 150, 120, 50 respectively) to classify the network traffic into either normal or attack. The performance of their method is not assessed on IoT network traffic but on the NSL-KDD dataset [BG15]. The model achieves an attack detection rate of 99% and a FPR of 0.85%.

K. Yang et al. [Yan+18] propose an active learning approach for wireless IoT network intrusion detection. Active learning is a sub-field of human-in-the-loop machine learning that allows to learn from a limited amount of labeled data. In active learning, the human expert does not label all the training instances which can be a tedious task if not impossible. Instead, the human expert labels a training instance upon the learning algorithm's request. The most common strategy is called *uncertainty sampling*. First, the model is trained on the labeled instances gathered so far and is used to make prediction on the unlabeled data. The instances for which the model is the most uncertain (for binary classification that corresponds to instances for which the model outputs a class probability close to 0.5) are given to the expert to be labeled. The process is iterated until it reaches a threshold of performance in terms of precision and recall. During the experiments, the authors choose to use the XGBoost [CG16] classification algorithm. The performance of the model is tested on two datasets: the NSL-KDD dataset [BG15] and the AWID dataset [Kol+15] (a dataset obtained from a real WiFi environment). Experiments show that the active learning method (based on uncertainty sampling) reaches the required performance much quicker than a random-select method (when the instances to be labeled by the human expert are selected randomly). Also, the total number of labeled instances required to achieve the same performance is reduced to almost one third.

E. Hodo et al. [Hod+16] present a preliminary work on the use of FCNNs for DDoS detection in an IoT network. They use an FCNN that consists of only one hidden layer. For experiments, an IoT network with 5 sensors is set up. Four of these sensors act as clients and one node acts as a server relay node. The server node receives information from the client nodes, perform data analytics and respond back to the client nodes. This allows the sensor nodes to collectively adapt their behavior and react to occurring events. The DDoS attack traffic was generated using a single host sending UDP packets to the server node. The authors do not describe the

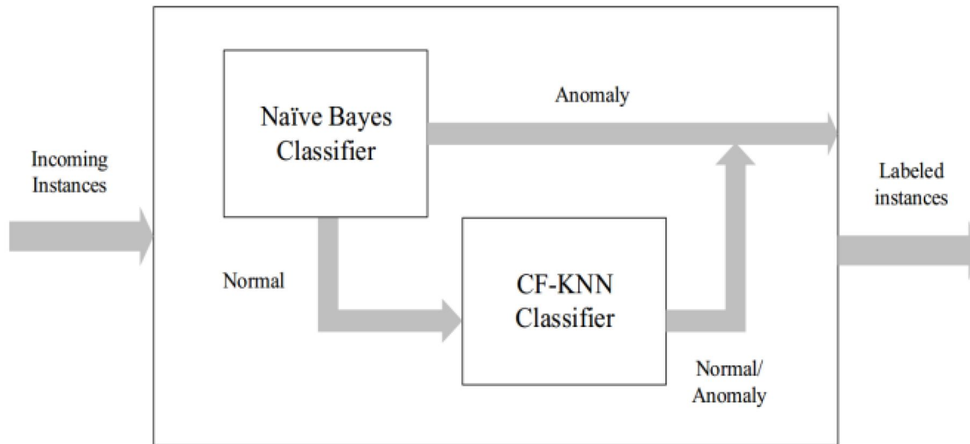


Figure 2.18: H. H. Pajouh et al. [Paj+16]: Two-tier classification module

features used to model the network traffic data. They claim an overall classification accuracy of 99.4% with a 1.9% FPR.

H. H. Pajouh et al. [Paj+16] propose a supervised intrusion detection method for IoT networks based on a two-layer dimension reduction module followed by a two-tier classification module. First, PCA is performed as a first dimensionality reduction step. This step maps the initial feature space to a lower dimensional space while preserving as much variance as possible. Next, LDA (Linear Discriminant Analysis), a supervised dimensionality reduction technique, is used to further reduce the dimensionality of the PCA-transformed data. The two-layer dimensionality reduction module is followed by a two-tier classification module described in Figure 2.18. The first layer of the classification module is a Naïve Bayes classifier that determine if a given instance is an attack or normal. If an instance is classified as normal by the Naïve Bayes classifier, it is fed to a second classifier for a second verification. The second classifier is a Certainty-Factor version of the k-NN classifier. Its role is to further refine the results of the first classifier and determine if a given instance is an attack or not. For experiments, the NSL-KDD dataset is used. The proposed model achieve an attack detection rate of 84.86% and a FPR of 4.86%.

N. Moustafa et al. [MTC18] propose an ensemble intrusion detection technique to detect botnet attacks against DNS, HTTP and MQTT protocols utilized in IoT networks. Connection records are collected and stored in a database. Then, features that describe statistics are computed over sequences of 100 recorded connections. The features used are categorized into four categories: flow features, MQTT features, DNS features, and HTTP features. Flow features include source-destination IP addresses and ports, as well as, the protocol type and the last time of connection. MQTT features consist of length of MQTT data, number of connections from the same source and number of connections to the same destination. DNS features are attributes, such as, the domain name, query type, length of the query and length of the answer. The HTTP features consist of the HTTP request method (e.g. GET,

POST, HEAD), the URI request, the size of the content, and the URL length. The model is trained as follows: first, correlation coefficient is applied to select important features; then, an AdaBoost ensemble learning method is used for classification. For the experiment, the UNSW-NB15 [MS15] and NIMS botnet [20d] datasets along with simulated IoT sensors data are used. The IoT sensors network is simulated with Raspberry Pis. Depending on the data type, the attack detection rate range from 97% to 99%, with a FPR between 1% and 3%.

P. Shukla et al. [Shu17] propose a hybrid machine learning method that combines unsupervised and supervised learning to detect wormhole attacks in RPL [Win+12] based 6LoWPAN IoT networks. In a wormhole attack, a tunnel is created between two compromised routers and is used to modify the routing behavior of the network. First, K-means clustering is used to determine the safe zone of each router in the network. Safes zones are then used to measure how far a router can communicate. A decision tree classifier is trained to further refine the results obtained from clustering. The model is evaluated for different network topologies: random (nodes located at random coordinates), mesh, ring and star. The model achieves a 71%-75% attack detection rate. The FPR is not reported.

2.3.2 Unsupervised NIDS/ Anomaly Detection

Y. Meidan et al. [Mei+18] describe N-BaIoT, a network based anomaly detection method for the IoT. They propose to use stacked autoencoders (autoencoders with multiple hidden layers) to detect IoT botnets. Whenever a packet arrives, a behavioral snapshot of the hosts and protocols which communicated the packet is extracted. It consists of 115 statistics computed over several time windows. The statistics summarize all of the traffic that has (1) originated from the same IP address, (2) originated from both the same source MAC and the same IP address, (3) been sent between the source and destination IPs, and (4) been sent between the source and destination TCP/UDP sockets. The statistics are essentially mean and variance of the packet sizes and the IAT. The time windows are the most recent 100 ms, 500 ms, 1.5 s, 10 s, and 1 mn. Note that those statistics can be computed only if the anomaly detector is deployed in the local network. One autoencoder is trained per device. That is, during the testing phase, prior knowledge about what device has generated the traffic is required. For the experimental evaluation, 9 IoT devices are used. The malicious traffic is generated by executing Mirai and BASHLITE malware in an isolated lab environment. The autoencoders consist of a 4-layer encoder and 4-layer decoders. The authors claim an attack detection rate of 100% and a FPR of 0.0070 ± 0.01.

Y. Mirsky et al. [Mir+18] present Kitsune: a plug and play NIDS which can learn to detect attacks on the local network in an unsupervised way and in an efficient online manner. Kitsune's core algorithm uses an ensemble of autoencoders to collectively differentiate between normal and abnormal traffic patterns. Kitsune

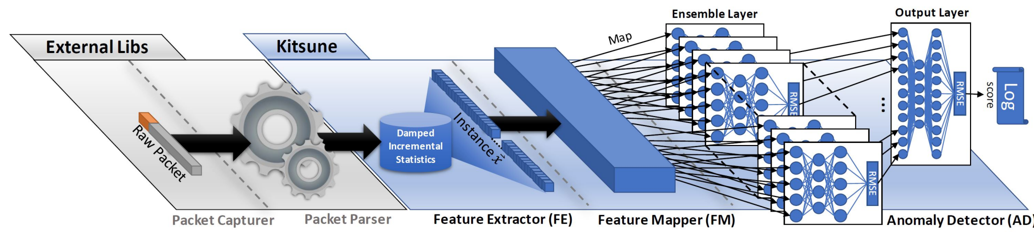


Figure 2.19: Y. Mirsky et al. [Mir+18]: An illustration of Kitsune's Architecture

is composed of 5 components: the Packet Capturer, the Packet Parser, the Feature Extractor, the Feature Mapper and the Anomaly Detector (see Figure 2.19). The Packet Capturer acquires each new packet and passes it to the Packet Parser. The Packet Parser parses the packet and extracts meta-information, such as, the packet size and arrival time. Kitsune reuses exactly the same 115 statistical features defined by Y. Meidan et al. [Mei+18] to describe network traffic. The role of the Feature Extractor is to compute those statistics from the meta-information provided by the Packet Parser. Then the role of the Feature Mapper is to map the 115-dimensional feature vector into k smaller sub-instances vector. In other terms, each of the k sub-instances vector contains a subset of the elements of the original feature vector. Note that the sum of the size of the k sub-instances vector is not necessarily equal to 115, as the same element of the original feature vector can be mapped to multiple sub-instances vector. The Anomaly Detector is composed of two layers: the Ensemble Layer and the Output Layer. The Ensemble Layer consists of k autoencoders (one for each of the k sub-instances vector). Each autoencoder of the Ensemble Layer reports the reconstruction error to the Output Layer. The Output Layer is also an autoencoder (the term 'Output Layer' used by the authors is misleading as it does not correspond to the output layer usually used to describe the output of a neural network) and its role is to compute the final anomaly score. Kitsune is tested on two real deployments of IP surveillance camera networks. Each network consists of four cameras. Attacks, such as, SYN Flood, OS scan and ARP spoofing are performed to generate malicious traffic. The authors report an AUC that range from 0.8 to 1, depending on the attack type.

T. D. Nguyen et al. [Ngu+19] present Dïot, an autonomous self-learning system for detecting compromised IoT devices. Dïot takes advantage of unsupervised machine learning techniques. It learns anomaly detection models using unlabeled crowdsourced data captured in client IoT networks. First, the type of the device that generated the traffic needs to be identified. The authors propose to reuse the device type identification techniques developed in [Mar+19]. Then, the normal communication profile of each device is learned and it can subsequently be used to detect anomalous deviations in communication patterns. Dïot monitors sequences of network packets. Each of the packet in the sequence are described using 7 features: the direction, the local and remote ports, the packet length, the TCP flags,

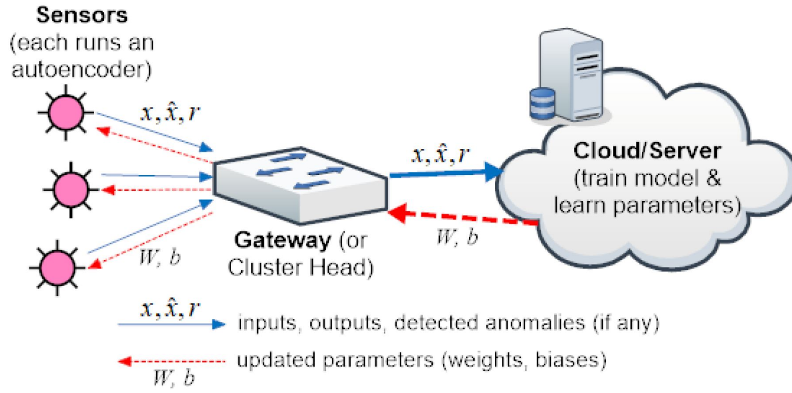


Figure 2.20: T. Luo et al. [LN18]: Architecture of a WSN that uses autoencoders for anomaly detection

the protocols and the IAT. The occurrence probability of each packet, given the k preceding packets is calculated. To compute these occurrence probabilities, a Gated Recurrent Unit (GRU) network is used. If the occurrence probability estimates of a sufficient number of consecutive packets fall below a detection threshold, it is deemed anomalous and an alarm is raised. The authors report that Dïot achieves an attack detection rate of 94.01% and a FPR of less than 1%.

T. Luo et al. [LN18] propose an autoencoder based anomaly detection method for Wireless Sensor Network (WSN) data. The method is not based on network traffic data but on the data collected by the sensors such as the temperatures. As illustrated in Figure 2.20, the computation-intensive task of model training is handled by the cloud. The trained model is then distributed to the sensors. Hence, the anomaly detection can be performed at sensor level without the need for communicating with the other sensors or the cloud. For the experiments, 8 sensors that monitor temperature and relative humidity are deployed at different locations of an office building. Each sensor has a sensing frequency of one reading every 2 minutes, which corresponds to 720 daily readings from a single sensor. Hence, the feature vector is a 720-dimensionnal vector that describes the data collected by each sensor on a daily basis. Anomalies are created by adding a Gaussian noise, with mean μ and variance σ^2, to the original data. The AUC is computed for different values of μ and σ^2. The AUC is greater than 0.8 most of the times. It is lower than 0.8 only when $|\mu| < 0.07$ and $\sigma^2 < 0.12$. That is, when the Gaussian noise component is too small to be noticeable by the anomaly detector.

2.3.3 Summary

Table 2.4 summarizes the different works on IoT network intrusion detection. It presents the algorithms and features used in the different works.

As shown in Table 2.5, the proposed IoT NIDSs lack at least one of the following capabilities: ability to detect new types of attack, possibility to place the NIDS

Table 2.4: Summary of works on intrusion detection in IoT networks

Reference	Type	Algorithms	Features
R. Doshi et al. [DAF18]	misuse detection	kNN, SVM with linear kernel, Decision Tree, Random Forest and FCNN	packet size, IAT, protocols used, total bandwidth, number of different destination IP
A. A. Diro et al. [DC18]	misuse detection	FCNN	NSL-KDD features: duration, protocol, source/destination bytes, number of failed login attempts, number of operation performs as root, number of file creation
K. Yang et al. [Yan+18]	misuse detection	XGBoost, Active Learning	NSL-KDD features
E. Hodo et al. [Hod+16]	misuse detection	FCNN	NA
H. H. Pajouh et al. [Paj+16]	misuse detection	Naïve Bayes, Certainty-Factor kNN	NSL-KDD features
N. Moustafa et al. [MTC18]	misuse detection	AdaBoost	source/destination IP addresses and ports, protocol type, last time of connection, length of MQTT data, number of MQTT connections from the same source and to the same destination, DNS names and query types, HTTP request method (e.g. GET, POST, HEAD)
P. Shukla et al. [Shu17]	misuse detection	Decision Tree	NA
Y. Meidan et al. [Mei+18]	anomaly detection	Vanilla Autoencoder	mean and variance of the packet sizes and the IAT computed over different time windows
Y. Mirsky et al. [Mir+18]	anomaly detection	Ensemble of Vanilla Autoencoder	mean and variance of the packet sizes and the IAT computed over different time windows
T. D. Nguyen et al. [Ngu+19]	anomaly detection	GRU network	local and remote ports, packet length, TCP flags, protocols, IAT
T. Luo et al. [LN18]	anomaly detection	Vanilla Autoencoder	readings from sensors (temperature, relative humidity)

outside the local network or the use of non-intrusive features.

A major limitation of misuse detection is that the trained model will not be able to detect attacks unseen during the training phase. The need to label the data is also another drawback of supervised learning. Indeed, labelling the data as being legitimate or malicious by a human supervisor is a daunting task. On the contrary, anomaly detection models do not require data labelling and allow the detection of previously unseen attacks.

Some works are limited because the type of devices that generate the network traffic needs to be known in order to redirect the traffic to the appropriate model (the one trained for that specific device). The developed model must be placed inside the local network because local information such as the MAC address of a device or the local port number are needed to determine the device type but also to derive the features. For example in [Mei+18], network traffic is aggregated using the MAC address of a device. One anomaly detector per device is trained and during the testing phase, the MAC address is used to determine what device is generating the ongoing network traffic in order to redirect it to the appropriate anomaly detector. In [Ngu+19], the local port number used by a device is used as a feature. In case the local IoT network is connected to the Internet through a NAT (which is often the case), information like MAC addresses or port used by the device are no longer available after the NAT.

IoT NIDSs can be categorized into two categories depending on the level of intrusiveness of the features they use: NIDS that only uses network level features such

Table 2.5: Limitations of existing works on intrusion detection in IoT networks

Reference	Anomaly detection	No need to know the type of the device that is generating the traffic/ Possibility to place the IDS outside the local network	Non-intrusive
R. Doshi et al. [DAF18]	No	Yes	Yes
A. A. Diro et al. [DC18]	No	Yes	Yes
K. Yang et al. [Yan+18]	No	Yes	Yes
E. Hodo et al. [Hod+16]	No	NA	NA
H. H. Pajouh et al. [Paj+16]	No	Yes	Yes
N. Moustafa et al. [MTC18]	No	No	No
P. Shukla et al. [Shu17]	No	NA	NA
Y. Meidan et al. [Mei+18]	Yes	No	Yes
Y. Mirsky et al. [Mir+18]	Yes	No	Yes
T. D. Nguyen et al. [Ngu+19]	Yes	No	Yes
T. Luo et al. [LN18]	Yes	Yes	No

as packets sizes or the number of bytes, and approaches that also include application level features such payload content or DNS names. In Table 2.5, approaches that use only network level features are said to be non-intrusive. Approaches using application level features cannot be used if the network traffic is encrypted.

In Chapter 3 of this thesis, we propose an IoT NIDS that can detect novel attacks, can be placed outside the local network and uses only network level features.

2.4 IoT Network Traffic Generation

Works on leveraging generative deep learning models to generate network traffic data can be categorized into two types: network flow generation and individual network packet generation.

A network flow is identified by its source IP address, destination IP address, source port, destination port and transport layer protocol. Features generated to characterize a flow are usually the duration, the total number of packets sent and the total number of bytes sent. A bidirectional flow also includes features such as the total number of packets received and the total number of bytes received. The features used to describe a packet are the different fields of the network layer (e.g., IP) or the transport layer (e.g., TCP or UDP) headers. Approaches addressing flow generation only generate flow-level features and do not attempt to characterize the individual packets that constitute the flow. On the other hand, work focusing on the generation of individual packets ignore the flow-level features and generated packets are independent from each other.

2.4.1 Flow-level Network Traffic Generation

M. Rigaki et al. [RG18] propose to leverage the power of GAN to create a malware that self-adapts its networking behavior to evade IPS. To this purpose, they train a GAN to learn to generate network flows that look like they were generated by the Facebook chat. The trained generator is then used by a malware to modify its network traffic and to make it look legitimate and avoid being blocked by an IPS.

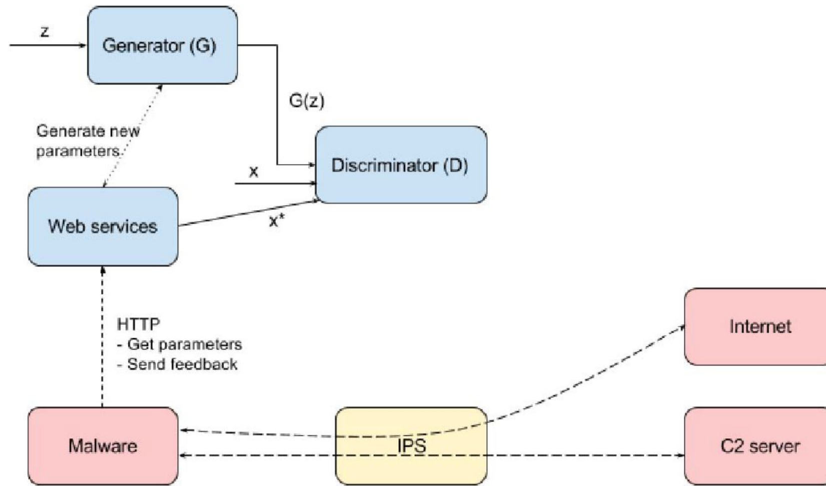


Figure 2.21: M. Rigaki et al. [RG18]: Network experiments setup. The GAN is implemented independently and communicates with the malware through a web service. The malware gets the parameters and modifies its traffic in real time. The C2 channel should be maintained and should be operational. The IPS blocks all the traffic that does not look like Facebook chat

Three features are generated: the total number of bytes in the flow, the duration of the flow and the time between the current flow and the next one. Both the generator and the discriminator of the GAN consist of one LSTM hidden layer. Once trained on Facebook chat network flow data, the generator part of the GAN is used to generate the three flow-level features. For the experiment, an open source remote access Trojan called Flu [20b] is used. A C&C server is also deployed for Flu to communicate with it. As illustrated in Figure 2.21, before any communication, Flu interrogates the GAN through an HTTP API to get the flow-level features. The features are then used by Flu to adapt its networking behavior. Stratosphere Linux IPS (SLIPS) system [Gar15] is used to detect whether or not the communication between Flu and the C&C server gets blocked. SLIPS uses Markov models to learn the profiles (total bytes size, duration and inter-flow time) of the network flows generated by a Facebook chat and by the Flu malware. A flow is blocked if it does not match the Facebook chat profile or if it matches the malicious profile. The obtained results show that, after 400 training epochs, the percentage of malicious traffic that is not blocked reaches 63.42%.

M. Ring et al. [Rin+19] propose to generate flow-based network traffic using GAN. They propose to generate legitimate network flows that can be used to evaluate NIDSs. The flow-level features they attempt to generate are the following: date first seen, duration of the flow, transport protocol, source/destination IP addresses, source/destination ports, total bytes in the flow, total number of packets in the flow and the TCP flags observed in the flow. They use GANs that process continuous data only. Therefore, they propose 3 different approaches to transform categorical features (IP addresses, ports, transport protocol, TCP flags) into continuous values.

The first method, referred to as numeric transformation, transforms the categorical variables into a set of continuous variables. For example, each byte of an IP address is converted to a value in the interval [0, 1]: 192.168.220.14 is transformed to four continuous attributes: $192/255 = 0.7529$, $168/255 = 0.6588$, $220/255 = 0.8627$ and $14/255 = 0.0549$. The second method, referred to as binary transformation, transforms each categorical attribute into a set of binary attributes. For example, each byte of an IP address is mapped to its 8-bit binary representation: 192.168.220.14 is transformed into a set of 32 binary attributes 11000000 10101000 11011100 00001110. The third method, referred to as embedding transformation or IP2Vec, transforms the original feature vector into an embedding vector in a continuous space, similarly to the Word2Vec model used in Natural Language Processing [Mik+13]. First, all the categorical features are one-hot encoded over their vocabulary size (for example the vocabulary size of the source IP address feature is the total number of unique source IP addresses present in the training dataset). For a given training instance, the value of some features are removed and are considered to be missing. Then, an FCNN with one hidden layer is trained to predict the missing values. The output of the hidden layer is later used as an embedded representation (in a continuous space) of the original feature vector. The original dataset is preprocessed using one of the three described transformation methods. Then a WGAN-GP (Wasserstein GAN with Gradient Penalty) [Gul+17b] is trained to learn to generate real looking feature vectors. For the experiment, the CIDDS-001 dataset [VR18] is used. The quality of the model is assessed by comparing the distributions of the generated ports and IP addresses to the real ones. The best results are achieved when embedding transformation is used to turn categorical attributes into continuous values.

Q. Yan et al. [Yan+19] propose to synthetize network flows that represent DoS attacks that looks legitimate enough to fool an NIDS. To this purpose, a WGAN-GP is trained on the KDD99Cup dataset [BG15]. The KDD99Cup dataset contains network flows of legitimate and attack traffic, described by 41 features. Some of these features are related to DoS attacks and hence cannot be modified. Otherwise, the generated network traffic will lose its DoS capability. Those are the *immutable* features and include the duration, the protocol or the TCP flags. The generator of the WGAN-GP generates feature vector that correspond to legitimate traffic. Then, a convertor (see Figure 2.22) is used to add DoS capability to the generated feature vectors before feeding them to the discriminator. The convertor just replaces the values of the immutable features so that the feature vector gains DoS capability. To assess the evasion capability of the synthetized DoS traffic, a CNN based NIDS is trained to detect DoS attacks using the same KDD99Cup dataset. It achieves an attack detection rate on real DoS traffic of 97.3%. But when presented with the synthetized DoS feature vectors, the attack detection rate drops to 47.6%. Hence, a significant proportion of the synthetized DoS traffic is able to bypass the CNN based NIDS.

J. Charlier et al. [Cha+19b] suggest to synthetize network flows that represent

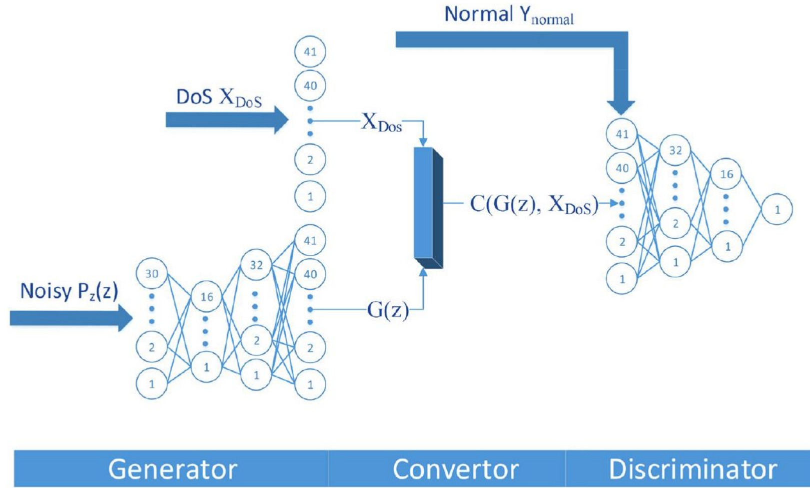


Figure 2.22: Q. Yan et al. [Yan+19]: Overview of DoS-WGAN architecture

DDoS attacks in order to create malicious datasets that can be used to assess the attack detection rate of commercial NIDSs. Hence, the synthetized DDoS traffic must be as realistic as possible. Two types of DDoS attacks are synthetized: Smurf and GoldenEye attacks. In a Smurf attack, the attacker sends a large number of ICMP echo request packets using a victim's spoofed source IP address. In a GoldenEye attack, HTTP requests are sent to a web server to open connections rendering the server unable to process legitimate requests. The NSL-KDD [BG15] and the CICIDS2017 [SLG18] datasets are used for experiment. The NSL-KDD dataset contains network flows representing Smurf DDoS attacks described by 41 features. The CICIDS2017 dataset contains network flows corresponding to GoldenEye attacks and described with 80 different features. A different WGAN-GP is trained for each of the datasets. A gradient boosting classifier is trained to distinguish between the real and the synthetized feature vectors. The AUC of the ROC Curve of the trained classifier is 0.75 (the closer the AUC is to 0.5 the better it is), highlighting that the classifier is partly unable to distinguish between the real and synthetized data.

D. Sun et al. [Sun+17] propose to generate DDoS attacks network traffics that mimic flash crowds to evade NIDS. A flash crowd is an unexpected surge in the number of legitimate visitors to a website. Network traffics are described by 5 flow-level statistical features: the total number of packets sent, the mean and standard deviation of the size of packets sent, the mean and standard deviation of the IAT between packets sent. To generate a feature vector, a Least Squares GAN (LSGAN) [Mao+17] is used. LSGAN adopts the least square loss function for the discriminator. For the experiment, CAIDA DDoS Attack 2007 dataset [20f] provides the DDoS data and the World Cup 1998 dataset [98] is used for the flash crowds data. The quality of the generated feature vectors is assessed by feeding them to a Random Forest classifier trained to differentiate between DDoS and flash

crowds. The classifier achieves an accuracy of 99% for the real traffic data. The accuracy drops to 55% when the classifier is presented with the generated DDoS feature vectors, which is very close to random guessing.

2.4.2 Network Packets Generation

Z. Lin et al. [Lin+19] propose to leverage the capability of GANs for oblivious network analysis (network analysis without any prior knowledge of the network protocol used). Indeed, most black-box devices use proprietary network protocols with unknown format. Two tasks are explored: 1) generating synthetic protocol compliant messages given only a few number of messages coming from a black-box device; and 2) generating attack inputs that can impair a black-box device that uses an unknown network protocol. For the generation of protocol-compliant packets, the authors first define a custom stateless network protocol. A packet from the custom protocol has two types of field dependencies: intra-field dependencies (a field can only take a certain number of valid candidate values) and inter-field dependencies (a field is a function of one or more other fields). The custom protocol contains two intra-field dependencies: a multiple of 4 and a printable ASCII character. It also contains inter-field dependencies commonly found in network protocols, such as a XOR operation of two other fields, a field that contains the length in bits of another field or a cyclic redundancy check (CRC). A WGAN-GP is trained to learn to generate packets that comply with the created custom protocol format. Note that the GAN has no prior knowledge about the protocol format and hence generate every single bit that compose a packet from scratch. After training, the generator has successfully learnt most of the intra and inter field dependencies and is able to generate protocol compliant packets. The only field that the generator fails to generate properly is the CRC. The authors think this is because computing a CRC requires repeated long division by a fixed generator polynomial that is unknown to the GAN. For the second task of identifying input packets that trigger network attacks, the authors consider the DNS protocol. They aim to train a GAN that learns to generate packets that trigger DNS amplification attacks. A generated malicious DNS packet is considered to be of interest if it triggers a response that is at least 10 times the size of the request. A packet that triggers a successful attack is referred to as a *positive* packet, the rest are *negative* packets. The main difficulty is the scarcity of positive packets. To overcome this issue, the authors propose to use Conditional GAN (CGAN) and a custom training process. The idea is that since positive and negative samples are complementary, a CGAN can be trained to jointly learn to generate both positive and negative packets. The CGAN takes a binary condition (positive or negative) as input that can later be used to specify to the generator the type of packet we want to generate. The authors also propose a new training mechanism, as shown in Figure 2.23: packets are generated during training with $condition = positive$, they are passed through the system to get their actual label, and are added to the training data. For 5 fields of the DNS request,

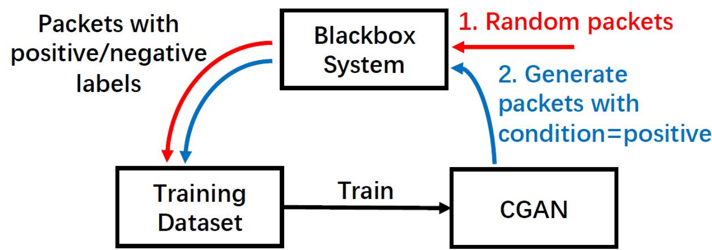


Figure 2.23: Z. Lin et al. [Lin+19]: CGAN training mechanism

the CGAN is provided with candidate values to choose from. But the CGAN has no prior knowledge about 12 other fields of the DNS request and all possible bits are explored. Hence, this is not really a complete black-box setting. Once trained, the generator is used to generate 100,000 packets with the $condition = positive$. 778 out of these 100,000 packets (0.78%) are actually positive and trigger DNS amplification attack.

A. Cheng [Che19] presents PAC-GAN, a CNN based GAN trained to generate network packets. A network packet is converted to an image-like representation. The value of each byte of a network packet is mapped to a pixel. However, as the primary use of CNN based GANs is for image generation, they might be prone to certain types of errors when dealing with raw packet data. First, when generating images, minor variations of the pixel values is not perceptible with the naked eye and hence does not degrade image quality. But for network packet generation, a slight variation of a single byte value might result in an invalid packet. Second, in an image, pixels located close to each other might share very similar values and CNNs exploit this specific property of an image. Hence, it is suitable to create clusters of similar byte values in the image-like representation of a network packet. With those characteristics of CNN based GANs in mind, the author proposes a specific encoding scheme to map the byte values of a network packet to the pixels of a grayscale image as shown in Figure 2.24. In the first step, the hexadecimal representation of a byte is split into two digits. Each hexadecimal digit is then assigned to a subrange. For example, the byte value 0x1F is first split into hexadecimal digits 1 and F. The digit 1 is assigned to the subrange 0x10 to 0x1F and "F" is assigned to the subrange 0xF0 to 0xFF. The midpoints of the subranges, 0x18 and 0xF8, are used as the values of the pixel intensities. This step makes the model robust to small variations of a pixel value, as any pixel value that lies in the subrange will decode to the same byte value. In the second step, the converted byte values are duplicated across the rows and the columns. That is, each converted byte value is mapped to a 2×2 sub-matrix. This allows the CNN to effectively exploit clusters of similar pixels. For experiments, PAC-GAN is trained to generate single packets representing ICMP Ping requests, DNS queries and HTTP GET requests. All byte fields of a packet are created by the GAN, except for the checksums. A packet is considered to be a success if when it has been sent to the Internet a valid network response is received. PAC-GAN

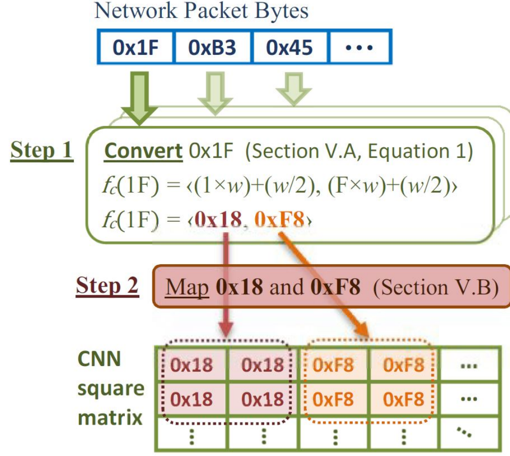


Figure 2.24: A. Cheng [Che19]: Conversion and one-to-multi mapping process

achieves the following success rates: 76%-90% for Ping requests, 95%-99% for DNS queries and 76%-79% for HTTP GET requests.

2.4.3 Summary

Table 2.6 summarizes the different works on generative deep learning based network traffic generation. For each work, it presents the types of traffic generated, as well as the algorithms and features used.

As shown in Table 2.7, existing works are limited in that they either focus on flow-level or packet-level traffic generation, but not a combination of both. The resulting traffic is incomplete since a flow and the packets composing it are closely linked. For example, the number of bytes exchanged for the duration of a flow usually equals the sum of the sizes of each individual packet that composes the flow. Moreover, none of the existing works focuses on IoT network traffic generation. The data used in all works come from general-purpose networks composed of devices such as PCs, laptops or smartphones.

The limitation of existing work on flow-level features generation is that they do not attempt to characterize individual packets that compose a flow. They do not consider more fine-grained features, such as the size of the individual packets composing a flow. For example, if the generated flow-level feature is the total number of bytes per flow, then any combination of packet sizes that adds up to that value are considered to be valid. Traffic generation based only on flow-level features is not realistic enough and will fail to fool network monitoring tools that perform packet-level analysis. Works focusing on packet generation treat packets individually and independently from each other, failing to capture the sequential nature of network communications. Traffic generation based solely on packet-level features will fail to fool network monitoring tools that perform flow-level analysis.

In Chapter 4 of this thesis, we propose a network traffic data generator for the IoT that generates both packet-level and flow-level features at the same time.

Table 2.6: Summary of works on network traffic generation using generative deep learning models

Reference	Level	Traffic Type	Algorithms	Features
M. Rigaki et al. [RG18]	Flow	Malicious	LSTM-GAN	total number of bytes in the flow, duration of the flow and the inter-flow time
M. Ring et al. [Rin+19]	Flow	Legitimate	WGAN-GP	date first seen, duration of the flow, transport protocol, source/destination IP addresses and ports, total bytes in the flow, total number of packets in the flow and the TCP flags observed in the flow
Q. Yan et al. [Yan+19]	Flow	DDoS	WGAN-GP	NSL-KDD features: duration, protocol, source/destination bytes, number of failed login attempts, number of operation performs as root, number of file creation
J. Charlier et al. [Cha+19b]	Flow	DDoS	WGAN-GP	NSL-KDD features
D. Sun et al. [Sun+17]	Flow	DDoS	LSGAN	total number of packets sent, mean and standard deviation of the size of packets sent, mean and standard deviation of the IAT between packets sent
Z. Lin et al. [Lin+19]	Packet	Legitimate and DNS amplification attack	WGAN-GP CGAN	different fields of a packet using a custom protocol, different fields of a DNS request packet
A. Cheng [Che19]	Packet	Legitimate (ICMP Ping, DNS query and HTTP GET)	CNN-GAN	every single byte composing a network packet

Table 2.7: Limitations of works on network traffic generation using generative deep learning models

Reference	Flow-level	Packet-level	IoT dataset
M. Rigaki et al. [RG18]	Yes	No	No
M. Ring et al. [Rin+19]	Yes	No	No
Q. Yan et al. [Yan+19]	Yes	No	No
J. Charlier et al. [Cha+19b]	Yes	No	No
D. Sun et al. [Sun+17]	Yes	No	No
Z. Lin et al. [Lin+19]	No	Yes	No
A. Cheng [Che19]	No	Yes	No

3

IoT Network Traffic Monitoring

"Education is the passport to the future, for tomorrow belongs to those who prepare for it today"

– Malcolm X

The study of the state of the art pointed out the limitations of existing works in machine learning based IoT network monitoring systems. Works on IoT device identification fail to be *delay-free*, *phase-independent* or *non-intrusive*. While works on IoT NIDSs are limited either because they can detect only certain types of attacks or because they require to know the type of the device that is generating the network traffic (which might not be possible if the NIDS is deployed outside the local network).

In this chapter, we focus on developing machine learning based IoT network monitoring tools that overcome the limitations of existing works. The following two types of monitoring applications are explored:

- *IoT device type recognition system* (Section 3.1): its purpose is to determine the type of a connected device by analyzing its network traffic. The developed system must be *delay-free*, *phase-independent* and *non-intrusive*. First, a set of features that respect the aforementioned constraints are defined to describe bidirectional flows. Then, an experimental smart home network composed of four devices is built to produce real IoT network traffic data. Data visualization using t-SNE is performed to get an insight of the collected bidirectional flows, and to assess the effectiveness of our selected set of features in differentiating the different IoT devices. Next, different machine learning algorithms are trained to classify bidirectional flows based on the IoT device type they belong to.

- *IoT NIDS* (Section 3.2): its purpose is to detect malicious activities. It must be able to detect new types of attack and not require prior knowledge about the type of the device that is generating the network traffic. To this purpose, a set of sparse autoencoders is trained. Due to the great diversity of IoT devices (camera, smart bulb, motion sensor, etc), the network communications behavior can vary greatly from one device to another. Hence, one separate sparse autoencoder is trained for each IoT device type. During the testing phase, we will explore two different possibilities: either it is possible to determine the type of the device that is generating the network traffic (and redirect it to the appropriate autoencoder) or it is not (we have to feed it to all the trained autoencoders).

3.1 IoT Device Recognition through Network Traffic Classification

3.1.1 Overview

Determining the type of device connected to the network will help to better enforce network security. For example, once it is determined that a device is a security camera from a specific manufacturer, appropriate filtering rules can be specified so that the camera will not be allowed to do anything else than what it is expected to do. Device type recognition can also be used to block access to the network to blacklisted devices. IoT device type recognition can also be used for malicious purposes. For example, an attacker can discover vulnerable IoT devices by performing passive network traffic analysis. Given the wireless nature of IoT networks, an attacker can easily capture the network traffic to perform further analysis with the purpose of recognizing the type of connected devices. Device fingerprinting through network traffic analysis can also help malware to identify vulnerable devices passively. Instead of actively scanning for a device to infect, passive vulnerable device discovery will reduce the network signature of the malware making its detection even more difficult for intrusion detection systems. Device type recognition also raises privacy concerns. Once the type of a device has been determined, further analysis of the traffic can allow an intruder to determine the current state of the device. For example, in [Aca+18], multi-class classification is performed to associate network traces of a device to a corresponding device state (see Section 2.2). In the case of a smart home, such information can help to infer what is happening inside the house, leading to potential privacy breach.

Given the huge diversity of IoT devices, it is difficult to come up with a specific network signature for each device type. Therefore, machine learning algorithms are better suited to learn patterns in the network traffic and differentiate one device type from another. The state of the art has highlighted the limitations of existing works on machine learning based IoT network traffic classification. Some of the works extract features from the full communication without using any timeout, requiring

Figure 3.1: Network traffic classification pipeline

the user to wait for a long period of time before being able to determine the type of a device [Mei+17a; Mei+17b]. Other works are limited because they focus on specific phase of the life cycle of a device or specific network packets type. For example, in [Mie+17], the type of an IoT device is determined at its setup phase. Hence, the proposed method cannot be used if we have missed the setup phase or if we want to determine the type of devices that are already connected to the network. A number of works are also limited because they use features extracted from the application layer [Siv+18], requiring the network traffic not to be encrypted. Looking at application level information also rises privacy concerns.

We propose to perform IoT device type recognition through network traffic classification. To overcome the limitations of existing works, we suggest to develop a method that is:

- *delay-free*: does not require to wait for an undetermined amount of time.
- *phase-independent*: can be used at any moment of a device's life cycle (provided that the device is generating network traffic).
- *non-intrusive/privacy-preserving*: does not require to look at application level data. Network traffic can be encrypted.

First, a set of features that respect the constraints described above is determined. Visualization techniques allow us to determine if our selected set of features is effective enough to differentiate between the different IoT device types. By IoT device type, we refer to a device model from a specific manufacturer. Then, machine learning based network traffic classifiers are trained to determine the type of IoT devices connected to a network. Figure 4.2 shows the workflow of the developed model. Raw network traffic is first preprocessed to extract useful features. The extracted features are then fed to a classifier to determine the type of the IoT device that generated the network traffic.

3.1.2 Features Description

The purpose of the described model is to provide a means to recognize devices based only on their networking behavior. Therefore, we need to define features that will appropriately describe the network activity. As described in Section 2.2, most existing works on IoT network traffic classification use features extracted from full TCP sessions. The issue with this method is that we have to wait until the end of the session in order to be able to extract all the features. For some IoT devices such

as the Nest security camera, a single TCP session can last for days. Other works focus only on the network activity during the setup phase of a device. However, this is of no help if we have missed the setup phase of the device, which is the case if we are sniffing a network in which all devices are already set up. Another constraint is that the features have to be extractable even if the network traffic is encrypted. Moreover, we want to avoid looking at application level information for privacy concerns. Hence, the features must be extractable from link, network or transport layer of the OSI model.

We propose to work with bidirectional flows identified by their source and destination IP addresses and ports. In the case of long TCP connections, we do not capture the whole session. Instead, a timeout is used to split long connections into several bidirectional flows. Each bidirectional flow is described by a feature vector composed of the following:

- The sizes of the first N packets sent
- The sizes of the first N packets received
- The $N-1$ packet inter-arrival times between the first N packets sent
- The $N-1$ packet inter-arrival times between the first N packets received

Packets *sent* and *received* are from the perspective of the IoT device we want to identify (usually internal to the network of the monitoring gateway). The sizes of the packets and the inter-arrival times have already proven their effectiveness in non-IoT works, especially works on application identification [Qaz+13][Lin+16]. Since all feature vectors must have the same size, if a bidirectional flow contains less than N packets, the remaining fields of the vector are set to zero. The more devices we have to classify the greater N has to be so that the classifier will have enough information to accurately differentiate the bidirectional flows generated by the different devices. On the other hand, N has to be small enough to avoid performance related issues. Unless specified otherwise, the variable N is set to 10 for the rest of this Section. That is, the total number of features is equal to 38. In Section 3.1.5, we will test other values of N to examine the impact of the variable N on the overall accuracy of a classifier. A timeout of 600 seconds is used to split long connections into multiple bidirectional flows.

3.1.3 Smart Home Dataset Description

Because of the lack of publicly available IoT network traffic data, we propose to build a small experimental smart home network to generate our own network traffic. It consists of four IoT devices: a Nest security camera, a D-Link motion sensor, a TP-Link smart bulb and a TP-Link smart plug. The four devices are managed using a smartphone. Note that none of the devices send the collected data directly to the smartphone. Instead, the data is sent to a cloud service provided by the vendors. The collected data is then transmitted from the cloud services to the

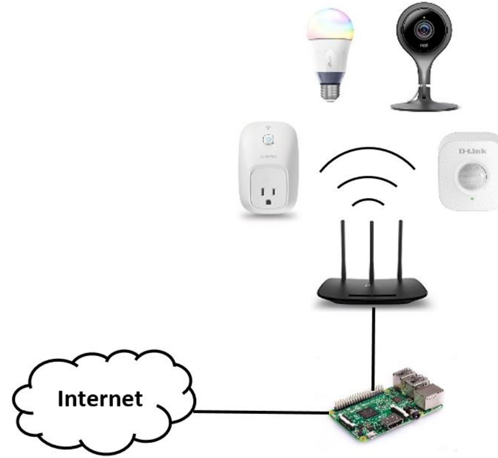


Figure 3.2: Experimental smart home network

Table 3.1: Functionalities provided by each device and explored during the network traffic collection phase

Device	Used functionalities
Nest security camera	switching on/off the camera, watching live the recorded video and audio data, turning on/off the microphone, turning on/off the alert mode (when the alert mode is turned on, a notification is sent to the mobile app whenever an unusual sound or motion is detected), sending audio messages (the camera comes with an integrated speaker that can be used to send audio messages), changing the quality of the video recording.
TP-Link smart bulb	switching on/off the bulb, changing the brightness and the color of the light emitted by the bulb
TP-Link smart plug	switching on/off the device remotely via the mobile application but also physically via the button on the device
D-Link motion sensor	switching on/off the sensor, setting its sensitivity.

application running on the smartphone. Similarly, when a command is sent from the smartphone to a device, it first goes to the cloud service. From there, it is redirected to the device. During the network traffic collection period, we actively interacted with the devices doing our best to explore all the functionalities provided by each device. Exploration of the maximum number of functionalities is important because, depending on the task performed by a device, its networking behavior might be different. For experimental reproducibility, we list the different ways we interacted with each device in Table 3.1.

All the devices connect to the Internet through a wireless access point. The network traffic is collected thanks to a Raspberry Pi placed between the wireless access point and the Internet as shown in Figure 3.2. The raw network traffic is then preprocessed as follows:

1. the MAC addresses of the devices are used to split the network traffic into different pcap files corresponding to different IoT devices. This will facilitate the labeling of the dataset.

2. the bidirectional flows along with their timestamp and protocol are extracted

Table 3.2: Total number of bidirectional flows per device

	train	test
D-Link Motion Sensor	867	207
Nest Security Camera	839	216
TP-Link Smart Bulb	821	219
TP-Link Smart Plug	695	163
Total	**3222**	**805**

from the different pcap files. Only TCP flows are kept as all of the devices use HTTP or HTTPS for communications.

3. the bidirectional flows of the different IoT devices are merged to form a single dataset. The timestamp is used to reorder the flows.

Network traffic has been collected for 7 days. Resulting to a dataset containing a total of 4027 bidirectional flows. The training set consists of the first 3222 collected bidirectional flows (80% of the dataset), while the test set consists of the remaining 805 bidirectional flows (20% of the dataset). Table 3.2 shows the number of flows for each device in the training and test sets.

3.1.4 Experimental Results - Network Traffic Visualization

Data visualization is important to get an insight of our 38-dimensional network traffic data and to assess the discriminative power of our model. We want to know if the features we have selected to describe the networking behavior are discriminative enough to differentiate the network traffic produced by the different IoT devices. Although, dimensionality reduction results in information loss, it still gives an idea of the general structure of the dataset.

We present the results of data visualization using t-Distributed Stochastic Neighbor Embedding (t-SNE) [MH08] which is a non-linear dimensionality reduction technique. t-SNE has no knowledge of the label of the input data. It is an unsupervised learning algorithm. t-SNE outperforms many other non-parametric data visualization and exploration techniques. Another commonly used dimensionality reduction technique is Principal Component Analysis (PCA). The limitation of PCA is that, like any other linear dimensionality reduction methods, it only focuses on placing dissimilar data points far apart in the lower dimension. It does not attempt to place similar data points close together. Differently, t-SNE attempts to represent similar data points close to each other while preserving the global structure of the dataset. Therefore, t-SNE is a much better option for visualization.

The obtained visual representation of the data is shown in Figure 3.3. The data points form visual clusters corresponding to the network traffic generated by different IoT devices. Most of the data points corresponding to the same IoT device lie close to each other whereas data points from different IoT devices lie far apart. The obtained representation indicates that the features selected to describe the network

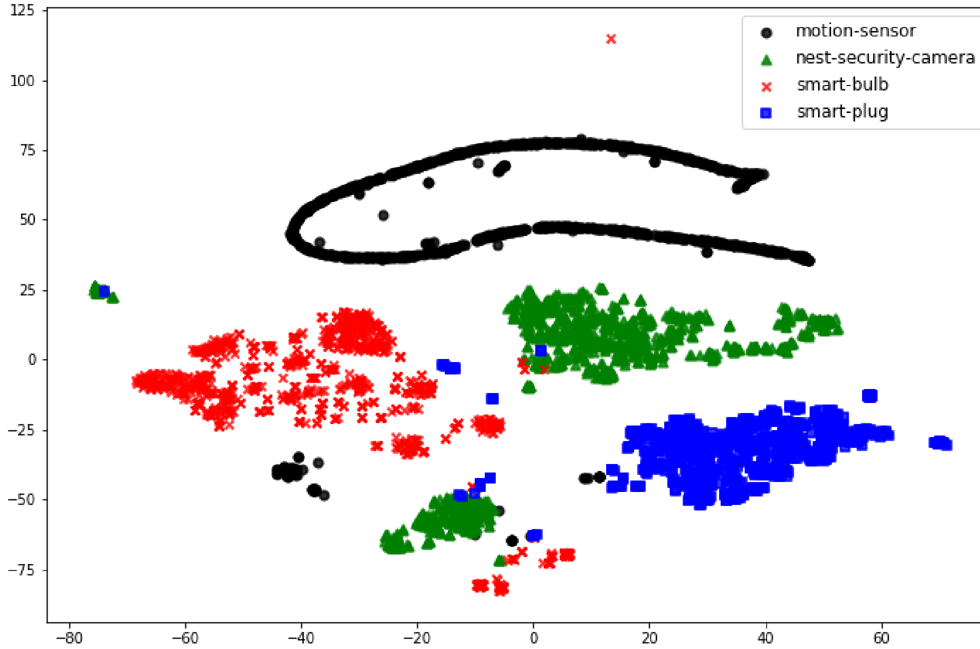


Figure 3.3: Dataset visualization using t-SNE

traffic are discriminative enough to distinguish between the different IoT devices present in our dataset.

3.1.5 Experimental Results - Classification

For network traffic data classification, six different classification algorithms are tested: Random Forest, Decision Tree, Support Vector Machines (with radial basis function kernel), k-Nearest Neighbors, Artificial Neural Network (ANN) and Gaussian Naïve Bayes.

A Decision Tree is a tree-like structure in which each internal node represents a "test" on the value of a feature (e.g. *is the size of the 3rd packet sent greater than 60 ?*), each branch represents the result of the test (yes or no), and each leaf node represents a class label. Hence, the paths from root to a leaf node represent a series of classification rules. Decision Trees tend to overfit the training set. To correct the tendency of a single Decision Tree to overfit, Random Forest classifiers average multiple Decision Trees, each trained on different random parts of the training set. An SVM classifier maps instances to points in a space where instances from separate categories can be divided by a "gap" (or margin) as wide as possible. With k-NN algorithm, the class of a new instance is the class that is the most common among its k nearest neighbors in the training set. Because k-NN require to store the whole training set, it requires high memory, is computationally expensive and can be very slow if the training set is large. Naïve Bayes are simple probabilistic classifiers with strong (naïve) independence assumptions between the features. The ANN used is a fully connected feedforward neural network consisting of two hidden layers with 10

Table 3.3: Best hyperparameter values for the different classifiers

RF	maximum depth: 10, number of estimators: 10
DT	maximum depth: 50
SVM	C: 1000, gamma: 0.001
KNN	number of neighbors: 5
ANN	learning rate: 0.01, number of epochs: 40 (early stopping)
GNB	None

Table 3.4: Overall performance on the test set of the different classifiers

	accuracy	**micro-av. precision**	**micro-av. recall**	**micro-av. F1 score**
RF	.999	.999	.999	.999
DT	.995	.995	.995	.995
SVM	.993	.993	.993	.993
KNN	.989	.989	.989	.989
ANN	.986	.986	.986	.986
GNB	.919	.919	.919	.919

neurons each and using a 0.5 dropout rate (more details on ANNs can be found in Section 2.1).

Most of the tested algorithms have hyperparameters that need to be appropriately tuned in order to avoid underfitting or overfitting. During the training phase, a validation set consisting of 25% of the training set is used to fine tune the hyperparameters. Once the best parameters have been found, the classifier is retrained on the whole training set. Table 3.3 shows the obtained best hyperparameter values for each classifier.

The performance of the different algorithms are measured on the test set. The metrics used are the accuracy, the precision, the recall and the F1 score. To assess the overall precision, recall and F1 score, micro-averaging is used (see Section 2.1.3 for more details about the metrics used and micro-averaging).

The precision, recall and F1 score for each device are also calculated individually as shown in Tables 3.5, 3.6 and 3.7. This is done by simply considering the problem as if it was a binary classification problem, with the device we are calculating the performance for, corresponding to the positive class.

Table 3.5: Precision on the test set of the different classifiers and for specific devices

	sensor	**camera**	**bulb**	**plug**
RF	1.	1.	.995	1.
DT	.986	1.	.995	1.
SVM	1.	.977	.995	1.
KNN	1.	.977	.986	.994
ANN	.986	.986	.978	1.
GNB	1.	.771	1.	.993

Gaussian Naive Bayes is the algorithm that performs the worst with an overall

Table 3.6: Recall on the test set of the different classifiers and for specific devices

	sensor	**camera**	**bulb**	**plug**
RF	1.	1.	1.	.994
DT	1.	.991	.995	.994
SVM	.995	1.	1.	.969
KNN	.990	1.	.986	.975
ANN	.995	.986	1.	.957
GNB	.971	1.	.785	.926

Table 3.7: F1 score on the test set of the different classifiers and for specific devices

	sensor	**camera**	**bulb**	**plug**
RF	1.	1.	.997	.997
DT	.993	.995	.995	.997
SVM	.997	.988	.997	.984
KNN	.995	.988	.986	.984
ANN	.990	.986	.989	.978
GNB	.985	.871	.880	.958

accuracy of 91.9%. All other algorithms achieve higher performance with an overall accuracy on the test set ranging from 98.6% to 99.9%. The best performance is achieved by the Random Forest classifier with equally high overall accuracy, precision and recall. Despite the relative small size of our dataset, ANN achieves an overall accuracy of 98.9%. The performance of the ANN can be improved by collecting more network traffic data to increase the size of the dataset. These positive experimental results indicate that it is possible to recognize IoT devices with high accuracy by passively analyzing network traffic characteristics such as the size of the first packets sent and received along with the inter-arrival times between those packets.

To analyze the impact of the variable N (the number of packets sent and received that are taken into consideration by a classifier) on the overall accuracy, we only consider the Random Forest classifier as it is the classifier that achieved the best performance. We train multiple Random Forest classifiers for different values of N, ranging from 2 to 10, to find out the optimal value of N. Figure 3.4 shows the overall accuracy achieved by the different trained classifiers. The overall accuracy increases as the value of N goes up. Surprisingly, when N is set to a value as small as 2 the classifier still achieves a high accuracy of 98.9%. Such a high accuracy is reached even with a small value of N because our experimental network is very small and consists of only four different devices. Therefore, the size of the first two packets sent and received and the inter-arrival times, provide enough information to the classifier to accurately differentiate between the different IoT devices. Indeed, the greater the number of different devices connected to the network is, the more packets the classifier has to consider in order to accurately differentiate the flows corresponding to the different devices. For our experimental network, the accuracy reaches the maximum value of 99.9% for N equal to 6 and higher. Hence, if we take time and resource efficiency into consideration in the case of our network, the

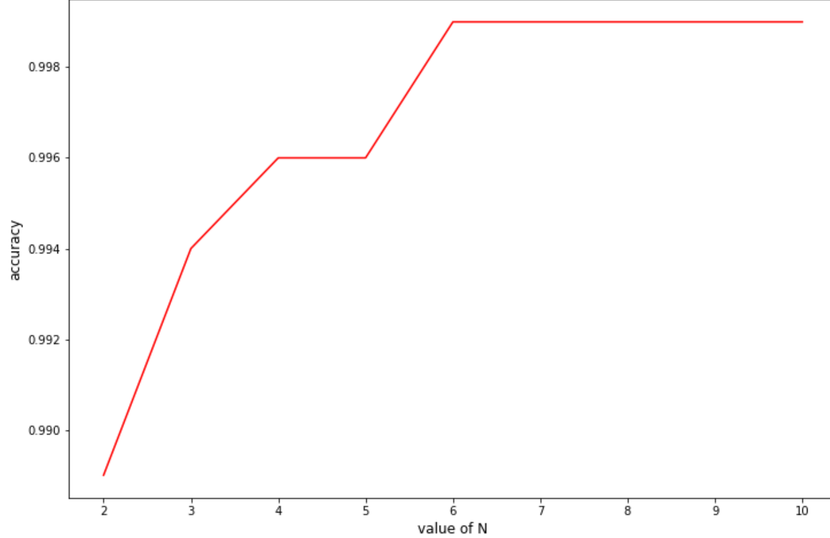


Figure 3.4: Overall accuracy achieved by Random Forest classifier for different values of N

optimal value of N for the Random Forest classifier is 6.

3.1.6 Discussion

One limitation of our work is the small number of IoT devices used. The experimental smart home network consists of only four different devices. The classifiers should be trained and tested on larger IoT networks containing a wide variety of devices. However, as the number of different devices connected to the network increases, more resources will be required to train the classifiers. With thousands of different types of devices available in the market, it can become difficult to train and maintain up to date a single classifier. If a new device has to be incorporated to the dataset or if a system update changes the networking behavior of a single device, the classifier should be retrained on the whole dataset. One solution is to train one classifier per device type as in [Mie+17]. However, in that case, all the different classifiers (one for each device type present in the network) have to be run in parallel, increasing the resource used during the operational use of the model. An intermediate solution is to train classifiers for groups of devices that share similar behaviors.

The fact that our experimental network is composed exclusively of IoT devices is another limitation of our work. In a general-purpose network, most activity will be generated by smartphones or laptops. Given the wide variety of tasks performed by a laptop, it is completely possible that some flows it generates share similar characteristics to the ones generated by a security camera and hence end up being classified as being generated by a security camera. However, the proposed approach can be easily adapted to general-purpose networks in which smartphones or laptops are also connected by adding an extra *non-IoT* class. During training, the classifier

will also be fed with network traffic data generated by non-IoT devices. Then, during testing, majority voting on a sequence of classified flows, similarly to what is proposed in [Mei+17a], can be used to determine the type of a device. For example, if the following sequence is obtained *(camera, non-IoT, camera, non-IoT, non-IoT)* then the device is considered as not being an IoT device.

3.1.7 Summary

In this Section, we proposed a machine learning based IoT device type recognition system that is *delay-free*, *phase-independent* and *non-intrusive.* Bidirectional flows, extracted from the raw network traffic, were described using features such as the size of the first N packets sent and received and the inter-arrival times between packets. Data visualization using t-SNE pointed out the effectiveness of our selected set of features in distinguishing between the bidirectional flows produced by the different IoT devices. Different machine learning algorithms were trained to classify the bidirectional flows based on the IoT device they belong to. An overall accuracy of 99.9% has been achieved by the Random Forest classifier.

3.2 IoT Network Anomaly Detection

3.2.1 Overview

Once the type of a device is known, the next step is to monitor its networking behavior to detect any malicious activities. The state of the art pointed out the limitations of existing works on IoT NIDSs. Some proposed approaches based on supervised machine learning algorithms are only able to detect certain types of malicious activities and do not allow the detection of previously unseen attacks [DAF18; DC18]. Other works require the type of device that is generating the network traffic to be known in order to redirect the traffic to the appropriate model (the one trained for that specific device) [Ngu+19; Mei+18]. In some cases it is not possible to know what device generates the network traffic. This is the case when the IoT network connects to the Internet through a NAT (*Network Address Translator*) and the NIDS is located outside the local network.

To overcome the limitations of existing works, we suggest to develop an intrusion detection method which is:

- capable to detect novel attack types;
- *device-agnostic*: i.e, it does not require to know the type of device that generated the network traffic. Hence, it can be placed outside the local network;
- *non-intrusive/privacy-preserving*: i.e., it does not require to look at application level data. Network traffic can then be encrypted without affecting analysis;
- *delay-free*: does not require to wait for an undetermined amount of time.

In this section, we develop an IoT NIDS based on unsupervised learning, specifically anomaly detection algorithms. This allow our model to detect new types of attacks. We also explore two different situations, depending on whether or not it is possible to know what device is generating the network traffic. To this purpose, we propose to detect anomalous communications in IoT networks using a set of sparse autoencoders. Autoencoders are unsupervised neural networks that can be used for anomaly detection thereby allowing the detection of new types of attacks. First, network communication data are preprocessed to extract useful features. The preprocessing step also includes features normalization. The normalized data is then fed to multiple sparse autoencoders. As an IoT network is composed of very different IoT device types (a device type is device model from a specific manufacturer), we train a distinct sparse autoencoder for each IoT device type present in the network. The autoencoder learns the legitimate communication profile of a given device type. During the testing phase (or actual deployment phase), the trained sparse autoencoders are used to detect any deviation from the learnt legitimate profiles in two different settings: either it is possible to know the type of the device producing the ongoing network traffic or it is not.

3.2.2 Features Description

The features used to describe the network traffic must be application agnostic. Moreover, the features must be extractable even if the application level data are encrypted. By removing the need to see the content of network packets, our solution will be less intrusive than other existing solutions (see Section 2.3). For our model to be *delay-free*, computing the features should not require to wait for an undetermined amount of time. To this purpose, timeout value is used.

Similarly to the work on IoT network traffic classification (Section 3.1), network traffic data are preprocessed in order to extract bidirectional TCP flows identified by their source and destination IP addresses and ports. The features describing the bidirectional TCP flows are statistics on the size of the first N packets sent and received, along with statistics on the corresponding inter-arrival times. The features are described in Table 3.8. Most of the selected statistics (mean, median, standard deviation, etc.) have proven to be effective in other works on anomaly detection [Mei+18] and application identification [Qaz+13; Kum+14]. A timeout is used to split long TCP connections into multiple bidirectional flows and avoid waiting indefinitely.

Count stands for the actual number of packets sent or received while waiting for the first N packets. For example, if the total number of packets sent for the duration of the timeout is equal to m so that m is lower than N, the corresponding value of count is equal to m. Otherwise, it is equal to N. For the inter-arrival times to exist, N should be equal or greater than 2. If a communication contains only one packet sent or received, the statistics that cannot be calculated, such as, the mean and standard deviation of the inter-arrival times between packets, are all set to 0.

Table 3.8: Features used to describe bidirectional TCP flows

Features
Mean, Median, Min, Max, Standard deviation and Count of the size of the first N packets sent
Mean, Median, Min, Max, Standard deviation and Count of the size of the first N packets received
Mean and Standard deviation of the IAT between the first N packets sent
Mean and Standard deviation of the IAT between the first N packets received

Note that contrary to the work on IoT network traffic classification, we do not use the raw values of the first N packet sizes and the inter-arrival times. Instead, we compute statistics over the sizes of the first N packets and the inter-arrival times. Computing statistics provide us with smoother values that are more robust to small variations of the underlying raw values. Robustness to small variation of the raw values can help to significantly reduce the number of false positive in anomaly detection based algorithms. For example, if, for any reason, the size of a single packet of a legitimate communication is unexpectedly large or small, the computation of statistics over multiple packet sizes will attenuate the impact of that single packet size.

Our study focuses only on TCP protocols as all the devices used for experiments use HTTP/HTTPS for communications. This also makes sense as most of the network communications generated by IoT malware use TCP [Sym20]. However, the proposed method can be easily extended to UDP communications.

3.2.3 Sparse Autoencoder for Anomaly Detection

To be able to detect new types of attacks, we will use unsupervised learning algorithms that allow to perform anomaly detection. More precisely, autoencoders that have proven to be successful in other network intrusion detection works [Mei+18; Mir+18]. In this section, we first define what a sparse autoencoder is, with specific mathematical notations. We also introduce the concept of reconstruction error and how it can be used to perform anomaly detection.

Sparse Autoencoder Training

As explained in Section 2.1.4, autoencoders are unsupervised artificial neural networks that learn to copy their inputs to their outputs under some constraints. The constraints are added at the hidden layer. They force the autoencoder to learn an efficient representation of the input data. Sparsity is such constraint that can be used to force the autoencoder to learn an efficient representation of the input data. In sparse autoencoders [Ng+11], the number of neurons in the hidden layer is usually greater than the number of inputs as shown in Figure 3.5. Sparsity is added by forcing the autoencoder to reduce the number of active neurons in the hidden

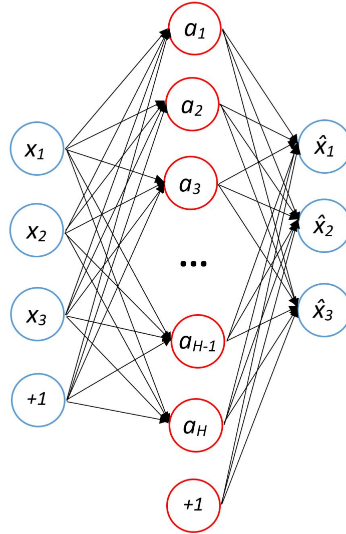


Figure 3.5: Sparse Autoencoder

layer. A neuron is considered to be active if its output is close to 1 and inactive if its output is close to 0. For example, one can constrain the autoencoder to have on average 1% of significantly active neurons in the hidden layer. To do so, first, the average activation of each neuron in the hidden layer is computed. Then neurons that have on average an activation greater than the targeted activation are penalized by adding a sparsity loss term to the cost function.

Let n be the total number of features of our input vector. Given an input vector $x = (x_1, x_2, \ldots, x_n)$, a neural network can be viewed as a complex non linear hypothesis function $h_{W,b}(x)$ that can be trained to fit a specific dataset. In the case of an autoencoder, we want the hypothesis function to output the input under some constraints $h_{W,b}(x) \approx x$. Although an autoencoder can have multiple layers, for our study, we will consider an autoencoder with one hidden layer. Let layers 1, 2 and 3 be respectively the input, the hidden and the output layers of our autoencoder. Let $a = (a_1, a_2, \ldots, a_H)$ be the vector representing the activation units of the hidden layer where H is the size of the hidden layer. Our autoencoder has parameters $(W, b) = (W^1, b^1, W^2, b^2)$, where W^l and b^l are respectively the matrix of parameters (or weights) and the bias vector associated with the connections between the units of layer l and the units of layer $l + 1$. For a given input vector x, the output of the hypothesis function $h_{W,b}(x)$ is computed as follows:

- First, the values of each activation unit of the hidden layer are calculated using the following formula:

$$a_i = f(\textstyle\sum_{j=1}^{n} W_{ij}^1 x_j + b_i^1)$$

or more compactly

$$a = f(W^1 x + b^1)$$

where f is the exponential linear unit (ELU) function.

- Similarly, the output of the hypothesis function is given by:

$$h_{W,b}(x) = W^2 a + b^2$$

For an autoencoder, we want to have $h_{W,b}(x) \approx x$. To this purpose, a cost function $J(W, b)$ is defined as follows:

$$J(W, b) = \frac{1}{m} \sum_{i=1}^{m} (\frac{1}{2} \left\| h_{W,b}(x^{(i)}) - x^{(i)} \right\|^2)$$

where m is the number of training examples in the dataset and $x^{(i)}$ is the ith training example. The purpose of the training phase is to find out values of W and b that minimize the cost function. In the case of a sparse autoencoder, we need to add an extra sparsity loss term to the cost function. Finally, we end up with the following cost function:

$$J_{SPARSE}(W, b) = J(W, b) + \beta \sum_{j=1}^{H} KL(\rho \parallel \hat{\rho}_j)$$

where β controls the weight of the sparsity loss term, KL is the Kullback-Leibler divergence, ρ is the targeted sparsity (on average the percentage of neurons we want to activate), $\hat{\rho}_j$ is the average activation of the jth unit in the hidden layer given by:

$$\hat{\rho}_j = \sum_{i=1}^{m} a_j(x^{(i)})$$

First, the parameters (W and b) are randomly initialized using the He initialization strategy [Gér19]. Then, the parameters are updated through gradient descent steps so as to minimize the cost function $J_{SPARSE}(W, b)$.

The difference between the input and the output is called the reconstruction error. Let $h_{W,b}(x) = (\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_n)$ be the output of the autoencoder when fed with a specific input $x = (x_1, x_2, \ldots, x_n)$. Then the reconstruction error RE is given by:

$$RE = \sum (\hat{x}_i - x_i)^2$$

During the training phase, the parameters (W, b) of an autoencoder are optimized in order to minimize the reconstruction error for a particular dataset. Once trained, if the autoencoder is fed with data that are similar to the data used during the training phase, the reconstruction error will be small. On the contrary, the reconstruction error will be large when test data are different from the data used during the training phase.

Detection Threshold Determination

The reconstruction error RE can be used as a measure of the outlierness of new samples. To use an autoencoder as an anomaly detector, a detection threshold has to be decided. The detection threshold is the value of the reconstruction error above which an instance is considered as being anomalous.

Figure 3.6: Proposed anomalous communications detection architecture using a set of sparse autoencoders (SAE) when the type of device producing the network traffic is unknown

Due to the base-rate fallacy phenomenon, the false alarm rate is the main limiting factor of any intrusion detection system (IDS) [Axe00]. For this reason, we propose to fix the detection threshold based on the false alarm rate we are ready to accept. It is determined using a validation dataset that is different from the one used for training. Let $FPR_{desired}$ be the false positive rate we are ready to accept. First, the reconstruction errors of the samples in the validation set are calculated. Note that we have no idea on the distribution of the reconstruction errors. Then, the threshold is set so as to have a false positive rate on the validation set FPR_{Val} that is equal to $FPR_{desired}$. The final assessment of the performance of the model is performed on the test set, which is different from the validation set that is used to fix the threshold. Hence, the false positive rate on the test set can be different from FPR_{Val}. Note that the validation data set consists only of legitimate communications data of the specific IoT device for which the autoencoder was trained.

3.2.4 Individual Autoencoders vs Set of Autoencoders

The developed anomalous communications detection model can be used in two different settings, depending on whether or not we know what type of device is producing the ongoing network communications. In the first case, we assume that it is possible to know what type of IoT device is producing the network traffic. This can be done, for example, by filtering the network traffic by MAC addresses if the IDS is located in the same local network as the monitored IoT devices. In this case, it suffices to redirect the network traffic to the appropriate sparse autoencoder (the one trained for that specific device).

However, if the IoT network is connected to the internet through a NAT proxy and the IDS is located somewhere outside the local network, it can be difficult to filter the traffic based on device types. In this second case, we assume that it is not possible to determine the type of the device that produced the network communications. Therefore, the feature vectors describing each communication are fed to all the trained sparse autoencoders as shown in Figure 3.6. A decision module

then takes the outputs of all the sparse autoencoders and determine whether or not a communication is anomalous. A communication is considered as being anomalous only if all the trained sparse autoencoders consider so. Formally, let us consider a total of D different IoT device types. Hence, the feature vectors describing network communications are fed to D different sparse autoencoders. The reconstruction errors on the output of each autoencoder are calculated and compared to their respective detection thresholds to determine whether the communication is anomalous or not. Let n be the number of features used to describe a network communication. $\{A_1, A_2, ..., A_D\}$ is the set of decisions obtained for the D sparse autoencoders, with $A_i : \mathbb{R}^n \mapsto \{0, 1\}$ the decision of the sparse autoencoder trained to learn the legitimate network communication profile of the ith device type. It takes as input a feature vector x from the feature space $\mathbb{R}^n$ and outputs 0 if the corresponding communication is legitimate and 1 if it is anomalous. During testing phase, x is fed to all the trained sparse autoencoders. The final decision *anomaly* is given by:

$$anomaly = \bigcap_{i=1}^{T} A_i(x)$$

where *anomaly* is equal to 1 if the communication is anomalous. Note that the communication is considered to be legitimate if at least one autoencoder considers so.

3.2.5 Dataset Description

To train and test the model, we need three different datasets: a training set, a validation set and a test set. The training set is used to optimize the parameters (W and b described in Section 3.2.3) of the sparse autoencoder. The validation set is used to fine tune the hyperparameters of the model, namely, the learning rate, the number of epochs used for training (early stopping) and the decision threshold (see Section 3.2.3). Both the training and validation sets only contain legitimate network traffic data as they are used to learn the normal communication profile. The test set is used to assess the performance of the developed model and it contains both legitimate and malicious network traffic data.

For the legitimate network traffic data, we reuse the smart home dataset used for network classification (see Section 3.1.3). Table 3.9 shows the total number of bidirectional TCP flows extracted for each device. Note that this table is different from Table 3.2 in that we do not partition the dataset beforehand. Instead, we will use 5-fold cross-validation.

Malicious bidirectional flows used in the test set are obtained from IoTPOT [Pa+15], an IoT honeypot designed to get infected by IoT malware. The collected network traffic data represents network traffic generated by IoT botnets. A total of 46,796 bidirectional TCP flows are extracted from one day of network traffic data. Note that the number of malicious bidirectional flows is much larger than the number of legitimate ones. This is not an issue as the malicious flows are only used during the testing phase to assess the attack detection rate.

Table 3.9: Total number of bidirectional flows per device

	Bidirectional flows
D-Link Motion Sensor	1074
Nest Security Camera	1055
TP-Link Smart Bulb	1040
TP-Link Smart Plug	858
Total	**4027**

3.2.6 Evaluation Methodology

As explained earlier, one sparse autoencoder per device is trained to learn the legitimate communication profile. The architecture of each sparse autoencoder for our experiment is as follows (for details about the parameters, see Section 3.2.3 and [Ng+11; Gér19]):

- the size of the input layer is 16 (equal to the number of features)
- the size of the hidden layer is 32
- the target sparsity (ρ) is equal to 0.1 (default value used in [Gér19])
- the sparsity weight (β) is equal to 0.2 (default value used in [Gér19])
- the validation set is used to select the best learning rate from the following set of values: 0.1, 0.01, 0.001.
- early stopping: training is ended as soon as the validation loss stops improving for more than 10 epochs.

We train and test the model for different values of N (introduced in Section 3.2.2) ranging from 2 to 10. For example, if N is equal to 5, the statistics described in Table 3.8, are calculated over the first 5 packets sent and 5 packets received. To get the best out of the limited number of legitimate communication samples, we perform 5-fold cross-validation. That is, the set of collected legitimate communications is split into 5 folds. The sparse autoencoders are trained using 4 folds (that actually consists of the training set and the validation set) and the remaining fold is used as the test set. The process is repeated 5 times, with each of the 5 folds used exactly once as the test set. The average performance over the 5 test folds gives us the overall performance of the model.

The performance assessment is done in two stages: first, we measure the performance of every single sparse autoencoder separately. Indeed, each autoencoder can be used separately if it is possible to know what type of device is producing the network traffic. Then we measure the overall performance obtained for the set of sparse autoencoders. As explained in Section 3.2.4, when it is not possible to know what device type a network communication belongs to, one can feed it to all the trained autoencoders. The true positive rate (TPR) (also referred to as the recall or the attack detection rate) along with the false positive rate (FPR) are used to

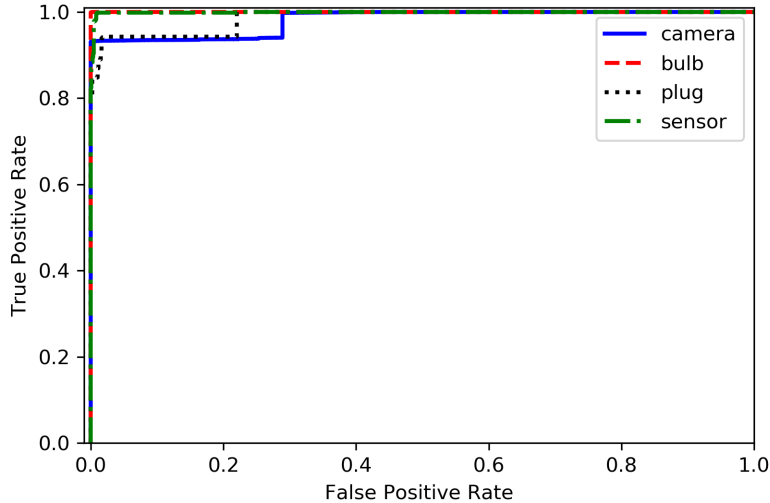


Figure 3.7: ROC curves of the different sparse autoencoders in the case of N=10

assess the performance of the model (see Section 2.1.3). Whether using the individual sparse autoencoders separately or as a set, we show that our proposed model achieves relatively high TPR while keeping the FPR small.

3.2.7 Experimental Results: Performance of Individual Autoencoders

We start by assessing the performance of individual sparse autoencoders, each trained to learn the profile of the legitimate communications of a specific IoT device type. Hence, for our experiment, we have four different sparse autoencoders trained to learn the communication profile of the security camera, the smart bulb, the smart plug, and the motion sensor, respectively. To assess the performance of one specific sparse autoencoder, we fed it with both legitimate and malicious communications and compute the reconstruction error for each communication (legitimate communications consist only of the communications generated by the specific IoT device for which the autoencoder was trained). Then, the reconstruction errors are used to calculate the TPR and FPR for various threshold values. To calculate the Area Under the Curve (AUC), we first need to plot the Receiver Operating Characteristic (ROC) curve. The ROC curve plots the TPR against the FPR at various detection threshold settings as shown in Figure 3.7. We refer the reader to Section 2.1.3 for more details on ROC curves and AUC. Figure 3.8 shows the obtained AUC for different values of N. Depending on the value of N and the specific device for which the sparse autoencoder was trained, the AUC oscillates between 0.962 and 1. This indicates a very high separability between malicious and legitimate communications for all the trained sparse autoencoders and for any value of N.

As described in Section 3.2.3, the detection threshold of each sparse autoencoder is set so that FPR_{Val} is equal to a desired value. Note that the value of the

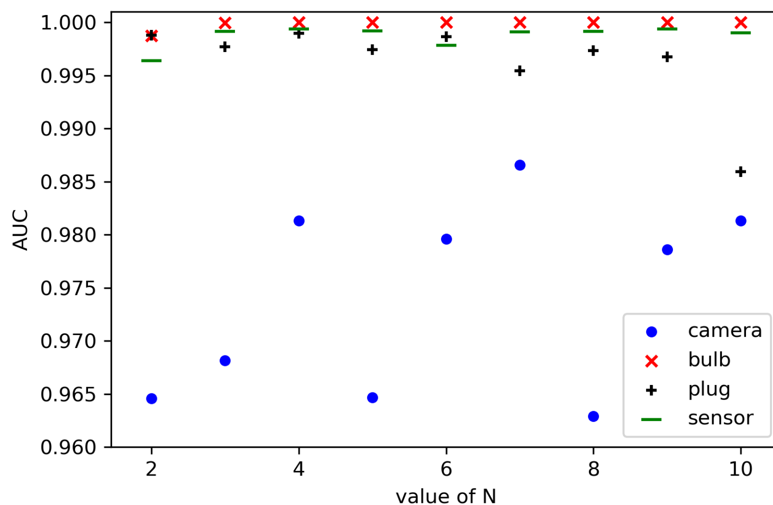
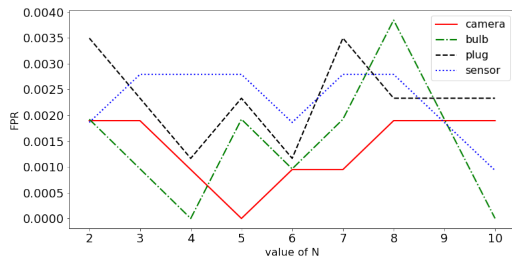


Figure 3.8: AUC of the different sparse autoencoders and for different values of N

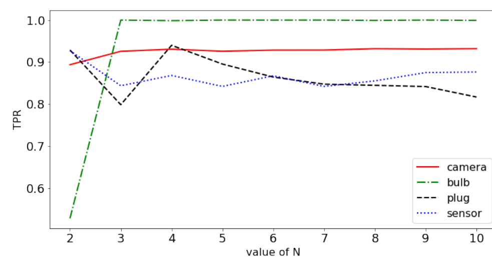
threshold can be different from one autoencoder to another. In our experiments, the threshold is set for the following values of FPR_{Val}: 0.002, 0.001, 0.0005, 0.0002. The performance achieved by the individual sparse autoencoders is shown in Figure 3.10. As expected, the FPR on the test set oscillates around the FPR_{Val} used to determine the threshold. For example, the FPR on the test set oscillates around 0.002 when $FPR_{Val} = 0.002$, 0.001 when $FPR_{Val} = 0.002$ and so on. The TPR achieved by each autoencoder increases with N and plateaus for N greater than 4. That is, looking at more packets than first 4 packets sent and the first 4 packets received no longer improves the TPR. For example, the TPR of the autoencoder trained for the security camera, for the threshold setting $FPR_{Val} = 0.0002$, stays in the range 92.1%-93.1% for N greater than 4. Similarly, the TPR of the autoencoders trained for the bulb, the plug and the sensor, for the same threshold setting, plateaus in the ranges 99.8%-99.9%, 80%-90.6% and 82.5%-86.1%, respectively.

3.2.8 Experimental Results: Performance of the Set of Autoencoders

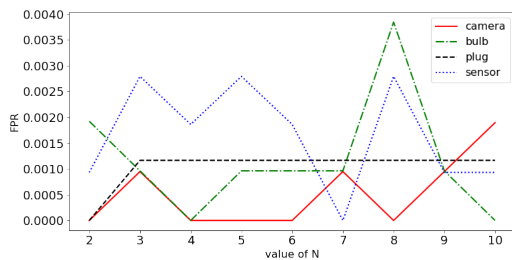
Here, we assess the overall performance of the model. Once the threshold has been determined based on the validation set, the test data is fed to all the sparse autoencoders and the final decision is obtained following the method described in Section 3.2.4. Figure 3.9 shows the achieved FPR and TPR for different values of N and for different threshold settings (corresponding to the different values of FPR_{Val}). The TPR of the set of autoencoders achieves a maximum value ranging from 82% to 87% depending on the value of N and the selected threshold setting. Unsurprisingly, the TPR increases with N and reaches its maximum for N equal to 4. This was expected as the TPR obtained for the individual autoencoder composing the set also plateaus for N greater than 4. The FPR is very close to FPR_{Val} for



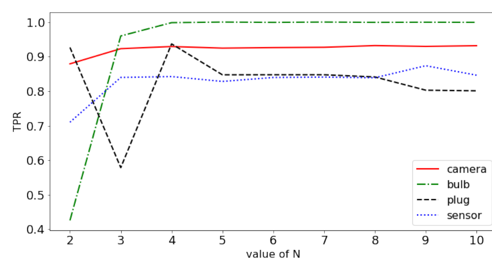
(a) FPR on the test set for $FPR_{Val} = 0.002$



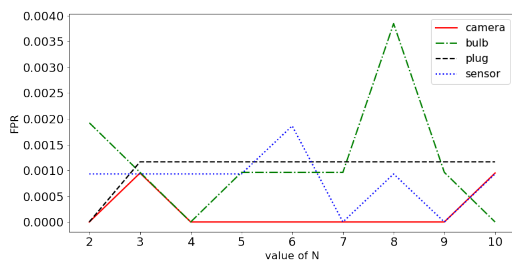
(b) TPR on the test set for $FPR_{Val} = 0.002$



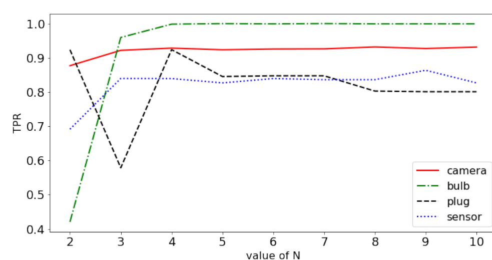
(c) FPR on the test set for $FPR_{Val} = 0.001$



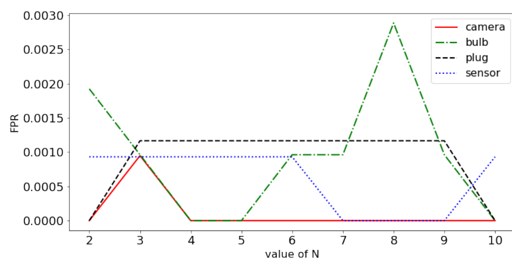
(d) TPR on the test set for $FPR_{Val} = 0.001$



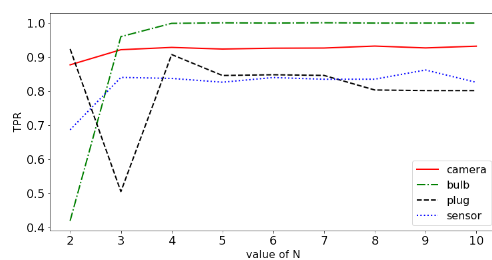
(e) FPR on the test set for $FPR_{Val} = 0.0005$



(f) TPR on the test set for $FPR_{Val} = 0.0005$

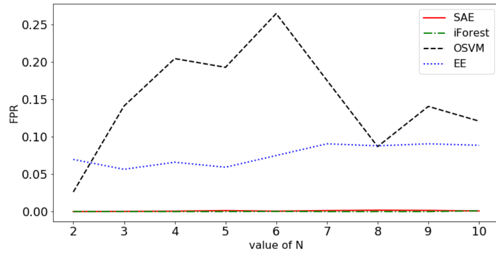


(g) FPR on the test set for $FPR_{Val} = 0.0002$

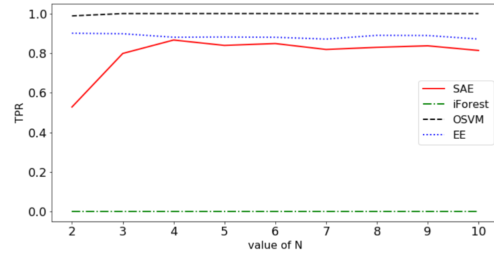


(h) TPR on the test set for $FPR_{Val} = 0.0002$

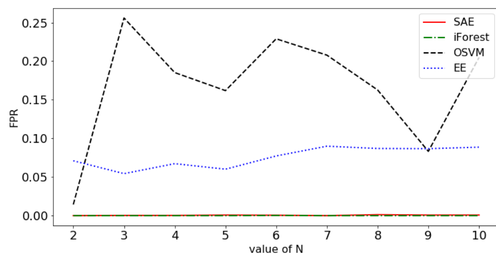
Figure 3.9: False positive rate (FPR) and True Positive Rate (TPR) of the individual sparse autoencoders (each trained for a specific device) for different threshold values.



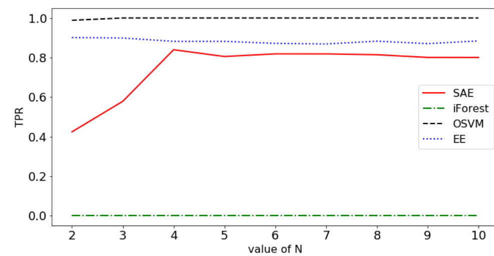
(a) FPR on the test set for $FPR_{Val} = 0.002$



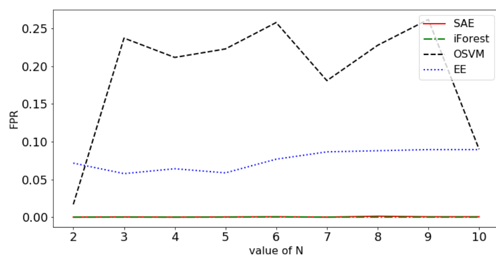
(b) TPR on the test set for $FPR_{Val} = 0.002$



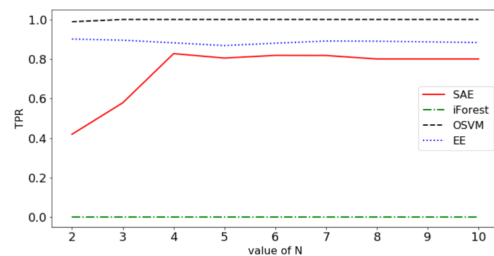
(c) FPR on the test set for $FPR_{Val} = 0.001$



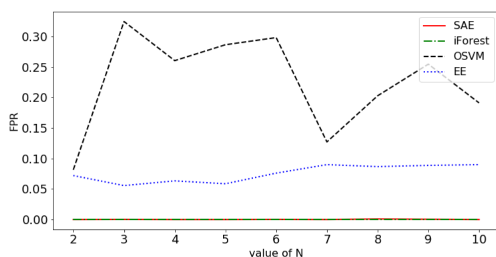
(d) TPR on the test set for $FPR_{Val} = 0.001$



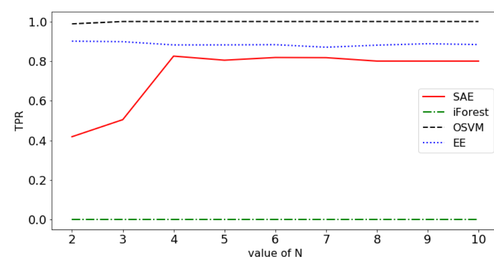
(e) FPR on the test set for $FPR_{Val} = 0.0005$



(f) TPR on the test set for $FPR_{Val} = 0.0005$



(g) FPR on the test set for $FPR_{Val} = 0.0002$



(h) TPR on the test set for $FPR_{Val} = 0.0002$

Figure 3.10: False positive rate (FPR) and True Positive Rate (TPR) of the set of sparse autoencoders (SAE) for different threshold values. The performance of the set of SAE is also compared to other machine learning models.

every threshold setting. For example, when the threshold is selected so as to have FPR_{Val} equal to 0.02%, FPR on the test set oscillates between 0% and 0.05% for the different values of N. The only exceptions occurs when N is equal to 8, in which case the FPR peaks at 0.1%.

The performance of the set of sparse autoencoders is compared to 3 other unsupervised machine learning models, namely, Isolation Forest, One Class SVM and Elliptic Envelope. All 3 algorithms provide a *contamination* parameter that can be used to control the threshold of the underlying decision function. Similarly to the case of the set of sparse autoencoders, one different model is trained for each IoT device type. Then, for testing purpose, the data is fed to all the trained models. One Class SVM and Elliptic Envelope achieve high TPR, ranging from 87% to 100% depending on the value of N and the threshold settings. However, they also achieve a high FPR ranging from 2% to 30%, making them useless in practice. As for the Isolation Forest algorithm, the FPR and TPR are always equal to 0 for every value of N and every tested threshold setting. In fact, the TPR achieves reasonable values, such as 80%, only for thresholds selected so as to have FPR_{Val} greater than 5%. In other words, the TPR for Isolation Forest becomes reasonable only for a FPR higher than 5%.

3.2.9 Discussion

For our model, we trained one separate autoencoder per device type. But another possibility is to train one separate autoencoder for a group of devices sharing similar networking behaviors. For example, if smart devices from a specific manufacturer produce very similar network traffic, then one single autoencoder can be trained for all of them. Another possibility is to first run a clustering algorithm to determine groups of devices sharing similar networking characteristics. Training one autoencoder for a group of devices will reduce the size of the final model making it faster to run and more resource efficient. However, this will reduce the flexibility of the model. For example if the networking behavior of a device in a group changes (because of a system update for example), we might have to redefine all the groups and retrain the autoencoders from scratch using data from all devices.

An important limitation of our work is the limited number of IoT devices used for the experiments. The obtained experimental results are based on a smart home network consisting of only four devices. Although, the ability of one autoencoder to model the legitimate behavior of a single IoT device should remain the same (as long as the IoT device is not versatile), the performance of the set of autoencoders is expected to decrease if the number of different IoT devices in the network increases. Indeed, as described in Section 3.2.4, a communication is considered to be legitimate if at least one autoencoder in the set considers so. As a consequence the attack detection rate will always be less than or equal to the attack detection rate of the worst performing autoencoder in the set.

Another limitation of our method is that a malware may be able to exchange a

few number of packets before being blocked. Indeed, every time a new communication is initiated, the first N packets sent and received are analyzed to determine if the communication is legitimate or not. Only after a communication has been identified as being anomalous, it can be blocked or redirected for further analysis. For example, one can train a model for N equal to 4 (as experimental results show that attack detection rate reaches its maximum for N equal to 4 and no longer improves for higher values of N). Once a communication is detected to be anomalous, only the subsequent packets can be blocked. That is, in our example, a malware may still be able to exchange 3 packets or less before being detected. However, this issue can be solved by combining our model with other approaches, such as, blacklisting IP addresses linked with anomalous activities so that a malware cannot exchange information using multiple short communications. For certain types of attack, such as port scanning, very few packets are exchanged and, by the time it is detected, it might be too late to do anything. One solution is to temporarily block all new communications that are initiated if an anomalous communication is detected. Only the existing legitimate connections are maintained. A notification can be sent to the network administrator in order to alert that a device is behaving suspiciously. In fact, the developed system might be either automated or under human supervision. It might be used as a tool to assist network administrators who can further analyze the detected anomalies to determine whether they correspond to false positives or not.

Due to the base-rate fallacy phenomenon [Axe00], the FPR is the limiting factor of the performance of our anomaly detection model. The four IoT devices of our testbed generated around 4000 bidirectional TCP flows in 7 days. This is equivalent to about 150 bidirectional flows per device and per day. As a consequence, with a FPR of 0.02%, the set of autoencoders produces one false alarm per month for every device. That is, for our testbed of four devices, one false alarm is generated every week. In the case of a smarthome or a smart building with 100 IoT devices, the proposed model will yield 3 false alarms every day, which is significant. Further studies need to be carried out to improve the FPR of our model.

The possibility for IoT devices to get software updates that can change their networking behavior is another constraint that can hinder the performance of our model. This can result in a rise in the number of false alarms. One solution to avoid such an issue is to regularly retrain the models so that the latest changes in the networking behavior are taken into account. For example, if the original models were distributed with the devices, then vendors can distribute retrained models through software updates. However, retraining the models on the whole dataset can be costly in terms of computational resources. Online learning techniques can be explored to develop a more resource efficient model retraining pipeline.

Finally, a malware might be able to fool our model if it is able to mimic the legitimate networking behavior of a device. To this purpose, an attacker might leverage adversarial machine learning. Once, a device is infected, the first step for the malware can be to learn the expected networking behavior of the device using built-

in machine learning algorithms. Then the malware can initiate communications that comply with what is considered to be legitimate. Such a technique can be effective for data exfiltration purposes if sensitive data are accessible through the infected device. However, complying with what is authorized limits the capability of the malware. It makes difficult to perform certain types of attack, such as, port scanning or brute force attacks. Moreover, it might be too costly for a malware to learn patterns as most IoT devices are resource-constrained.

3.2.10 Summary

We introduced a method to detect anomalous communications in IoT networks using a set of sparse autoencoders. The features used to describe bidirectional flows are statistics on the size of the first N packets sent and received, along with statistics on the inter-arrival times between the packets. During the training phase, for each IoT device type present in the network one separate sparse autoencoder is trained to learn the legitimate communication profile. During the testing phase however, it might not be possible to know the type of device that has produced the bidirectional flow. In that case, the bidirectional flow is fed to all the trained autoencoders. The flow is considered to be legitimate if at least one autoencoder considers so. Promising experimental results show that our proposed method can achieve high TPR with a reasonable FPR. The proposed approach also outperforms other anomaly detection algorithms such as one-class SVM, Isolation Forest and Elliptic Envelope.

3.3 General Conclusion

In this chapter we presented two IoT network monitoring solutions that overcome some of the limitations of existing state of the art works. To experimentally assess the performance of the proposed solutions, a smart home network composed of four devices was built.

The first solution is a system that recognizes the type of an IoT device by analyzing its networking behavior. It improves the state of the art by being *delay-free*, *phase-independent* and *non-intrusive*. Features used to describe bidirectional flows are the size of the first N packets sent and received, along with the inter-arrival times between packets. Multiple supervised machine learning algorithms were trained to classify the bidirectional flows based on the device type they belong to. An overall accuracy as high as 99.9% was achieved by the Random Forest classifier, pointing out the effectiveness of our approach. Because of the small size of the dataset, the ANN failed to outperform the Random Forest classifier.

The second solution is an IoT NIDS. It overcomes the limitations of existing works in that it can be used even when it is not possible to determine the type of the device that is generating the network traffic (for example if the NIDS is located outside the local network). Moreover, it is capable of detecting new types of attack as it leverages anomaly detection algorithms. The features used to describe

bidirectional flows are statistics on the size of the first N packets sent and received, along with statistics on the inter-arrival times between the packets. The proposed model consist of a set of sparse autoencoders, each trained to learn the legitimate networking behavior profile for a specific IoT device type. Promising experimental results show that our method can achieve high TPR with a reasonable FPR.

For future works, we suggest to train and test both models on more data coming from networks composed of a greater number of different IoT devices. One should also consider the case of a general purpose network that contains not only IoT devices but also personal computers or smartphones.

Both proposed IoT network monitoring tools achieved promising experimental results. However, one major limitation of the experiments was the limited number of IoT devices used (only 4 devices). Moreover, it is difficult to find publicly available IoT network traffic data. Because of privacy concerns companies or institutions are reluctant to share their data. One solution is to generate synthetic IoT network traffic data. In the next chapter, we propose to leverage recent advances in the field of generative deep learning to generate synthetic IoT network traffic data.

4

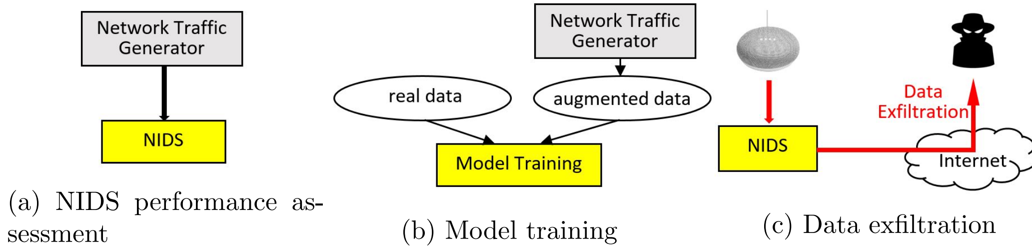
IoT Network Traffic Generation

"Live as if you were to die tomorrow. Learn as if you were to live forever."
– Mahatma Gandhi

Brought into the spotlight by their success in the field of computer vision, GANs are more and more used for network traffic generation purposes (see Section 2.4). Generating real-looking synthetic IoT network traffic is of practical interest for both network defenders as well as attackers. While NIDS developers need network traffic data to assess their developed products, malware developers want to mimic legitimate networking behavior to evade NIDSs.

To protect IoT networks, NIDSs specifically designed for IoT are being developed. To assess the ability of those NIDSs to correctly detect intrusions and avoid false alarms, both legitimate and malicious network traffic data are required. While malicious network traffic is used to evaluate attack detection rate, legitimate network traffic is necessary to assess the FPR. One main difficulty is that very few IoT network traffic datasets are publicly available. One solution is to physically deploy real IoT devices to produce network traffic data. Yet, this can become very costly if one needs a network with thousands of smart devices. Moreover, due to privacy concerns, it might not be possible to share real network traffic data. An alternative is to synthetically generate IoT network traffic. As shown in Figure 4.1.a, the generated network traffic data can be used to assess the performance of NIDSs. Synthetic network traffic generation can also be useful for data augmentation purposes (see Figure 4.1.b). Using data augmentation, a machine learning model can be trained on both real and synthetic network traffic allowing a faster convergence of the model and a better achieved performance.

The capability of generating real-looking network traffic can also be leveraged by



(a) NIDS performance assessment

(b) Model training

(c) Data exfiltration

Figure 4.1: Motivation for IoT network traffic generation: Scenario (a): realistic network traffic generation for NIDS performance evaluation; Scenario (b): data augmentation to train machine learning based NIDS; Scenario (c): mimicry attack generation for data exfiltration purpose.

attackers for malicious purposes like data exfiltration. Indeed, contrary to general-purpose computers, IoT devices perform very specific tasks, hence their networking behavior is very stable and follows specific patterns. Any network communication that does not follow the legitimate behavioral pattern can be easily pinpointed as being anomalous. For example, as illustrated in Figure 4.1.c, the microphone of a compromised Google Home Mini can be used to spy on and listen to conversations. However, the network being protected by an NIDS, the attacker needs to find a way to generate network traffic that looks legitimate in order to bypass detection, and exfiltrate sensitive data. Hence, prior to data exfiltartion, as a first step, an adversary who has the ability to sniff the network, may collect legitimate network traffic; the attacker then proceeds to train generative models from the collected traffic, so as to learn how to generate network traffic resembling the real, legitimate one. In order to deceive an NIDS, the trained model may be used to initiate network communications that mimic the legitimate behavior of the infected device (also known as *mimicry attack*).

Network traffic consists of bidirectional flows identified by source and destination IP addresses and ports, and the transport layer protocol. Bidirectional flows are represented as a sequence of packets sent and received between an internal source and an external destination. A bidirectional flow is also characterized by its duration. Despite the fact that a flow and the packets composing it are closely related (for example, the number of bytes exchanged for the duration of a flow usually amounts to the sum of the sizes of each packet that composes the flow), existing works either focus on the generation of flow-level features [Rin+19; RG18] or packet-level features [Lin+19; Che19], but not both at the same time. Traffic generation based only on flow-level features is of no interest for network monitoring tools that perform packet-level analysis, while traffic generation based only on packet-level features is not adapted for tools that perform flow-level analysis. Another limitation of existing works is that none of them focuses on IoT devices.

We aim at generating synthetic sequences of packet sizes that correspond to bidirectional flows that look like they were generated by a real IoT device. In addition to generating packet-level features which are the sizes of individual packets,

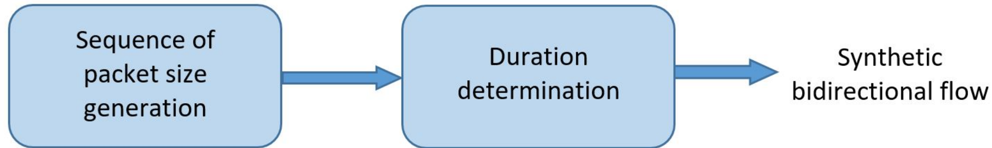


Figure 4.2: Bidirectional flow generation pipeline: the sequence of packet sizes generation module is followed by the duration determination module

our developed generator implicitly learns to comply with flow-level characteristics, such as the total number of packets and bytes in a bidirectional flow. For each generated sequence of packet size, we also determine its total duration. Figure shows the two steps of our proposed bidirectional flow generation process. First, the sequence of packet sizes received and sent is generated. The generated sequence is then fed to another model to determine its duration. The modules for the generation of a sequence of packet sizes on one hand, and the determination of the sequence duration on the other hand are independent from each other, and will be addressed in two different sections (Section 4.1 and Section 4.2 respectively). For the sequence of packet sizes, two types of ordering of the packet sizes sent and received will be explored: a simple naive ordering that assumes that each packet sent is followed by a packet received, and a more realistic ordering where multiple packets can be sent before receiving any reply or vice versa. For the experimental part of our study, we use network traffic data produced by a Google Home Mini, a widely used smart speaker. Comparing the distributions of different network characteristics, such as the number of packets and bytes per flow, shows that the generated bidirectional flows exhibit behavioral patterns that are very similar to the real traffic.

4.1 Generating Sequences of Packet Sizes

4.1.1 Modeling Sequences of Packet Sizes

The packet size is a commonly used feature in studies that focus on the development of IoT network monitoring systems. Hence, it is an important network traffic characteristic for both security product developers and malware authors that want to mimic the legitimate behavior of a device. We intend to generate sequences that represent the size of the individual packets composing a bidirectional flow. Hence, while learning to generate packet-level features which are the sizes of individual packets, our developed model implicitly learn to comply with flow-level characteristics, such as the ordering of the packets or the total number of packets and bytes in a bidirectional flow. As for the ordering of the packet sizes, two different models will be examined: *simplified ordering* and *realistic ordering*.

Simplified Ordering

A bidirectional flow is identified by a 5-tuple, the source and destination IP addresses and ports, and the transport layer protocol. A sequence representing a bidirectional flow contains the sizes of packets sent as well as the sizes of packets received during a single communication. A communication is a complete TCP session (from SYN to FIN). Note that a timeout is used to split lengthy communications into multiple bidirectional flows. Let N be the maximum number of packets that can be sent (or received) within a single bidirectional flow. That is, a bidirectional flow contains a maximum of $2 \times N$ packets (N packets sent and N packets received). We use a separate notation L with $L = 2 \times N$ to denote the length of a sequence. N might be device- or application-specific. We have witnessed that some devices never generate flows with more than a certain number of packets. For experimental purposes, one could set N and truncate sequences that are too long. This might be useful to control the cost of the training process, in terms of resources, which increases as N grows. The simplified ordering model assumes that each packet sent is followed by a packet received. Let S be a sequence of packet sizes corresponding to a bidirectional flow, let s_i be the size of the ith packet sent and r_i the size of the ith packet received. S can therefore be defined as follows:

$$S = \{s_1, r_1, s_2, r_2, ..., s_N, r_N\}$$

The direction of a packet, whether it is a packet sent or received is determined by its position in the sequence: odd positions correspond to sent packets while even positions corresponds to received packets. In fact, the sequence S is composed of two interleaved unidirectional flows: one unidirectional flow for the packets sent and the other for the packets received. The order of the packet sizes for each unidirectional flow is correct. However, the ordering of packet sizes for the bidirectional flow might not be correct, as in real settings one packet sent might trigger the reception of multiple packets (because of network packet fragmentation for example) and vice versa. If a bidirectional flow contains less than N packets sent, then the remaining elements of the sequence are filled with zeros (the same is true for the packets received). Hence, zero acts as an end of sequence marker.

It is important to notice that we consider S to be a sequence of categorical data rather than numerical data. For example, let D be a device which produces sequences of packet sizes (in bytes) of the following form:

$$\{60\ 60\ 52\ 52\ 123\ 135\ 52\ 52\ 52\}$$

Considering the dataset of collected sequences, we may notice that the device D never produces packets of size 61, 53 or 50. If our trained generator generates the following (approximate) sequence:

$$\{60\ 61\ 52\ 53\ 123\ 135\ 52\ 50\ 52\}$$

we may decide that it cannot have been generated by D as it contains packet sizes that D never produced in its history, namely 61, 53, and 50. The approximate

sequence is still very similar to the original one (erroneous packet sizes are close to real ones, if considered as numerical values). To avoid ending up generating such close but unrealistic sequences, one needs to consider the packet size as a categorical variable. Hence, one-hot-encoding over all the possible packet size values is performed to represent each element of S.

Realistic Ordering

In realistic packet ordering, it is possible to send multiple packets before receiving any reply and vice versa. It is called realistic because we have witnessed this type of packet ordering in the real network traffic data produced by our experimental smarthome devices (see Section 3.1.3). Packet sizes are represented using tuples that contain the size of the packet and its direction (whether it is a packet sent or received). Let R be a sequence of packet *(size, direction)* tuples corresponding to a bidirectional flow, let $size_i$ and $direction_i$ be respectively the size and the direction of the ith packet in the sequence. R can therefore be defined as follows:

$$R = \{(size_1, direction_1), (size_2, direction_2), ..., (size_L, direction_L)\}$$

If a bidirectional flow contains less than L packets then the remaining elements of the sequence are filled with a zero-padding tuple that also acts as an end of sequence marker. Using the realistic packet ordering representation, the previous sequence example becomes:

{(60, sent), (60, received), (52, sent), (123, sent), (52, received), (135, received),
(52, sent), (52, sent), (52, received), (52, sent)}

Note that a packet sent is note necessarily followed by a packet received. Again the sequence is a sequence of categorical value. The main difference is that (60, sent) and (60, received) now represent two distinct categories. Hence, the vocabulary size (total number of distinct categories) is greater for the realistic ordering representation. After one-hot encoding, the dimension of the one-hot vectors representing each elements of R is also greater.

An element of the simplified ordering sequence only contains information about the size of a packet. The direction of the packet depends on the position of the element in the sequence. Differently, in the realistic ordering sequence, each element of the sequence contains information on both, the size and direction of a packet. The advantage of the realistic ordering model is that it is more flexible and closer to what real traffic looks like. However, as the number of distinct categories is greater, the dimension of the one-hot encoded representation of each element of the sequence is larger. As the search space for the learning algorithm is larger and complex (the algorithm needs to figure out the relationship between packet sizes and directions by itself), the training process might become more difficult, possibly yielding poor quality results. Generative models will be trained using both the simplified and the realistic representation. The obtained results will then be compared to see if the used representation significantly alters the quality of the generated sequences.

Unless specified otherwise, in the rest of this paper S refers to a sequence of packet sizes under the simplified ordering assumption while R is a sequence of packet *(size, direction)* tuples. Both S and R represent a bidirectional flow. Note that, regardless of using the simplified or realistic representation, our problem is similar to word by word text data generation in that a bidirectional flow is equivalent to a sentence (of length L), while packet sizes (or size/direction tuples), as categorical data, are equivalent to the words that compose the sentence. Hence, to develop our generative model, we will first have a look at the recent developments in the field of natural language processing, specially natural language generation.

4.1.2 Generative Models for Sequences of Categorical Data Generation

On the Difficulty of Generating Sequences of Categorical Data

GANs were introduced by I. Goodfellow et al. [Goo+14] and have been successfully applied in computer vision to generate realistic images [KLA19]. As illustrated in Figure 2.11, a GAN consists of a *generator* and a *discriminator*. The role of the generator is to generate observations as similar as possible to the samples present in a given dataset. To this purpose, the generator transforms random noise into samples that look as if they have been drawn from the original dataset. The role of the discriminator is to predict whether a given sample comes from the original dataset or has been generated by the generator. Both, the discriminator and generator are neural networks. At the beginning of the training process, their weights are randomly initialized. The GAN is trained by alternatively training the generator and the discriminator. The term *adversarial* refers to the fact that the generator and the discriminator are competing with each other. As the generator begins to fool the discriminator, the discriminator must learn new patterns to differentiate real samples from generated ones. In turn, the generator needs to find new ways to fool an ever improving discriminator. This cycle continues up to the point the generator starts generating samples that the discriminator cannot discriminate from real samples anymore. However, GANs are very hard to train and are prone to *mode collapse*. Mode collapse occurs when the generator starts generating one or a small set of possible observations that always fool the discriminator [Fos19]. In that case, the generator stops learning anything useful. It raises the issue of how representative the generated samples are of the diversity of the original dataset. The Wasserstein GAN (WGAN) proposed by M. Arjovsky et al. [ACB17] improves traditional GANs. It provides more stable training process and gets rid of mode collapse issues. This motivates us to use WGAN for our experiments. The loss function of a WGAN is the Wasserstein loss, given by:

$$-\frac{1}{m}\sum_{i=1}^{m} y_i p_i$$

where m is the total number of training instances, y_i and p_i are respectively the label, and the prediction of the *critic* (the discriminator of a WGAN is called the critic),

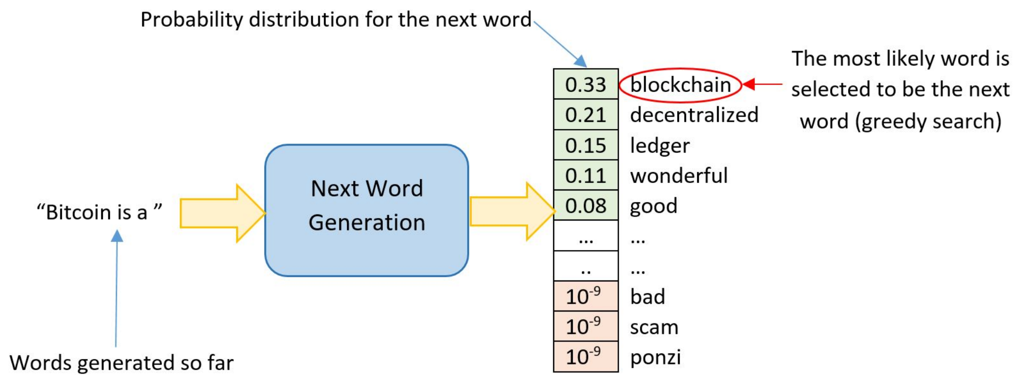


Figure 4.3: Word by word text generation: When generating the next word of the sequence, the generator actually provides a probability distribution over the vocabulary. The actual sequence is constructed by picking the next word from this probability distribution.

corresponding to the ith training instance. The label y_i is either equal to 1 (real) or -1 (generated), and the prediction p_i is in the range $[-\infty, +\infty]$. Hence, by minimizing the loss function, the critic of a WGAN tries to maximize the difference between its predictions for real samples and generated samples. Without any additional constraint, the Wasserstein loss can be very large and become intractable. This is why the critic of a WGAN must be a 1-Lipschitz continuous function. In the original paper, the 1-Lipschitz constraint is enforced by clipping the weights (forcing the weight values to be within a certain range) of the critic. In [Gul+17a], the authors proposed to enforce the 1-Lipschitz constraint by penalizing the norm of the gradient of the critic with respect to its input, which is a more natural way to achieve the 1-Lipschitz constraint. We will test both WGAN with weight clipping (WGAN-C) and WGAN with gradient penalty (WGAN-GP).

Although successful for image generation, GANs have known little success with the generation of sequences of categorical data, until recently. Indeed, as shown in Figure 4.3, when generating the next element of a sequence of categorical values, the generator actually provides a probability distribution over all possibilities (e.g., the vocabulary for text data). The actual sequence is constructed by picking the next element from this probability distribution. This picking operation is not differentiable and thereby hard to back-propagate [KH16; Yu+17]. To overcome this issue, many solutions have been proposed in the context of text generation. Kusner et al. [KH16] propose to use the Gumbel-softmax distribution as the output of the generator. Yu et al. [Yu+17] describe how Reinforcement Learning can be used to bypass the issue. The model we use to generate sequences of packet sizes is highly inspired from the work of D. Donahue et al. [DR18] who propose to combine a vanilla autoencoder with a GAN to generate text data. First, the autoencoder is trained to learn to convert sequences of categorical data (sentences composed of words) into a latent vector in a continuous space. Then a GAN is trained in the continuous latent space to learn to generate latent vectors that can be decoded into sequences

of categorical data. The advantage of this method over the other approaches, is that the GAN can be trained on a lower dimensional latent space making the training process less computationally expensive.

Combining an Autoencoder with a GAN

To generate sequences of packet sizes S and sequences of packet *(size, direction)* tuples R, we propose to combine an autoencoder with a GAN as described in Figure 4.4.

An autoencoder is composed of an encoder and a decoder. The encoder compresses the input to obtain a latent representation of it. The role of the decoder is to reconstruct the original input from its latent representation. Hence, the training process aims at minimizing the reconstruction error between the input and the output. For our sequence of one-hot encoded packet size data, this corresponds to minimizing the cross-entropy loss between the input and the output of the autoencoder. Let L be the total length of a sequence. Let V be the vocabulary size, that is, the number of possible values that an element of a sequence can take: this corresponds to the total number of possible packet sizes for S and the total number of possible packet size and direction combinations for R. Hence, the one-hot encoded representation of a single element of a sequence is a vector of dimension V. The sequence of one-hot encoded vectors can be represented with a matrix X:

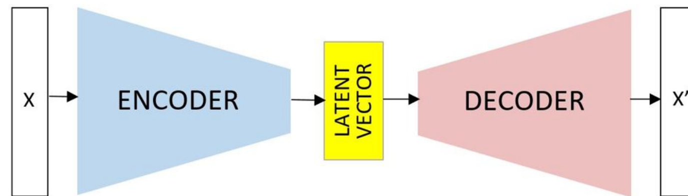
$$X = \begin{bmatrix} x_{11} & x_{12} & ... & x_{1L} \\ x_{21} & x_{22} & ... & x_{2L} \\ ... & ... & & ... \\ x_{V1} & x_{V2} & ... & x_{VL} \end{bmatrix}$$

where the ith column of X corresponds to the one-hot encoded representation of the ith element in the sequence. Let X be the matrix representation of the input of the autoencoder, and $\hat{X}$ be the equivalent matrix representation of the output of the autoencoder. Then, the cross-entropy loss L_{CE} between X and $\hat{X}$ is given by summing the binary cross-entropies between every single element of X and $\hat{X}$:

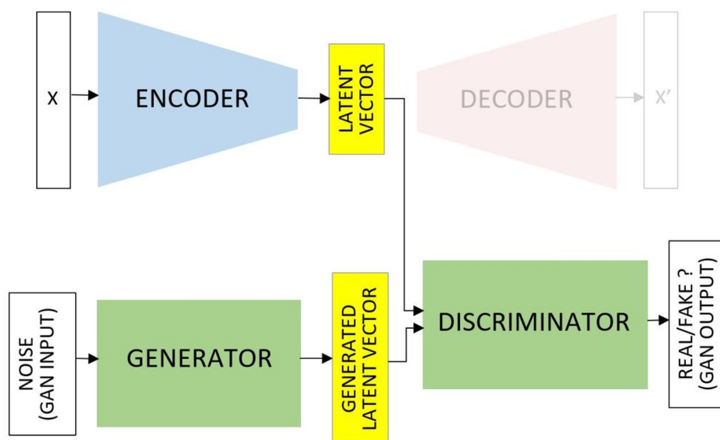
$$L_{CE}(X, \hat{X}) = \sum_{j=1}^{L} \sum_{i=1}^{V} (x_{ij} log(\hat{x}_{ij}) + (1 - x_{ij}) log(1 - \hat{x}_{ij}))$$

Once the autoencoder has been trained, a GAN is trained on the latent space. For our model, we will use a WGAN as it provides more stability in the training stage and is more resilient to mode collapse [ACB17]. Figure 4.4 shows the different steps from the training phase to the generation phase. Step 1: an autoencoder is trained to learn to compress and reconstruct real sequences of categorical values (sizes or sizes/directions). Step 2: the encoder part of the autoencoder is then used to obtain the continuous latent representation of the real sequences. Next, a WGAN is trained on the continuous latent space to learn to generate realistic latent vectors. Step 3: once trained, the generator of the WGAN is used to generate real-looking latent vectors. The generated latent vectors are then fed to the decoder of the autoencoder to generate realistic sequences that correspond to bidirectional flows.

Step 1: Train an autoencoder on real data



Step 2: Train a GAN to learn to generate realistic latent vectors



Step 3: Generation phase

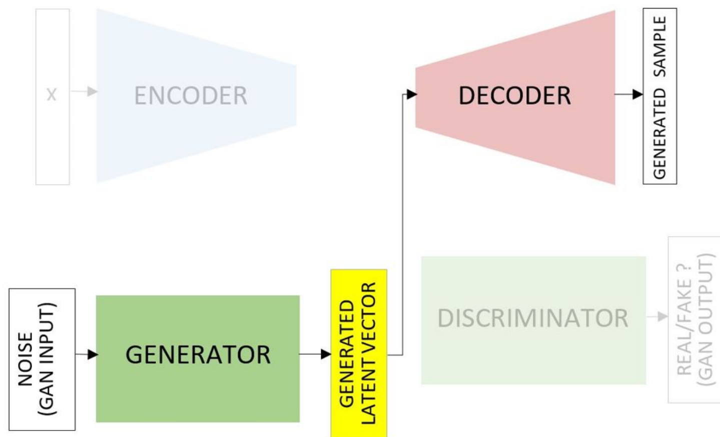


Figure 4.4: Combining an autoencoder with a GAN to generate sequences of categorical values

4.1.3 Evaluation Methodology

Different methods are used to assess the quality of the generated bidirectional flows. First, we measure the percentage of generated sequences that are valid. Then, we compare the distributions of different network characteristics. The purpose of the comparison is to determine if the generated bidirectional flows are diverse enough (no mode collapse) and if they behave like the real ones. We plot histograms to compare the empirical distributions of different network characteristics of the synthetic bidirectional flows with the real ones. The characteristics that are compared are:

- the distribution of the packets sizes;
- the distribution of the number of packets per bidirectional flow;
- the distribution of the number of bytes per bidirectional flow.

Those network characteristics are widely used to describe bidirectional flows [BG15; Rin+19; RG18; Wu+19; Yan+19]. Finally, we also assess the proportion of synthetic bidirectional flows that can evade a potential anomaly detection based NIDS. To this purpose, we train different anomaly detection algorithms to learn the legitimate networking behavior profile of a Google Home Mini. Synthetic bidirectional flows are then fed to the trained anomaly detectors to determine the proportion of synthetic flows that is able to fool the trained detectors into labeling them as legitimate.

4.1.4 Smart Speaker Dataset

A smart speaker, namely the Google Home Mini was used to produce real network traffic data. It allows users to speak voice commands to interact with different services like listening to music, asking for the weather or any other question. For the experiment, the Google Home Mini was mainly used by lab members to ask questions. The device was actively used for 7 days. We set the value of N, described in Section 4.1.1, to 21. That is, we only keep bidirectional flows that contain at most 21 packets sent and 21 packets received, which correspond to sequences of length $L = 42$. As explained in Section 4.1.1, if a bidirectional flow contains less than 21 sent packets, the remaining elements are filled with an end of sequence marker (the same is applied to received packets).

The following is an example of a sequence of packet sizes (after zero padding) S under the simplified ordering assumption, where 6 packets have been sent and 4 packets have been received:

60 60 52 52 123 135 52 52 52 0 52 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0

Below, the same bidirectional flow is represented using a sequence R of *(size, direction)* tuples. (0, 0) is the end of sequence marker and is used for padding:

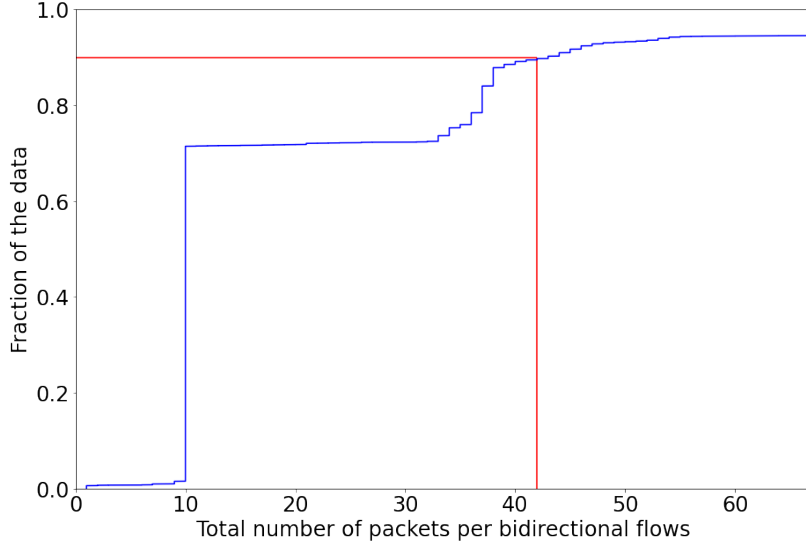


Figure 4.5: Cumulative distribution function of the total number of packets per bidirectional flows produced by the Google Home Mini (a partial view is presented for better clarity). 90% of the flows contain 42 packets or less.

$$\{(60, \text{sent}), (60, \text{received}), (52, \text{sent}), (123, \text{sent}), (52, \text{received}), (135, \text{received}), (52, \text{sent}), (52, \text{sent}), (52, \text{received}), (52, \text{sent}), (0, 0), ...,(0, 0)\}$$

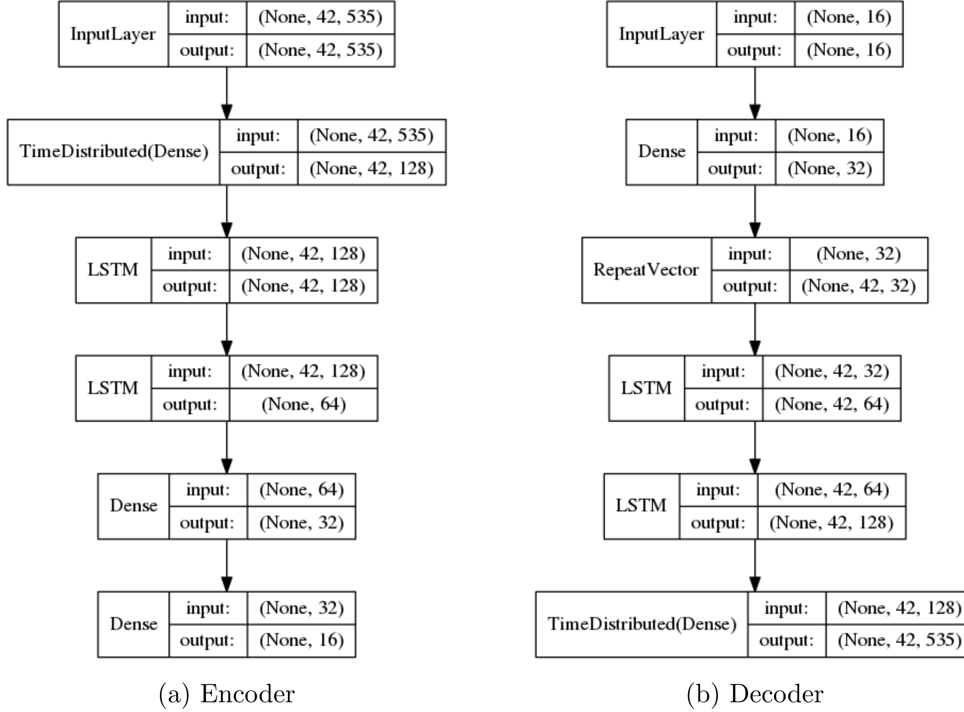
The reason we only keep sequences of maximum length 42 is because they correspond to 90% of the bidirectional flows produced by our hosted Google Home Mini (see Figure 4.5). The remaining 10% of the bidirectional flows contain a number of packets exchanged ranging from 43 to 7674 packets. Meaning that if we were to represent all the bidirectional flows of the Google Home Mini, we would need sequences of length 7674, which can be computationally expensive. For this reason, limiting sequences to 42 packets is reasonable. Moreover, in the case of a malware that want to mimic legitimate behavior, it makes more sense to focus on the 90% most common flows rather than on the 10% rarest flows. The final dataset contains 12,198 sequences of packet sizes representing bidirectional flows. The duration of those 12,198 flows is also collected and will be used later in Section 4.2.

4.1.5 Experimental Results - Simplified Packet Ordering

In this section, we present the results obtained when bidirectional flows are described using the simplified ordering model.

Autoencoder and WGAN Architecture

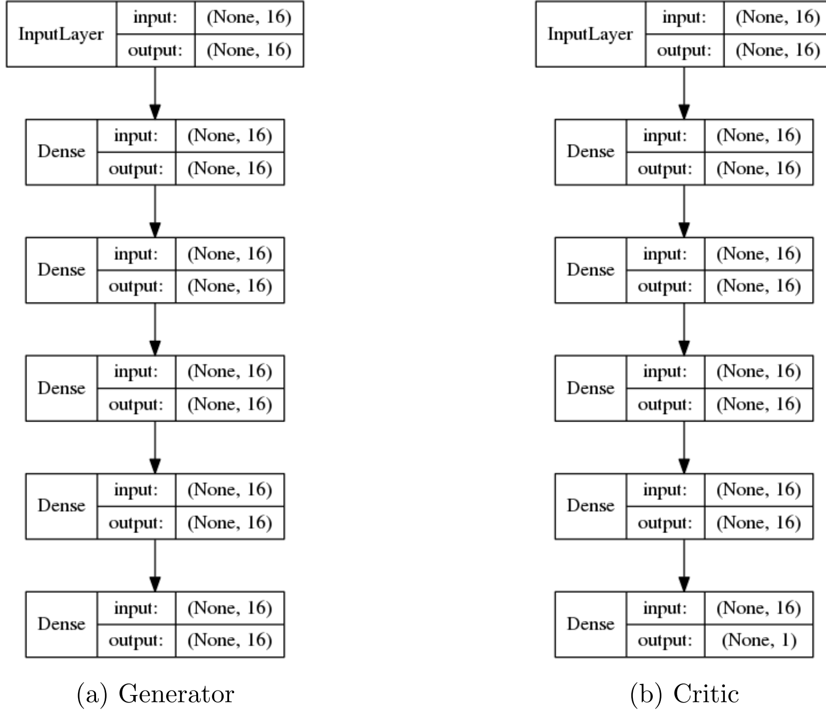
The total number of possible packet size values produced by the Google Home Mini is 535, which corresponds to the vocabulary size under the simplified ordering assumption. Hence, each element of a sequence S is a one-hot encoded vector of dimension 535. Hence, the shape of the matrix X (defined in Section 4.1.2) when describing one sequence of packet sizes is $(V, L) = (535, 42)$.



(a) Encoder (b) Decoder

Figure 4.6: Architecture of the autoencoder used for the experiment

The architecture used for the autoencoder is given in Figure 4.6. The sequences of shape (535, 42) are compressed to 16-dimensional latent vectors by the encoder. The encoder is composed of a time distributed dense layer, followed by two LSTM layers and two densely connected layers. An LSTM network is a type of recurrent neural network that can capture temporal dependencies in sequential data. Hence, LSTM layers are well suited to capture the ordering of the packet sizes in a sequence. The activation function used in all layers of the encoder is the hyperbolic tangent (`tanh`). The decoder consists of a dense layer, followed by two LSTM layers and one time distributed dense layer. The activation function used for all layers is also `tanh`, except for the output layer for which a `sigmoid` activation is used. Note that, for dense layers, to avoid vanishing gradient issues that occur when using `tanh` and `sigmoid` activation functions, Batch Normalization [IS15] is performed before applying the activation function. The loss function used is the cross-entropy loss, as described in Section 4.1.2. The autoencoder is trained for 300 epochs using the `Adam` optimizer. As for the learning rate, if it is too large the weights of the model will converge to a sub-optimal solution or may even diverge, while if it is too small the model will take a significantly longer period of time to converge or may even get stuck in a sub-optimal solution. One strategy is to start with a large learning rate and then gradually decreases it. Hence, to train the autoencoder, the learning rate is set to 0.001 for the first 100 epochs, then to 0.0005 for another 100 epochs, finally to 0.0001 for the last 100 epochs. We stop training at 300 epochs because training the model for more epochs or further decreasing the learning rate no longer improve the training loss.



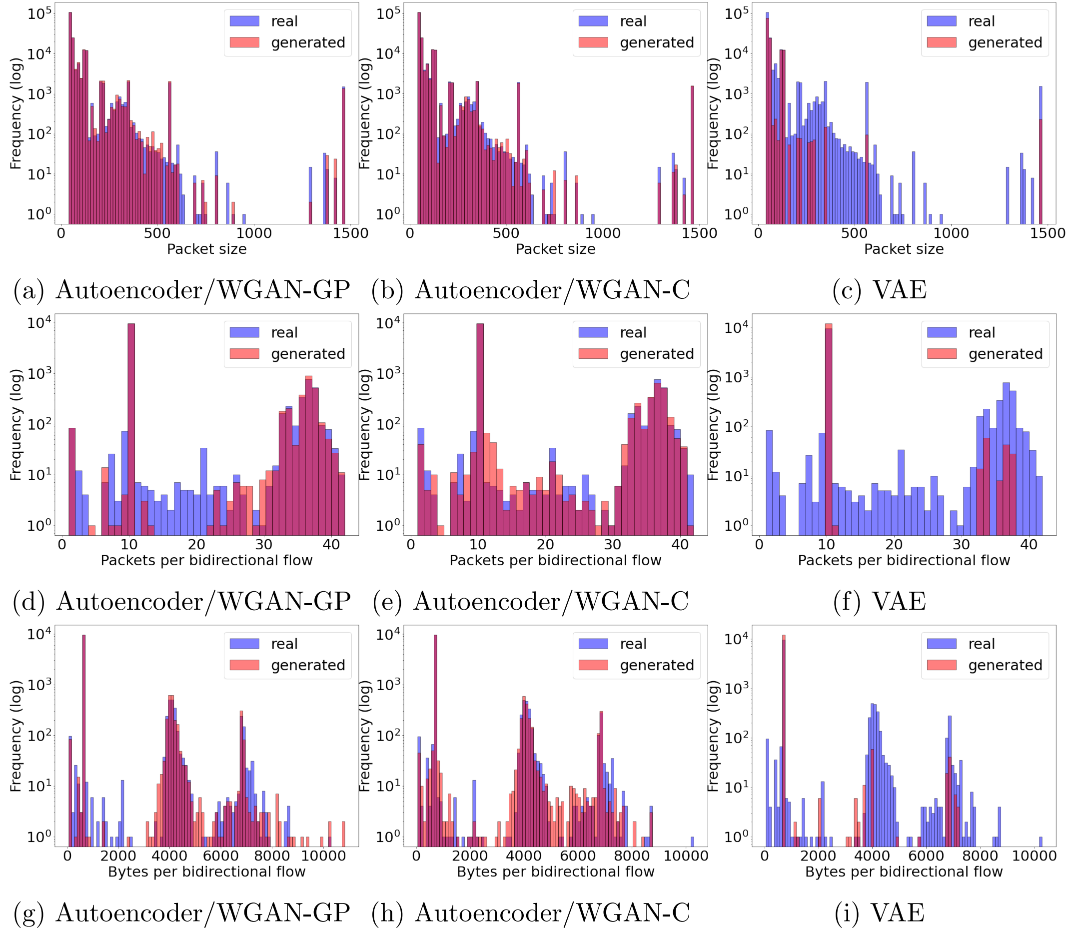
(a) Generator (b) Critic

Figure 4.7: Architecture of the WGAN used for the experiment

As for the WGAN used to learn to generate latent vectors that decode into realistic features vector, both the generator and the critic are densely connected neural networks as shown in Figure 4.7. The generator takes as input a noise of dimension 16 drawn from a standard Gaussian distribution. It consists of 4 hidden layers, each composed of 16 units with `tanh` activation. As with the autoencoder, to avoid vanishing gradient issues, Batch Normalization is performed before applying the `tanh` activation function. The output layer of the generator is a densely connected layer with 16 units and no activation function. The critic consists of 4 hidden layers. Each composed of 16 units with `LeakyReLU` (alpha=0.01) activation. The output of the critic is a single neuron with no activation. As proposed in [ACB17], the `RMSprop` optimizer, with learning rate 0.00005, is used for training. The critic is trained 5 times between each generator updates. Both, WGAN-C and WGAN-GP are tested. As explained in Section 4.1.2, they only differ in the way the 1-Lipschitz constraint of the critic is enforced.

Percentage of Valid Sequences

The generated sequences are of length 42 and consist of the size of the packets sent and received. As described in Section 4.1.1, if a sequence has less than 21 packets sent then the remaining elements must be zeros (because zero act as an end of sequence marker and is used for padding). As a consequence, if the size of a sent packet in the sequence is equal to zero then all the subsequent elements of the sequence that correspond to sent packets must also be equal to zero. The generated sequences that do not comply with this basic rule are considered invalid. The same reasoning holds



(a) Autoencoder/WGAN-GP (b) Autoencoder/WGAN-C (c) VAE

(d) Autoencoder/WGAN-GP (e) Autoencoder/WGAN-C (f) VAE

(g) Autoencoder/WGAN-GP (h) Autoencoder/WGAN-C (i) VAE

Figure 4.8: Simplified packet ordering - Comparison of the distribution of packet sizes (a, b, c), the number of packets per bidirectional flow (d, e, f), the number of bytes per bidirectional flows (g, h, i) for different models (autoencoder/WGAN-GP, autoencoder/WGAN-C, VAE)

for the elements of the sequence that corresponds to the size of packets received. Our trained autoencoder/WGAN-GP and autoencoder/WGAN-C models generate 99.5% and 99.1% of valid sequences respectively. For the evaluation performed in the next subsections, invalid sequences are discarded.

Statistical Characteristics Comparison

The distributions of different network characteristics of the generated and real bidirectional flows are compared for both autoencoder/WGAN-GP and autoencoder/WGAN-C models. The obtained results are also compared to the ones obtained by a baseline variational autoencoder (VAE) model. A VAE is a special type of autoencoder that can be used as a generative model [KW13]. The architecture of the VAE used is very similar to the architecture of the autoencoder described in Figure 4.6. As shown in Figure 2.10, the only difference is that, for the VAE, the encoder outputs two parameters: a mean vector and a variance vector. Those two parameters are used to sample the latent vector. Once trained, to generate new instances, latent vectors are sampled from the standard normal distribution and fed to the decoder of the VAE.

Table 4.1: Simplified packet ordering - Earth mover's distance (10^{-4}) between the real and generated traffic histograms of Figure 4.8. WGAN-C based model achieves the smallest distance.

	WGAN-GP	**WGAN-C**	**VAE**
Packet sizes	1.694	**1.539**	27.439
Packets per bidirectional flows	10.228	**9.799**	97.362
Bytes per bidirectional flows	4.673	**3.837**	40.597

The aim is to determine if combining an autoencoder with a WGAN is necessary, or if a VAE model is enough to generate realistic data.

The 12,198 real bidirectional flows produced by the Google Home Mini along with 12,198 synthetic bidirectional flows generated by our trained WGAN/autoencoders models are used to plot each histogram. Figure 4.8 shows the obtained distributions of packets sizes, number of packets per bidirectional flow and the number of bytes per bidirectional flow. The frequency is represented in a logarithmic scale, as the number of occurrences of rare events is very small compared to the occurrences of common events. The distribution plots indicate that the bidirectional flows generated by both autoencoder/WGAN-GP and autoencoder/WGAN-C models share very close characteristics with the real ones. The similarity is even more emphasized for occurrences that are very common in the real traffic (more than 10 times). For example, packet sizes between 40 bytes and 600 bytes, bidirectional flows containing between 32 and 40 packets, or bidirectional flows containing around 4000 bytes, are very common in the real traffic and are also very common in the generated traffic. Both the WGAN-GP and the WGAN-C based models outperform the VAE which suffers from severe mode collapse. Indeed, the flows generated by the VAE lack diversity and do not cover all the possible flow types that the Google Home Mini produces. In terms of the diversity of the generated bidirectional flows, the WGAN-C based model seems to perform better than the WGAN-GP based model. For example, the WGAN-C based model generates bidirectional flows containing between 16 and 21 packets, which is not the case for the WGAN-GP based model (see Figures 4.8.d and 4.8.e).

We use the Earth Mover's Distance (EMD, also referred as the first Wasserstein distance) to compare the histograms [20g]: informally, if two distributions are seen as two masses of earth, the EMD between those two distributions is proportional to the minimum amount of work required to transform one distribution into the other (one unit of work is the amount of work necessary to move one unit of weight by one unit of distance). The EMD is a statistical distance that provides a measure to quantify the dissimilarity between the histograms of the generated traffic and the histograms of the real traffic. The smaller the EMD between the generated and real traffic histograms the closer the generated traffic is to the real one. As the EMD is used to compare probability distributions, the histograms are normalized

Table 4.2: Simplified packet ordering - TPR and FPR on the test set achieved by the trained anomaly detectors

	OCSVM	**IForest**	**EE**
TPR	.9504	.8947	.9234
FPR	.0238	.0217	.1246

to have a total area equal to 1. Table 4.1 shows the EMD for the histograms in Figure 4.8. The WGAN-C based model achieves the smallest EMD for every compared network characteristics. Hence, it is performing slightly better than the WGAN-GP based model in generating sequences of packet sizes that behaves closely to the real bidirectional flows.

Evading an Anomaly Detection Based NIDS

In this subsection, the aim is to assess how our proposed generative model can be used by a malware to mimic legitimate behavior and evade an anomaly detection based NIDS. In anomaly detection, during the training phase, the model learns the profile of the legitimate networking behavior of a device. Then during the testing phase, the model is applied to new data to detect any deviation from the learnt legitimate behavior profile. We train different anomaly detection algorithms on real Google Home Mini network data to learn the legitimate behavior profile. The trained anomaly detectors are tested against legitimate traffic to assess their False Positive Rate (FPR) and against malicious traffic to assess their True Positive Rate (TPR) (also referred as the recall or the attack detection rate). Malicious network traffic is obtained from IoTPOT [Pa+15], an IoT honeypot designed to be infected by IoT malware. The trained anomaly detectors are also tested against synthetically generated bidirectional flows to evaluate the proportion of synthetic flows that can evade them. We assume that the synthetic flows have been generated by a malware and hence their ground truth label is 'malicious'.

Three anomaly detection algorithms are tested: One-Class SVM (OCSVM), Isolation Forest (IForest), and Elliptic Envelope (EE). The features used as input for the anomaly detectors are the normalized packet sizes. As for the datasets, 80% of the 12,198 real Google Home Mini bidirectional flows are used for training and 20% for testing. 2440 malicious bidirectional flows from IoTPOT are used during the testing phase to assess the TPR of the trained detectors.

Table 4.2 presents the performance on the test set achieved by the different anomaly detectors. In terms of the attack detection rate, OCSVM seems to perform the best with a TPR of 95.04%. While in terms of the FPR, IForest performs the best with an FPR of 2.17%. EE yields the worst FPR (12.46%).

Table 4.3 shows the False Negative Rate (FNR) on the synthetic bidirectional flows, denoted $FNR_{synthetic}$, which corresponds to the proportion of synthetic bidirectional flows that are predicted as being legitimate despite the fact that those flows are not coming from the Google Home Mini but potentially from a malware. It is

Table 4.3: Simplified packet ordering - FNR when the anomaly detectors are fed with synthetic flows ($FNR_{synthetic}$) compared to the FNR and TNR on the test set

	OCSVM	IForest	EE
$FNR_{synthetic}$ (WGAN-GP)	.9817	.9833	.9047
$FNR_{synthetic}$ (WGAN-C)	.9798	.9886	.8948
FNR_{test}	.0496	.1053	.0766
TNR	.9762	.9783	.8754

compared to the FNR and the True Negative Rate (TNR) on the test set containing real bidirectional flows. The FNR on the test set (FNR_{test}) is the proportion of malicious flows that are incorrectly predicted as being legitimate. While the TNR on the test set corresponds to the proportion of bidirectional flows actually coming from the Google Home Mini that are correctly labeled by the anomaly detector as legitimate.

The $FNR_{synthetic}$ is very high compared to the FNR_{test} meaning that if a malware was to use our trained generative model to mimic the legitimate networking behavior, it would considerably improve its evasion capability and the malware would be able to evade the anomaly detectors most of the time. In fact, the $FNR_{synthetic}$ indicates that from 89.48% to 98.86% of the synthetic bidirectional flows (depending on the type of anomaly detector and the WGAN type used for training the generator) are able to fool the anomaly detectors into labeling them as legitimate. While without the use of a generative model, the FNR_{test} indicates that only 4.96% to 10.53% of the malicious flows are incorrectly predicted as being legitimate. The $FNR_{synthetic}$ is also slightly higher than the TNR in most of the cases, indicating that a synthetic bidirectional flow is more likely to be labeled as legitimate by the anomaly detectors than a real flow actually coming from a Google Home Mini. This can be explained by the presence of bidirectional flows coming from the Google Home Mini that are very rare and end up being wrongly labeled as being malicious (rare instances that appear in the test set but were not seen during anomaly detector training). While the generative model tends to generate the rarest bidirectional flow types less often and give priority to the frequent ones.

4.1.6 Experimental Results - Realistic Packet Ordering

In this subsection, we present the results obtained when bidirectional flows are described using the realistic ordering model.

Autoencoder and WGAN Architecture

The total number of possible unique *(size, direction)* tuples produced by the Google Home Mini is 635, which corresponds to the vocabulary size. Note that this value is slightly higher than 535, the total number of unique packet size values. Which

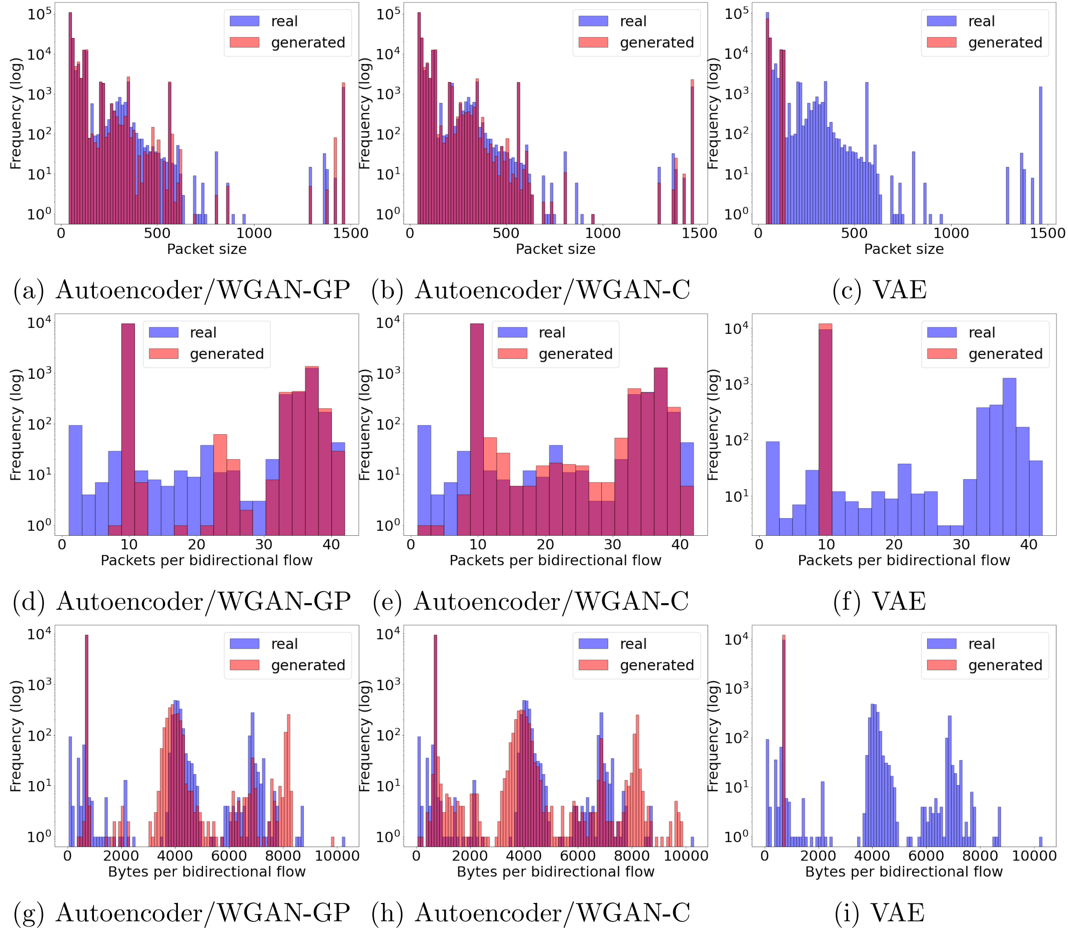
means that most packet size values are associated with only one packet direction (sent or received). Each element of a sequence R is a one-hot encoded vector of dimension 635. Hence, the shape of the matrix X (defined in Section 4.1.2) when describing one sequence of packet sizes is $(V, L) = (635, 42)$. The architectures of the autoencoder and the WGAN are the same as the ones described in Figure 4.6 and 4.7, respectively. Only the dimensions of the input and output of the autoencoder are adapted to the new vocabulary size.

Percentage of Valid Sequences

The generated sequences are of length 42 and consist of *(size, direction)* tuples. As described in Section 4.1.1, if a sequence contains less than 42 packets then the sequence is padded using a specific token that also acts as an end of sequence marker. As a consequence, if an element of the sequence is equal to the padding token then all the subsequent elements of the sequence must also be equal to that token. The generated sequences that do not comply with this basic rule are considered invalid. Our trained autoencoder/WGAN-GP and autoencoder/WGAN-C models generate 99.0% and 99.2% of valid sequences respectively. For the evaluation performed in the next subsections, invalid sequences are discarded.

Statistical Characteristics Comparison

Similarly to Section 4.1.5, the distributions of different network characteristics (packet sizes, packets per bidirectional flows, bytes per bidirectional flows) of the generated and real bidirectional flows are compared for both autoencoder/WGAN-GP and autoencoder/WGAN-C models (see Figure 4.9). We refer the reader to Section 4.1.5 for details about the methodology. The obtained results are also compared to the ones obtained by a baseline variational autoencoder (VAE) model. Table 4.4 presents the EMD used to measure the dissimilarity between the different distributions. The values in brackets represent the EMD obtained for the model trained under the simplified ordering assumption. Both the WGAN-GP and the WGAN-C based models outperform the VAE which suffers from severe mode collapse. In terms of the diversity of the generated bidirectional flows, the WGAN-C based model seems to perform better than the WGAN-GP based model as it achieves a smaller EMD for every compared network characteristics. Overall the performance achieved by models trained using sequences of *size, direction* tuples are not as good as the ones obtained under the simplified ordering assumption. Indeed, for all compared network characteristics, the EMD is smaller under the simplified ordering assumption. Implying that the dissimilarity (as measured by the EMD) between the distributions of real and generated traffic is smaller under the simplified ordering assumption. Whether training a WGAN-GP or WGAN-C based model, the results obtained using simplified ordering assumption are always better than the ones obtained using sequences of *size, direction* tuples.



(a) Autoencoder/WGAN-GP (b) Autoencoder/WGAN-C (c) VAE

(d) Autoencoder/WGAN-GP (e) Autoencoder/WGAN-C (f) VAE

(g) Autoencoder/WGAN-GP (h) Autoencoder/WGAN-C (i) VAE

Figure 4.9: Realistic packet ordering - Comparison of the distribution of packet sizes (a, b, c), the number of packets per bidirectional flow (d, e, f), the number of bytes per bidirectional flows (g, h, i) for different models (autoencoder/WGAN-GP, autoencoder/WGAN-C, VAE)

Evading an Anomaly Detection Based NIDS

In this subsection, we assess how our proposed generative model (generating sequences of *(size, direction)* tuples representing bidirectional flows) can be used by a malware to mimic legitimate behavior and evade an anomaly detection based NIDS. For details about the tested anomaly detection algorithms (OCSVM, IForest, EE), the features and the composition of the training and testing datasets, we refer the reader to Section 4.1.5.

Table 4.5 presents the performance on the test set achieved by the different anomaly detectors. In terms of the attack detection rate, IForest seems to perform the best with a TPR of 88.41%. While in terms of the FPR, OCSVM performs the best with an FPR of 7.88%. EE yields the worst FPR (19.69%).

Table 4.3 shows the FNR on the synthetic bidirectional flows, denoted $FNR_{synthetic}$. It is compared to the FNR (FNR_{test}) and the TNR on the test set containing real bidirectional flows. We refer the reader to Section 4.1.5 for more details about $FNR_{synthetic}$, FNR_{test} and TNR.

The $FNR_{synthetic}$ is greater than the FNR_{test}, implying that if a malware was

Table 4.4: Realistic packet ordering - Earth mover's distance (10^{-4}) between the real and generated traffic histograms of Figure 4.9 WGAN-C based model achieves the smallest distance. The EMD obtained under the simplified ordering assumption are shown in brackets.

	WGAN-GP	**WGAN-C**	**VAE**
Packet sizes	3.822 (1.694)	**2.899 (1.539)**	29.635 (27.439)
Packets per bidirectional flows	15.417 (10.228)	**14.086 (9.799)**	201.667 (97.362)
Bytes per bidirectional flows	6.738 (4.673)	**6.672 (3.837)**	43.451 (40.597)

Table 4.5: Realistic packet ordering - TPR and FPR on the test set achieved by the trained anomaly detectors

	OCSVM	**IForest**	**EE**
TPR	.6416	.8841	.6502
FPR	.0788	.1069	.1969

to use our trained generative model to mimic the legitimate networking behavior, it would improve its evasion capability and would be able to evade the anomaly detectors most of the time. More precisely, the $FNR_{synthetic}$ indicates that from 78.91% to 80.91% of the synthetic bidirectional flows are able to fool the anomaly detectors into labeling them as legitimate. While without the use of a generative model, the FNR_{test} indicates that 11.59% to 35.84% of the malicious flows are incorrectly predicted as being legitimate. Note however that contrary to the results obtained when generating sequences under the simplified ordering assumption (see Section 4.1.5), the $FNR_{synthetic}$ is slightly smaller than the TNR. Meaning that a synthetic bidirectional flow is still slightly more likely to be labeled as malicious by the anomaly detectors than a real flow actually coming from a Google Home Mini.

4.1.7 Discussion

Mimicking the legitimate networking behavior of a device is interesting for data exfiltration purposes. For example, one can imagine a malware intended for Google Home Minis that uses the microphone of the compromised device to listen to con-

Table 4.6: Realistic packet ordering - FNR when the anomaly detectors are fed with synthetic flows ($FNR_{synthetic}$) compared to the FNR and TNR on the test set

	OCSVM	**IForest**	**EE**
$\boldsymbol{FNR_{synthetic}}$ **(WGAN-GP)**	.8091	.8091	.8091
$\boldsymbol{FNR_{synthetic}}$ **(WGAN-C)**	.7891	.7891	.7891
$\boldsymbol{FNR_{test}}$	.3584	.1159	.3498
TNR	.9212	.8931	.8031

versations and exfiltrate the data. However for other types of malware, such as botnets used to perform large-scale DDoS attacks, complying with the legitimate networking behavior might be too much of a constraint. In an attempt to comply with the legitimate behavior, the malware might end up losing its malicious capability altogether. Further studies need to be carried out on how to find a balance between complying with legitimate behavior and not loosing malicious capabilities.

The types of sequences of packet sizes generated by the trained generator are very dependent on the data used during training. The generator will only be able to generate sequences similar to the sequences it has seen during the training phase. For our experimental setup, we interacted with the Google Home Mini primarily to ask simple questions like "what's the weather today?" or "what's the news today?". Hence, our trained generator will generate sequences of packets that are representative of our interactions with the Google Home Mini like asking questions. Now, If someone makes a very different use of the Google Home Mini, like asking it to play music via a Spotify account, then the Google Home Mini might produce very different types of packet sequences. For the generator to be able to generate these new types of sequences, it will need to be retrained on the new data.

4.1.8 Summary

In this Section, we presented a method to generate sequences of packet sizes or sequences of *(size, direction)* tuples, representing bidirectional flows that look as if they were generated by a real smart device. Sequences of packet sizes (or *(size, direction)* tuples) are sequences of categorical data. To overcome the issue with the use of GANs for the generation of sequences of categorical data, we decided to combine an autoencoder with a WGAN. First, the autoencoder is trained to learn to convert sequences of categorical data into a latent vector in a continuous space. Then a WGAN is trained on the latent space to learn to generate latent vectors that can be decoded into realistic sequences, through the decoder of the autoencoder. Experimental results using a Google Home Mini show that our method allows us to generate high quality and realistic looking sequences of packet sizes representing bidirectional flows. The next step is to determine the duration of the generated bidirectional flows.

4.2 Determining the Duration of a Generated Bidirectional Flow

Once a sequence of packet sizes corresponding to a bidirectional flow has been generated, the next step is to determine its duration. In this Section, we present a method leveraging Mixture Density Networks to determine the duration of a bidirectional flow represented by a sequence of packet sizes (simplified ordering) or by a sequence of *(size, direction)* tuples (realistic ordering).

4.2.1 Duration as a Random Variable

The duration of a bidirectional flow is the time length the flow lasts from beginning to end. For example, in the case of a bidirectional flow that represents a TCP connection, the duration is the time that elapses from the establishment of the connection (the moment the client sends a SYN to the server to initiate the connection) to its termination (when a FIN packet is transmitted). Note that the duration is not deterministic but rather a random variable. The same bidirectional flow, identified with the same source and destination addresses/ports, the same protocol, and consisting of the same sequence of packets, can take a wide range of different duration values. Network congestion is one of the reason why the duration of identical bidirectional flows can differ. Let take the example of F_1, a very specific bidirectional flow often produced by the Google Home Mini. F_1 is the following sequence of packet sizes:

$$F_1 = \{60\ 60\ 52\ 52\ 123\ 135\ 52\ 52\ 52\}$$

Figure 4.10 shows the distribution of the durations for F_1. The duration of F_1 is not always the same. It can vary from 0.011s to 0.6s. It is often equal to a value around 0.015s. Now let us imagine that F_1 is a bidirectional flow generated by one of our generator trained in Section 4.1. How do we determine the duration of the generated sequence of packets F_1? Our aim is to train a model that takes a synthetic bidirectional flow (sequence of packet sizes or sequence of *(size, direction)* tuples) as input and determines its duration. If we were to train a deterministic machine learning model, it would take F_1 as input and would always output the most likely duration value (0.015s in our case). However, when generating synthetic flows, it is not realistic for a specific bidirectional flow to always have the same duration. Therefore, we need a machine learning model that takes F_1 as input and outputs $p(duration|F_1)$, the entire probability distribution of all possible duration values for F_1 (and not only the duration value with the highest likelihood). Then, to determine the duration of F_1, we can just sample a value from $p(duration|F_1)$. This way, the duration of synthetic bidirectional flows will behave closely to the duration of real bidirectional flows (taking a wide range of possible values instead of always taking one specific values). Because duration is a noisy variable that can vary greatly, it needs to be modeled with probabilistic deep learning algorithms that output probability distributions. Let F_i be a bidirectional flow generated by one of our trained model from Section 4.1. To determine $p(duration|F_i)$ (the distribution of the durations of F_i), we will train Mixture Density Networks (MDN), a particular type of neural network that output probability distributions.

4.2.2 Mixture Density Networks

A Mixture Density Networks (MDN) is obtained by combining a deep neural network with a mixture of distributions [Bis06]. An MDN is a neural network that provides the parameters of multiple distributions that are then combined using some weights.

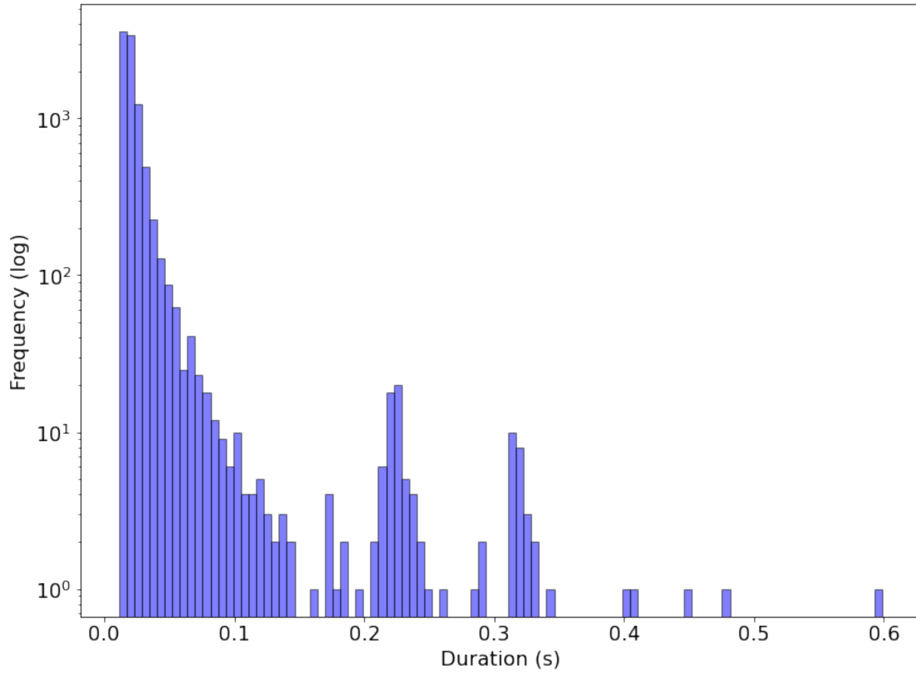


Figure 4.10: Duration distribution for the bidirectional flow F_1

The weights are also provided by the neural network. In this subsection, we describe how MDNs work using specific mathematical notations.

Given a sufficient number of components, a Gaussian mixture is capable of modelling any arbitrary probability density. The conditional probability density $p(t|x)$ of a Gaussian mixture is given by:

$$p(t|x) = \sum_{c=1}^{C} \alpha_c(x)\mathcal{N}(t|\mu_c(x), \sigma_c^2(x))$$

where:

- c denotes the index of the corresponding mixture component. C is the total number of components and is an hyperparameter of the model.
- $\mathcal{N}$ denotes a Gaussian distribution.

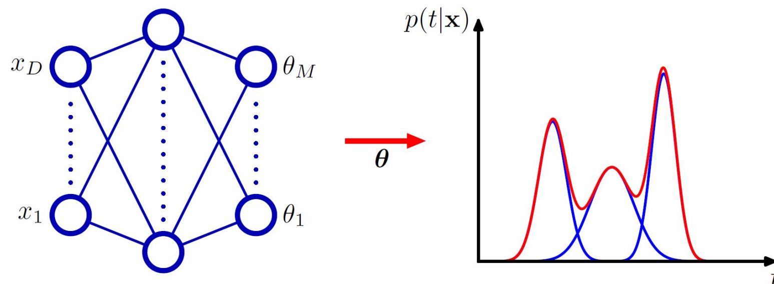


Figure 4.11: The mixture density network can represent general conditional probability densities $p(t|x)$ by considering a parametric mixture model for the distribution of t whose parameters are determined by the outputs of a neural network that takes x as its input vector [Bis06]

- $\mu_c(x)$ is the mean of the Gaussian distribution representing the cth component of the mixture. it is conditioned on the input x.
- $\sigma_c(x)$ is the standard deviation of the Gaussian distribution representing the cth component of the mixture. it is conditioned on the input x.
- $\alpha_c(x)$ is the mixing coefficient. It corresponds to the weight of the cth component of the mixture. it is conditioned on the input x.

An MDN is a neural network that takes x as input and provides the parameters $\alpha_c(x)$, $\mu_c(x)$, $\sigma_c(x)$ of the Gaussian mixture model as output. Hence, its total number of outputs is equal to $3 \times C$. The mixing coefficients must be positive and sum to one: $\sum_{c=1}^{C} \alpha_c(x) = 1$. This ensures that the conditional probability density $p(t|x)$ integrates to one. In practice, a softmax function is used over the mixing coefficients to ensure that they respect the constraints. As for the standard deviations $\sigma_c(x)$, they must be positive. To this purpose, a variant of the Exponential Linear Unit (ELU) activation function with a unit offset can be used. Let W be the vector of weights and biases of the MDN. The MDN is trained by minimizing the negative logarithm of the likelihood given by:

$$E(W) = -\sum_{i=1}^{m} ln \left\{ \sum_{c=1}^{C} \alpha_c(x_i, W) \mathcal{N}(t|\mu_c(x_i, W), \sigma_c^2(x_i, W)) \right\}$$

where m is the number of training instances.

Note that any neural network architecture (CNN, RNN, etc.) can be extended to become an MDN. For our duration determination problem, the input x is a bidirectional flow F_i, represented either by a sequence of packet sizes (simplified ordering) or a sequence of *(size, direction)* tuples (realistic ordering). The MDN outputs the parameters $\alpha_c(x)$, $\mu_c(x)$, $\sigma_c(x)$ of the Gaussian mixture model that represents the conditional probability density function $p(duration|F_i)$. With $p(duration|F_i)$ in hands, it is possible to sample a duration value for F_i.

4.2.3 Evaluation Methodology

To assess the quality of the generated duration values, we compare the distribution of the duration of synthetic bidirectional flows, as determined by the MDN, with the distribution of the duration of real bidirectional flows. First, synthetic sequences of packet sizes are generated using the generators trained in Section 4.1. The generators used are the ones trained using Autoencoder/WGAN-C model, as they yielded the best results (see Sections 4.1.5 and 4.1.6). Then, the generated packet size sequences are fed to the trained MDN to determine their duration. More precisely, the MDN will output a distribution of possible duration values from which a single value will be sampled. Hence, a generated synthetic bidirectional flow consist of a sequence of packet sizes along with a duration value. The distribution of the duration of the synthetic bidirectional flows is compared to the distribution of the duration of real bidirectional flows in two different ways:

- Comparison of the overall duration distribution (obtained from the whole set of generated and real bidirectional flow).
- Comparison of the duration distribution for a specific bidirectional flow, namely the bidirectional flow F_1 (see Section 4.2.1 for more details about F_1)

For the latter comparison, we fed the MDN with the sequence F_1 to obtain a duration distribution which is compared to the real duration distribution for F_1.

4.2.4 Duration Dataset

To train the MDNs we use the same Google Home Mini dataset described in Section 4.1.4. Indeed, for each bidirectional flow produced by the Google Home Mini we not only collected the sequence of packet sizes but also its corresponding duration. The MDNs take as input the sequences of packet sizes and is trained to predict the duration values.

4.2.5 Experimental Results

In this subsection, we first describe the architecture of the MDN used. Then, we compare the distribution of the duration of the synthetic bidirectional flows, as determined by the MDN, with the distribution of the duration of real bidirectional flows.

MDN Architecture

Whether it is a sequence of packet sizes or a sequence of *(size, direction)* tuples, the input of the MDN is a sequence of categorical data. Therefore, the MDN is composed of two parts (see Figure 4.12): the first part transforms the sequence of categorical data into a dense vector, the second part computes the parameters $\alpha_c(x)$, $\mu_c(x)$, $\sigma_c(x)$ of the conditional probability density $p(duration|F_i)$.

For the first part of the MDN, we reuse the encoder of the autoencoder trained in Section 4.1. All the layers of the encoder are frozen (the weights are made non-trainable), so that Gradient Descent will not modify them. Such technique of reusing pretrained layers as part of a model is also known as *transfer learning*. This reduces the computational cost of the training process as we no longer have to train the first layers of the MDN. It also helps the model to better generalize (avoid overfitting), as the pretrained encoder was already trained to extract a compressed meaningful representation of the sequences of packet sizes. The second part of the MDN, is just a Feedforward Neural Network (FNN) composed of 4 densely connected hidden layers, each with 100 neurons and using ReLU activation functions. The conditional probability density $p(duration|F_i)$ is a mixture of 10 Gaussian distributions ($C = 10$). Hence, the size of the output of the MDN is $3 \times 10 = 30$.

Two separate MDNs are trained: one that takes sequences of packet sizes as input and another that takes sequence of *(size, direction)* tuples as input. The MDNs are trained (more precisely only the weights of the FNN part are trainable) using real

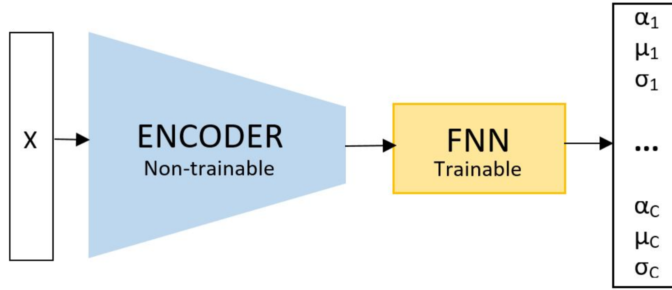


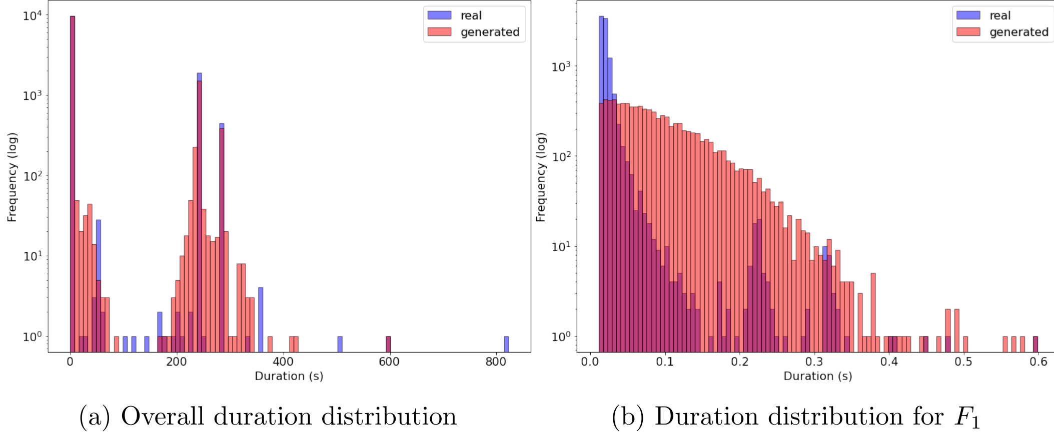
Figure 4.12: MDN architecture

bidirectional flows produced by the Google Home Mini. The duration values are standardized so that they have a zero mean and unit variance. During the training phase, part of the dataset is used as a validation set to determine the appropriate number of epochs. Once the best number of epochs has been determined, the MDNs are retrained on the whole dataset.

Comparison of Synthetic Duration Values with Real Duration Values

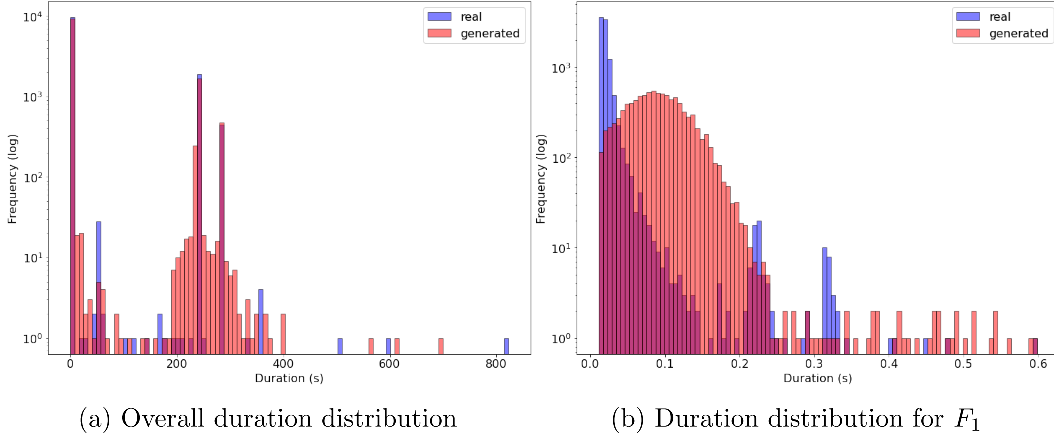
Once trained, the MDNs are used to determine the duration of synthetic bidirectional flows. The MDNs take as input synthetic bidirectional flows and output for each of them a probability distribution of possible duration values. Then, for each flow, a duration value is sampled. Note that the sampled duration value is constrained to vary between the minimum duration value $duration_min_{train}$ and the maximum duration value $duration_max_{train}$ observed in the training set. If the sampled duration value is less than $duration_min_{train}$, it is rounded up to be equal to $duration_min_{train}$. While if it is greater than $duration_max_{train}$, it is rounded down to $duration_max_{train}$. We first compare the overall duration distribution of the synthetic bidirectional flows (as predicted by the MDNs) with the overall duration distribution of real bidirectional flows (Figure 4.13.a and 4.14.a). Then, we compare the duration distribution predicted by the MDNs for bidirectional flow F_1 with the real duration distribution associated with F_1 (Figure 4.13.b and 4.14.b). Figure 4.13 shows the results obtained when bidirectional flows are described using sequences of packet sizes, while Figure 4.14 shows the results obtained when bidirectional flows are described using sequences of *(size, direction)* tuples.

The overall duration of generated bidirectional flows is close to the duration of real ones. Both distributions present peaks at 0.001s, 240s and 280s. However, the distribution of generated durations is smoother than the distribution of real durations. This fact is even more visible when looking at the generated and the real duration distributions for F_1. Both, generated and real duration distributions for F_1 are similar in that both are skewed right. However, the real duration distribution for F_1 is multimodal (with 3 modes), while the generated duration distribution is unimodal. Moreover, in the case of the MDN trained with sequences of *(size, direction)* tuples, the generated duration distribution for F_1 is shifted to the right: the



(a) Overall duration distribution (b) Duration distribution for F_1

Figure 4.13: Simplified packet ordering - Comparison of real duration values with durations generated by the MDN trained using sequences of packet sizes



(a) Overall duration distribution (b) Duration distribution for F_1

Figure 4.14: Realistic packet ordering - Comparison of real duration values with durations generated by the MDN trained using sequences of *(size, direction)* tuples

real duration distribution peaks at 0.015s while the generated duration distribution peaks at 0.1s. Note that overall the duration of real bidirectional flows range from 0.001s to 814s, while the duration of F_1 varies from 0.011s to 0.6s. In fact, by looking at the duration distribution for F_1, we are zooming in to look at a specific part (ranging from 0.011s to 0.6s) of the overall duration distribution. Comparison of the duration distributions for F_1 confirms that the generated duration distribution is smoother than the real one: the 3 peaks that arises in the real duration distribution are smoothed out to form only one mode in the generated duration distribution.

4.2.6 Summary

We presented a method to determine the duration of the bidirectional flows generated in Section 4.1. The duration is a noisy variable that can vary greatly because of external factors, such as network congestion. Therefore, it needs to be modeled with probabilistic deep learning algorithms such as MDN. An MDN takes a bidirectional flow as input (represented by a sequence of packet sizes for example) and outputs a probability distribution of possible duration values. Then, a dura-

tion value is sampled from that distribution and assigned to the bidirectional flow. Experimental results show that MDNs are capable to grasp the overall duration distribution of real bidirectional flows. However, a closer look to specific bidirectional flows shows that the duration distribution predicted by MDNs are too smooth and fail to model multiple modes. Further works are required to improve the ability of the described method to model multimodal duration distributions associated with specific bidirectional flows .

4.3 General Conclusion

In this chapter, we presented a method to generate realistic IoT network traffic data. Our proposed method improves the state of the art in that it allows the generation of both packet-level and flow-level data at the same time. Packet-level features are the sizes of the individual packets in a bidirectional flow, while flow-level features are the total number of packets and bytes per flow, the packet sizes ordering and the duration. We leveraged generative deep learning models to generate ordered sequences of packet sizes representing bidirectional flows. More precisely, to generate sequences of categorical data (packet size values or *(size, direction)* tuples), we combined an autoencoder with a WGAN. Next, MDNs a type of neural networks that output probability distributions helped us to determine the corresponding duration of the generated bidirectional flows. Experimental results using Google Home Mini data shows that our developed approach achieves promising results.

For future works, we suggest to train and test our model on more data and with a variety of smart devices. We also suggest to include other packet-level features such as the status of the TCP flags or the TTL value of each packet in the sequence. The inter-arrival times between packets in the sequence can also help to have a more fine-grained definition of the temporal aspect of the flows.

5

Conclusion

"There is no good in anything until it is finished."

– Genghis Khan

The rapid development of the IoT raises security concerns as most smart devices are vulnerable. IoT devices have become low hanging fruits for botnet creators. Moreover, the constantly evolving malware landscape makes it challenging to secure IoT networks. Because IoT devices are task-specific, their networking behaviors follow a stable and predictable pattern, making them well suited for data analysis techniques. Advances in the field of machine learning, especially deep learning can be leveraged to develop IoT network monitoring tools. However, deep learning algorithms training often require huge amount of data that might not be easily available. In this thesis, we attempted to answer the following two questions:

- How can deep learning help to monitor IoT networks?
- how can deep learning help to overcome the lack of IoT network traffic data?

The first part of this thesis focused on developing two types of IoT network monitoring tools: IoT device type recognition systems and IoT NIDS.

Existing works on IoT device type recognition through network traffic analysis are limited in that they do not satisfy the following requirements: be *delay-free* (do not require the user to wait for long periods of time), be *phase-independent* (do not focus only on a specific phase of a device life cycle), and be *non-intrusive* (do not require to look at application level data and thus allow full data encryption). We proposed a machine learning based IoT device type recognition system that is *delay-free*, *phase-independent* and *non-intrusive*. To this purpose, we first defined an appropriate set of features to describe bidirectional flows. It consists of the size of the

first N packets sent and received along with the corresponding IAT between packets. We performed data visualization using t-SNE to point out the effectiveness of our selected set of features in distinguishing between the bidirectional flows produced by the different IoT devices. We trained and tested different machine learning algorithms to classify the bidirectional flows based on the IoT device they belong to. An overall accuracy as high as 99.9% was achieved by the Random Forest classifier. Because of the small size of the dataset the ANN did not outperform the Random Forest classifier.

Existing works on IoT NIDSs are limited either because they can detect only certain types of attacks or because they require prior knowledge about the type of the device that is producing the network traffic (and hence cannot be deployed outside the local network). To overcome those limitations, we proposed a method to detect anomalous communications in IoT networks based on a set of sparse autoencoders. Features used to describe bidirectional flows are statistics on the size of the first N packets sent and received, along with statistics on the IAT between packets. During the training phase, separate autoencoders are trained to learn the legitimate communication profile of the IoT devices present in the network, an autoencoder for each type. During the testing phase, if for a bidirectional flow it is possible to know the type of the device it belongs to, then it is redirected to the autoencoder trained for that particular device type. However, if it is not possible to know the type of device the bidirectional flow belongs to, then it is fed to all the trained autoencoders. The flow is considered to be legitimate if at least one autoencoder considers so. Promising experimental results show that our method can achieve high TPR with a reasonable FPR. For example, a TPR of 82% is achieved for a FPR of 0.02%. The set of autoencoders also outperforms other anomaly detection algorithms such as one-class SVM or Isolation Forest.

The second part of this thesis focused on leveraging recent advances in the field of generative deep learning to generate real-looking IoT network traffic data. Although a flow and the packets composing it are closely related (for example the number of bytes exchanged for the duration of a flow amounts to the sum of the sizes of each packet that composes the flow), existing works on network traffic generation, either focus on flow-level features or packet-level features generation, but not both at the same time. Moreover, none of the existing works uses IoT network data. We proposed to generate sequences of packet sizes representing bidirectional flows. Hence, while generating packet-level features which are the sizes of individual packets, our developed generator implicitly learn to comply with flow-level characteristics, such as the ordering of the packets or the total number of packets and bytes in a bidirectional flow. Sequences of packet sizes were modeled as being sequences of categorical data. Hence, to overcome the issue with the use of GANs for the generation of sequences of categorical data, we decided to combine an autoencoder with a WGAN. First, the autoencoder was trained to learn to convert sequences of categorical data into a latent vector in a continuous space. Then a WGAN was trained on the latent space to learn to generate latent vectors that can be decoded

into realistic sequences, through the decoder of the autoencoder. For each generated sequence of packet sizes, we also determined its total duration. The duration being a noisy variable, its value was determined using MDNs, a type of neural networks that output probability distributions allowing to model uncertainty. Experimental results using data collected from a Google Home Mini shows that our developed approach achieves promising results.

Perspectives for Future Work

In this thesis, we leveraged deep learning to develop IoT network monitoring tools such as IoT device type recognition system and IoT NIDS. The proposed models achieved promising results. However, one of the major limitation of the study is the small size of the dataset used. The experimental smarthome used to produce the dataset contained only four IoT devices. For future work, we suggest to test the models proposed in this thesis on larger IoT network traffic datasets containing more devices. One should also consider how the models can be adapted to general purpose networks containing not only IoT devices but also personal computers or smartphones. In this thesis, We also proposed to take advantage of generative deep learning to produce synthetic IoT network traffic data. We primarily focused on packets sizes and duration of a bidirectional flow. But those are not the only features characterizing network traffic data. Variables like the TTL value or the state of the TCP flags of a network packet should also be considered to generate even more fine-grained network traffic data. In addition to the aforementioned propositions, for future work we also suggest the following complementary research perspectives:

- Develop flexible, efficient and scalable features extractor. In this thesis, we focused on developing IoT network monitoring applications without wondering how the features fed to the models are extracted from raw network traffic in an operational setting. As shown in Figure 1.4, a critical part of the whole system is the features extractor. For large networks, it might have a lot of data to preprocess and hence must be scalable. Network traffic must be preprocessed efficiently to avoid introducing extra latency. Currently there is no tool that can easily be integrated in an operational networking environment in a scalable manner and without inducing significant latency. Developing a general features extraction framework for network traffic data will allow to easily deploy machine learning based network monitoring systems in real world settings. Such frameworks should give the possibility to select features from a large number of choices ranging from packet-level features like specific fields of IP headers (size, TTL, fragmentation flags, etc.) to flow-level features like the number of bytes per flow or the duration of a flow. The possibilities offered by Software Defined Networking (SDN) and protocols like OpenFlow might be leveraged to develop such feature extraction framework. Programming the data plane of network switches using languages such as P4 might even allow to extract more customized and fine-grained features.

- Leverage machine learning to develop Host based Intrusion Detection Systems (HIDS) or anti-viruses for the IoT. The trained models must be able to run on resource-constrained IoT devices. To this purpose, different deep learning model compression techniques might be explored, such as pruning, quantization, knowledge distillation[20a]. Pruning involves removing connections between neurons by zeroing out values in the weights matrix. Quantization consists of decreasing the size of the weights by using smaller bit-widths: for example, by mapping weights represented using 32 bits floats to 8-bits integers. Knowledge distillation is the process of transferring the knowledge from a large trained model to a smaller model by training it to mimic the larger model's output. Compressing techniques will dramatically reduce the number of parameters of deep learning models making them suitable to run on resource-constrained environment. Of course, the main challenge is to find the best trade-off between compressing the model and keeping its performance reasonable.

- Explore how Graph Neural Networks (GNN)[Zho+18a] can be used to secure IoT networks. GNNs are a special type of neural network architectures designed to work on graphs. A graph is a set of nodes and a set of edges that connect related node together. A graph is said to be directed if edges have orientations. A GNN takes as input a graph and is mainly used for two tasks: node classification and graph classification. In node classification, the GNN is used to determine the label of each node composing the input graph. In graph classification, the GNN outputs a single label for the whole graph. Graphs are well suited to describe IoT networks: devices can be represented by nodes while interdependencies between them can be represented by edges. For example, if a vocal assistant can be used to switch on/off a bulb then the nodes representing those two devices are connected. Note that the relationship is directed, the vocal assistant can change the state of the bulb but the opposite is not necessarily true. If the networks contains hundreds or even thousands of interdependent devices, the obtained graph might be very complex. A node classification problem might be to determine if a device (or node) is *compromised* or *safe*. For example, assume that we already know the label of a small number of nodes (semi-supervised setting), then a GNN can be trained to find out the labels of the remaining nodes. A graph classification problem might be to determine if an IoT network architecture is secure or not. In this case, the GNN will take different graphs representing different IoT network architectures and will output a score determining their security level.

- Develop *interpretable* network monitoring models. In this thesis, we developed machine learning based network monitoring tools that were used as black boxes. Although, they achieved promising performances, they can be improved. With a black box model, once a prediction has been made, it is not possible to determine which input features have contributed the most to

the final decision. In technical terms, our models are not *interpretable*. *Interpretability* refers to the ability to determine the cause-and-effect relationship of a machine learning model. It reinforces trust in the model. It is also helpful for debugging purposes to detect any bias in machine learning models. Knowing what input features are responsible for the model's final decision can also help network administrators to discard false positives. Developing interpretable neural networks is a hot topic and is seen as the future of deep learning. We suggest for future work, to explore how existing works to make models interpretable, such as the use of saliency maps [BF20] and methods like DeepLIFT [SGK17] or LIME [RSG16], can be adapted to deep learning based network monitoring tools. Recently brought into the spotlight thanks to the Transformer [Vas+17] architecture in the field of natural language processing, attention-based neural networks also seem to be an interesting option to achieve interpretability [SS19; WP19].

- Define a set of comprehensive metrics that allow to assess the quality of generated network traffic. At present, each work uses its own metrics to assess the realism of the generated network traffic data, making comparison between works very difficult. In fact, it is worth asking the broader question of what *realism* is when generating synthetic network traffic. Given the complex nature of network traffic data, one needs metrics to assess the realism at different levels of granularity from packet-level to flow-level: what is a realistic network packet? What is a realistic network flow? One might even needs to describe realism beyond the only networking considerations by taking into account human activities. For example, the network traffic in a real smart home might follow a precise temporal pattern with peak of activities early in the morning (switching on lights, asking the vocal assistant about the weather, using a smart coffee machine, etc) and less activities during sleeping time. Applied deep learning being an empirical subject, defining proper metrics is crucial to its success. Researchers in the networking community must come with a set of metrics to assess the realism of synthetic network traffic that they all agree upon.

List of Publications

M. R. Shahid, G. Blanc, H. Jmila, Z. Zhang and H. Debar, ”Generative Deep Learning for Internet of Things Network Traffic Generation,” *2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC)*, Perth, Australia, 2020, pp. 70-79, doi: 10.1109/PRDC50213.2020.00018.

M. R. Shahid, G. Blanc, Z. Zhang and H. Debar, ”Anomalous Communications Detection in IoT Networks Using Sparse Autoencoders,” *2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)*, Cambridge, MA, USA, 2019, pp. 1-5, doi: 10.1109/NCA.2019.8935007.

M. R. Shahid, G. Blanc, Z. Zhang and H. Debar, ”IoT Devices Recognition Through Network Traffic Analysis,” *2018 IEEE International Conference on Big Data (Big Data)*, Seattle, WA, USA, 2018, pp. 5187-5192, doi: 10.1109/BigData.2018.8622243.

Bibliography

[20a] *An Overview of Model Compression Techniques for Deep Learning in Space*. 2020 (accessed December 19, 2020). URL: https://medium.com/gsi-technology/an-overview-of-model-compression-techniques-for-deep-learning-in-space-3fd8d4ce84e5.

[20b] *Flu project*. 2017 (accessed July 6, 2020). URL: https://github.com/fluproject/flu.

[20c] *Future-proof IoT with RIOT-fp*. 2020 (accessed December 4, 2020). URL: https://future-proof-iot.github.io/RIOT-fp//.

[20d] *Network Information Management and Security Group (NIMS)*. (accessed July 6, 2020). URL: https://projects.cs.dal.ca/projectx/Download.html.

[20e] *NSL-KDD dataset*. 2009 (accessed June 30, 2020). URL: https://www.unb.ca/cic/datasets/nsl.html.

[20f] *The CAIDA DDoS Attack 2007 Dataset*. 2007 (accessed July 6, 2020). URL: https://www.caida.org/data/passive/ddos-20070804_dataset.xml.

[20g] *The Earth Mover's Distance (EMD)*. 2020 (accessed June 24, 2020). URL: http://infolab.stanford.edu/pub/cstr/reports/cs/tr/99/1620/CS-TR-99-1620.ch4.pdf.

[20h] *Virustotal*. 2020 (accessed June 29, 2020). URL: https://www.virustotal.com/gui/home/upload.

[98] *World cup 1998 dataset*. 1998. URL: https://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[Aca+18] Abbas Acar et al. "Peek-a-Boo: I see your smart home activities, even encrypted!" In: *arXiv preprint arXiv:1808.02741* (2018).

[ACB17] Martin Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein gan". In: *arXiv preprint arXiv:1701.07875* (2017).

[Ala+17] Fadele Ayotunde Alaba et al. "Internet of Things security: A survey". In: *Journal of Network and Computer Applications* 88 (2017), pp. 10–28.

[Alp20] Ethem Alpaydin. *Introduction to machine learning.* MIT press, 2020.

[Alt20] Palo Alto. *2020 Unit 42 IoT Threat Report.* 2020 (accessed June 24, 2020). URL: https://start.paloaltonetworks.com/unit-42-iot-threat-report.

[Ang17] Kishore Angrishi. "Turning internet of things (iot) into internet of vulnerabilities (iov): Iot botnets". In: *arXiv preprint arXiv:1702.03681* (2017).

[Ant+17] Manos Antonakakis et al. "Understanding the mirai botnet". In: *26th {USENIX} security symposium ({USENIX} Security 17).* 2017, pp. 1093–1110.

[ANT19] Nesrine Ammar, Ludovic Noirie, and Sébastien Tixeuil. "Network-protocol-based IoT device identification". In: *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC).* IEEE. 2019, pp. 204–209.

[ARF17] Noah Apthorpe, Dillon Reisman, and Nick Feamster. "A smart home is no castle: Privacy vulnerabilities of encrypted iot traffic". In: *arXiv preprint arXiv:1705.06805* (2017).

[Aum+10] Jean-Philippe Aumasson et al. "Quark: A lightweight hash". In: *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer. 2010, pp. 1–15.

[Ava20] Avast. *Let's play Hide 'N Seek with a botnet.* 2018 (accessed June 25, 2020). URL: https://blog.avast.com/hide-n-seek-botnet-continues.

[Axe00] Stefan Axelsson. "The base-rate fallacy and the difficulty of intrusion detection". In: *ACM Transactions on Information and System Security (TISSEC)* 3.3 (2000), pp. 186–205.

[Bai+18] Lei Bai et al. "Automatic device classification from network traffic streams of internet of things". In: *2018 IEEE 43rd Conference on Local Computer Networks (LCN).* IEEE. 2018, pp. 1–9.

[Bek+15] Dmitri Bekerman et al. "Unknown malware detection using network traffic classification". In: *2015 IEEE Conference on Communications and Network Security (CNS).* IEEE. 2015, pp. 134–142.

[Bez+18] Bruhadeshwar Bezawada et al. "Iotsense: Behavioral fingerprinting of iot devices". In: *arXiv preprint arXiv:1804.03852* (2018).

[BF20] Jasmijn Bastings and Katja Filippova. “The elephant in the interpretability room: Why use attention as explanation when we have saliency methods?” In: *arXiv preprint arXiv:2010.05607* (2020).

[BG15] Anna L Buczak and Erhan Guven. “A survey of data mining and machine learning methods for cyber security intrusion detection”. In: *IEEE Communications surveys & tutorials* 18.2 (2015), pp. 1153–1176.

[BI17] Elisa Bertino and Nayeem Islam. “Botnets and internet of things security”. In: *Computer* 50.2 (2017), pp. 76–79.

[Bis06] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[Bit20] Bitdefender. *New Hide ‘N Seek IoT Botnet using custom-built Peer-to-Peer communication spotted in the wild*. 2018 (accessed June 25, 2020). URL: https://labs.bitdefender.com/2018/01/new-hide-n-seek-iot-botnet-using-custom-built-peer-to-peer-communication-spotted-in-the-wild/.

[Bog+07] Andrey Bogdanov et al. “PRESENT: An ultra-lightweight block cipher”. In: *International workshop on cryptographic hardware and embedded systems*. Springer. 2007, pp. 450–466.

[BW20] Mario Ballano Barcena and Candid Wueest. *Symantec security response: insecurity in the internet of things*. 2015 (accessed June 24, 2020). URL: https://www.researchgate.net/profile/Hadeel_Saleh_Haj_Aliwi/post/What_are_the_best_papers_in_IoT_Security/attachment/59dda4b44cde260ad3cea425/AS:5481386438853312@1507697844002/download/paper1.pdf.

[Cap+20] Davide Caputo et al. “Are you (Google) Home? Detecting Users’ Presence through Traffic Analysis of Smart Speakers.” In: *ITASEC*. 2020, pp. 105–118.

[CG16] Tianqi Chen and Carlos Guestrin. “Xgboost: A scalable tree boosting system”. In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016, pp. 785–794.

[CGP18] Kim-Kwang Raymond Choo, Stefanos Gritzalis, and Jong Hyuk Park. “Cryptographic solutions for industrial Internet-of-Things: Research challenges and opportunities”. In: *IEEE Transactions on Industrial Informatics* 14.8 (2018), pp. 3567–3569.

[Cha+19a] Nadia Chaabouni et al. “Network intrusion detection for IoT security based on learning techniques”. In: *IEEE Communications Surveys & Tutorials* 21.3 (2019), pp. 2671–2701.

[Cha+19b] Jeremy Charlier et al. “SynGAN: Towards Generating Synthetic Network Attacks using GANs”. In: *arXiv preprint arXiv:1908.09899* (2019).

[Che19] Adriel Cheng. “PAC-GAN: Packet Generation of Network Traffic using Generative Adversarial Networks”. In: *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE. 2019, pp. 0728–0734.

[Cho+14] Kyunghyun Cho et al. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: *arXiv preprint arXiv:1406.1078* (2014).

[Chr+18] Maximilian Christ et al. “Time series feature extraction on basis of scalable hypothesis tests (tsfresh–a python package)”. In: *Neurocomputing* 307 (2018), pp. 72–77.

[Cop+16] Bogdan Copos et al. “Is anybody home? Inferring activity from smart home network traffic”. In: *2016 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2016, pp. 245–251.

[Coz+18] Emanuele Cozzi et al. “Understanding linux malware”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 161–175.

[CZ18] Andrei Costin and Jonas Zaddach. “Iot malware: Comprehensive survey, analysis framework and case studies”. In: *BlackHat USA* (2018).

[DAF18] Rohan Doshi, Noah Apthorpe, and Nick Feamster. “Machine learning ddos detection for consumer internet of things devices”. In: *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2018, pp. 29–35.

[DC18] Abebe Abeshu Diro and Naveen Chilamkurti. “Distributed attack detection scheme using deep learning approach for Internet of Things”. In: *Future Generation Computer Systems* 82 (2018), pp. 761–768.

[DR18] David Donahue and Anna Rumshisky. “Adversarial text generation without reinforcement learning”. In: *arXiv preprint arXiv:1810.06640* (2018).

[EP16] Sam Edwards and Ioannis Profetis. “Hajime: Analysis of a decentralized internet worm for IoT devices”. In: *Rapidity Networks* 16 (2016).

[Fos19] David Foster. *Generative deep learning: teaching machines to paint, write, compose, and play*. O’Reilly Media, 2019.

[Fra17] Chollet Francois. *Deep learning with Python*. 2017.

[Fru+17] Mario Frustaci et al. “Evaluating critical security issues of the IoT world: Present and future challenges”. In: *IEEE Internet of things journal* 5.4 (2017), pp. 2483–2495.

[Gar15] Sebastian Garcia. “Modelling the network behaviour of malware to block malicious patterns. the stratosphere project: a behavioural ips”. In: *Virus Bulletin* (2015), pp. 1–8.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[Gér19] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[Goo+14] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[GPP11] Jian Guo, Thomas Peyrin, and Axel Poschmann. "The PHOTON family of lightweight hash functions". In: *Annual Cryptology Conference*. Springer. 2011, pp. 222–239.

[GS17] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey". In: *ACM Computing Surveys (CSUR)* 50.4 (2017), pp. 1–36.

[Gua20] The Guardian. *DDoS attack that disrupted internet was largest of its kind in history, experts say.* 2016 (accessed June 25, 2020). URL: `https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet`.

[Gul+17a] Ishaan Gulrajani et al. "Improved Training of Wasserstein GANs". In: *Advances in Neural Information Processing Systems 30*. 2017.

[Gul+17b] Ishaan Gulrajani et al. "Improved training of wasserstein gans". In: *Advances in neural information processing systems*. 2017, pp. 5767–5777.

[Har19] Sam Haria. "The growth of the hide and seek botnet". In: *Network Security* 2019.3 (2019), pp. 14–17.

[Hin+12] Geoffrey E Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580* (2012).

[Hod+16] Elike Hodo et al. "Threat analysis of IoT networks using artificial neural network intrusion detection system". In: *2016 International Symposium on Networks, Computers and Communications (ISNCC)*. IEEE. 2016, pp. 1–6.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[HZ14] Debiao He and Sherali Zeadally. "An analysis of RFID authentication schemes for internet of things in healthcare environment using elliptic curve cryptography". In: *IEEE internet of things journal* 2.1 (2014), pp. 72–83.

[IDC20] International Data Corporation (IDC). *The Growth in Connected IoT Devices Is Expected to Generate 79.4ZB of Data in 2025, According to a New IDC Forecast.* 2019 (accessed June 24, 2020). URL: `https://www.idc.com/getdoc.jsp?containerId=prUS45213219`.

[Ind20] The Independent. *Hackers Now Able to Take Control of Cars to Cause Deliberate Accidents, Scientists Warn.* 2017 (accessed June 24, 2020). URL: `https://www.independent.co.uk/life-style/gadgets-and-tech/news/computer-hackers-control-car-deliberate-accidents-national-security-issue-a8066466.html`.

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *International Conference on Machine Learning.* 2015.

[Jav+16] Ahmad Javaid et al. "A deep learning approach for network intrusion detection system". In: *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS).* 2016, pp. 21–26.

[Jer17] James A Jerkins. "Motivating a market or regulatory solution to IoT insecurity with the Mirai botnet code". In: *2017 IEEE 7th annual computing and communication workshop and conference (CCWC).* IEEE. 2017, pp. 1–5.

[JM09] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2nd Edition).* USA: Prentice-Hall, Inc., 2009. ISBN: 0131873210.

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[KH16] Matt J Kusner and José Miguel Hernández-Lobato. "Gans for sequences of discrete elements with the gumbel-softmax distribution". In: *arXiv preprint arXiv:1611.04051* (2016).

[KK16] Dan Klinedinst and Christopher King. "On board diagnostics: Risks and vulnerabilities of the connected vehicle". In: *Software Engineering Institute-Carnegie Mellon University* 10 (2016).

[KKS17] Georgios Kambourakis, Constantinos Kolias, and Angelos Stavrou. "The mirai botnet and the iot zombie armies". In: *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM).* IEEE. 2017, pp. 267–272.

[KLA19] Tero Karras, Samuli Laine, and Timo Aila. "A style-based generator architecture for generative adversarial networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2019.

[Kol+15] Constantinos Kolias et al. "Intrusion detection in 802.11 networks: empirical evaluation of threats and a public dataset". In: *IEEE Communications Surveys & Tutorials* 18.1 (2015), pp. 184–208.

[Kol+16] Bojan Kolosnjaji et al. "Deep learning for classification of malware system call sequences". In: *Australasian Joint Conference on Artificial Intelligence.* Springer. 2016, pp. 137–149.

[Kol+17] Constantinos Kolias et al. “DDoS in the IoT: Mirai and other botnets”. In: *Computer* 50.7 (2017), pp. 80–84.

[KS15] Sheetal Kalra and Sandeep K Sood. “Secure authentication scheme for IoT and cloud servers”. In: *Pervasive and Mobile Computing* 24 (2015), pp. 210–223.

[KS18] Minhaj Ahmad Khan and Khaled Salah. “IoT security: Review, blockchain solutions, and open challenges”. In: *Future Generation Computer Systems* 82 (2018), pp. 395–411.

[Kum+14] Yuichi Kumano et al. “Towards real-time processing for application identification of encrypted traffic”. In: *2014 International Conference on Computing, Networking and Communications (ICNC)*. IEEE. 2014, pp. 136–140.

[Kum+18] Saru Kumari et al. “A secure authentication scheme based on elliptic curve cryptography for IoT and cloud servers”. In: *The Journal of Supercomputing* 74.12 (2018), pp. 6428–6453.

[KW13] Diederik P Kingma and Max Welling. “Auto-encoding variational bayes”. In: *arXiv preprint arXiv:1312.6114* (2013).

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444.

[Le+19] Franck Le et al. “Policy-Based Identification of IoT Devices’ Vendor and Type by DNS Traffic Analysis”. In: *Policy-Based Autonomic Data Governance*. Springer, 2019, pp. 180–201.

[Lea+07] Gregor Leander et al. “New lightweight DES variants”. In: *International Workshop on Fast Software Encryption*. Springer. 2007, pp. 196–210.

[LeC+98] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[Li+18] Zhen Li et al. “Vuldeepecker: A deep learning-based system for vulnerability detection”. In: *arXiv preprint arXiv:1801.01681* (2018).

[Lin+16] W. Linlin et al. “On the Impact of Packet Inter Arrival Time for Early Stage Traffic Identification”. In: *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 2016.

[Lin+19] Zinan Lin et al. “Towards Oblivious Network Analysis using Generative Adversarial Networks”. In: *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. 2019, pp. 43–51.

[LN18] Tie Luo and Sai G Nagarajan. “Distributed anomaly detection using autoencoder neural networks in wsn for iot”. In: *2018 IEEE International Conference on Communications (ICC)*. IEEE. 2018, pp. 1–6.

[Lou] Louis Columbus. *Roundup Of Internet Of Things Forecasts And Market Estimates, 2016.* https://www.forbes.com/sites/louiscolumbus/2016/11/27/roundup-of-internet-of-things-forecasts-and-market-estimates-2016/.

[Mah+15] Rwan Mahmoud et al. "Internet of things (IoT) security: Current status, challenges and prospective measures". In: *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST).* IEEE. 2015, pp. 336–341.

[Mao+17] Xudong Mao et al. "Least squares generative adversarial networks". In: *Proceedings of the IEEE international conference on computer vision.* 2017, pp. 2794–2802.

[Mar+19] Samuel Marchal et al. "Audi: Toward autonomous iot device-type identification using periodic communication". In: *IEEE Journal on Selected Areas in Communications* 37.6 (2019), pp. 1402–1412.

[Mei+17a] Yair Meidan et al. "Detection of unauthorized iot devices using machine learning techniques". In: *arXiv preprint arXiv:1709.04647* (2017).

[Mei+17b] Yair Meidan et al. "ProfilIoT: a machine learning approach for IoT device identification based on network traffic analysis". In: *Proceedings of the symposium on applied computing.* 2017, pp. 506–509.

[Mei+18] Yair Meidan et al. "N-baiot—network-based detection of iot botnet attacks using deep autoencoders". In: *IEEE Pervasive Computing* 17.3 (2018), pp. 12–22.

[MH08] Laurens van der Maaten and Geoffrey Hinton. "Viualizing data using t-SNE". In: *Journal of Machine Learning Research* (2008).

[Mic20] Trend Micro. *Persirai: New Internet of Things (IoT) Botnet Targets IP Cameras.* 2017 (accessed June 25, 2020). URL: https://blog.trendmicro.com/trendlabs-security-intelligence/persirai-new-internet-things-iot-botnet-targets-ip-cameras/.

[Mie+17] Markus Miettinen et al. "Iot sentinel: Automated device-type identification for security enforcement in iot". In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS).* IEEE. 2017, pp. 2177–2184.

[Mik+13] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[Mir+18] Yisroel Mirsky et al. "Kitsune: an ensemble of autoencoders for online network intrusion detection". In: *arXiv preprint arXiv:1802.09089* (2018).

[Mit97] TM Mitchell. "Machine Learning, McGraw-Hill Higher Education". In: *New York* (1997).

[MS15] Nour Moustafa and Jill Slay. “UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)”. In: *2015 military communications and information systems conference (MilCIS)*. IEEE. 2015, pp. 1–6.

[MTC18] Nour Moustafa, Benjamin Turnbull, and Kim-Kwang Raymond Choo. “An ensemble intrusion detection technique based on proposed statistical flow features for protecting network traffic of internet of things”. In: *IEEE Internet of Things Journal* 6.3 (2018), pp. 4815–4830.

[Nes+19] Nataliia Neshenko et al. “Demystifying IoT security: an exhaustive survey on IoT vulnerabilities and a first empirical look on internet-scale IoT exploitations”. In: *IEEE Communications Surveys & Tutorials* 21.3 (2019), pp. 2702–2733.

[Ng+11] Andrew Ng et al. “Sparse autoencoder”. In: *CS294A Lecture notes* 72.2011 (2011), pp. 1–19.

[Ngu+19] Thien Duc Nguyen et al. “DÏoT: A federated self-learning anomaly detection system for IoT”. In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2019, pp. 756–767.

[OWA20] OWASP. *Owasp Internet of Things Top 10 2018*. 2018 (accessed June 24, 2020). URL: `https://owasp.org/www-pdf-archive/OWASP-IoT-Top-10-2018-final.pdf`.

[Pa+15] Yin Minn Pa Pa et al. “IoTPOT: analysing the rise of IoT compromises”. In: *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)*. 2015.

[Paj+16] Hamed Haddad Pajouh et al. “A two-layer dimension reduction and two-tier classification model for anomaly-based intrusion detection in IoT backbone networks”. In: *IEEE Transactions on Emerging Topics in Computing* (2016).

[Pas+15] Razvan Pascanu et al. “Malware classification with recurrent networks”. In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2015, pp. 1916–1920.

[Qaz+13] Zafar Ayyub Qazi et al. “Application-awareness in SDN”. In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM ’13. 2013.

[Rad20a] Radware. *”BrickerBot” Results In PDoS Attack*. 2017 (accessed June 25, 2020). URL: `https://security.radware.com/ddos-threats-attacks/brickerbot-pdos-permanent-denial-of-service/`.

[Rad20b] Radware. *Reaper Botnet*. 2017 (accessed June 25, 2020). URL: `https://security.radware.com/ddos-threats-attacks/threat-advisories-attack-reports/reaper-botnet/`.

[RG18] Maria Rigaki and Sebastian Garcia. “Bringing a gan to a knife-fight: Adapting malware communication to avoid detection”. In: *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2018, pp. 70–75.

[Rin+19] Markus Ring et al. “Flow-based network traffic generation using generative adversarial networks”. In: *Computers & Security* 82 (2019), pp. 156–172.

[RS16] Eyal Ronen and Adi Shamir. “Extended functionality attacks on IoT devices: The case of smart lights”. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 3–12.

[RSG16] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. “” Why should I trust you?” Explaining the predictions of any classifier”. In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 2016, pp. 1135–1144.

[Sac+17] Vinay Sachidananda et al. “Let the cat out of the bag: A holistic approach towards security analysis of the internet of things”. In: *Proceedings of the 3rd ACM International Workshop on IoT Privacy, Trust, and Security*. 2017, pp. 3–10.

[SB15] Joshua Saxe and Konstantin Berlin. “Deep neural network based malware detection using two dimensional binary program features”. In: *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE. 2015, pp. 11–20.

[Sci20] Scikit-learn. *Receiver Operating Characteristic (ROC)*. (accessed June 29, 2020). URL: `https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html`.

[SGK17] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. “Learning Important Features Through Propagating Activation Differences”. In: *International Conference on Machine Learning*. 2017, pp. 3145–3153.

[Shm20] Boaz Shmueli. *Multi-Class Metrics Made Simple, Part II: the F1-score*. 2019 (accessed June 29, 2020). URL: `https://towardsdatascience.com/multi-class-metrics-made-simple-part-ii-the-f1-score-ebe8b2c2ca1`.

[Sho+18] Nathan Shone et al. “A deep learning approach to network intrusion detection”. In: *IEEE transactions on emerging topics in computational intelligence* 2.1 (2018), pp. 41–50.

[Shu17] Prachi Shukla. “Ml-ids: A machine learning approach to detect wormhole attacks in internet of things”. In: *2017 Intelligent Systems Conference (IntelliSys)*. IEEE. 2017, pp. 234–240.

[Sin+17] Saurabh Singh et al. “Advanced lightweight encryption algorithms for IoT devices: survey, challenges and solutions”. In: *Journal of Ambient Intelligence and Humanized Computing* (2017), pp. 1–18.

[Siv+18] Arunan Sivanathan et al. "Classifying IoT devices in smart environments using network traffic characteristics". In: *IEEE Transactions on Mobile Computing* 18.8 (2018), pp. 1745–1759.

[SLG18] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. "Toward generating a new intrusion detection dataset and intrusion traffic characterization." In: *ICISSP*. 2018, pp. 108–116.

[SMP18] Saleh Soltan, Prateek Mittal, and H Vincent Poor. "BlackIoT: IoT botnet of high wattage devices can disrupt the power grid". In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 15–32.

[SNB18] Susha Surendran, Amira Nassef, and Babak D Beheshti. "A survey of cryptographic algorithms for IoT devices". In: *2018 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*. IEEE. 2018, pp. 1–8.

[SP10] Robin Sommer and Vern Paxson. "Outside the closed world: On using machine learning for network intrusion detection". In: *2010 IEEE symposium on security and privacy*. IEEE. 2010, pp. 305–316.

[Sri+14] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[SS19] Sofia Serrano and Noah A Smith. "Is attention interpretable?" In: *arXiv preprint arXiv:1906.03731* (2019).

[Sun+17] Degang Sun et al. "A New Mimicking Attack by LSGAN". In: *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2017, pp. 441–447.

[Sym20] Symantec. *Internet Security Threat Report 2019*. 2019 (accessed June 24, 2020). URL: `https://docs.broadcom.com/doc/istr-24-2019-en`.

[TH12] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[Tha+18] Vijayanand Thangavelu et al. "Deft: A distributed iot fingerprinting technique". In: *IEEE Internet of Things Journal* 6.1 (2018), pp. 940–952.

[Vas+17] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017), pp. 5998–6008.

[VR18] Abhishek Verma and Virender Ranga. "Statistical analysis of CIDDS-001 dataset for network intrusion detection systems using distance-based machine learning". In: *Procedia Computer Science* 125 (2018), pp. 709–716.

[Wan+17] Aohui Wang et al. “An inside look at IoT malware”. In: *International Conference on Industrial IoT Technologies and Applications*. Springer. 2017, pp. 176–186.

[Win+12] Tim Winter et al. “RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks.” In: *rfc* 6550 (2012), pp. 1–157.

[WP19] Sarah Wiegreffe and Yuval Pinter. “Attention is not not explanation”. In: *arXiv preprint arXiv:1908.04626* (2019).

[Wu+19] Di Wu et al. “Evading machine learning botnet detection models via deep reinforcement learning”. In: *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE. 2019, pp. 1–6.

[Yan+18] Kai Yang et al. “Active learning for wireless IoT intrusion detection”. In: *IEEE Wireless Communications* 25.6 (2018), pp. 19–25.

[Yan+19] Qiao Yan et al. “Automatically synthesizing DoS attack traces using generative adversarial networks”. In: *International Journal of Machine Learning and Cybernetics* 10.12 (2019), pp. 3387–3396.

[Ye+17] Yanfang Ye et al. “A survey on malware detection using data mining techniques”. In: *ACM Computing Surveys (CSUR)* 50.3 (2017), pp. 1–40.

[Yin+17] Chuanlong Yin et al. “A deep learning approach for intrusion detection using recurrent neural networks”. In: *Ieee Access* 5 (2017), pp. 21954–21961.

[YLR11] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. “Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning”. In: *Proceedings of the 5th USENIX conference on Offensive technologies*. 2011, pp. 13–13.

[Yu+17] Lantao Yu et al. “Seqgan: Sequence generative adversarial nets with policy gradient”. In: *Thirty-First AAAI Conference on Artificial Intelligence*. 2017.

[Zha+14] Zhi-Kai Zhang et al. “IoT security: ongoing challenges and research opportunities”. In: *2014 IEEE 7th international conference on service-oriented computing and applications*. IEEE. 2014, pp. 230–234.

[Zho+18a] Jie Zhou et al. “Graph neural networks: A review of methods and applications”. In: *arXiv preprint arXiv:1812.08434* (2018).

[Zho+18b] Wei Zhou et al. “The effect of iot new features on security and privacy: New threats, existing solutions, and challenges yet to be solved”. In: *IEEE Internet of Things Journal* 6.2 (2018), pp. 1606–1616.

ECOLE
DOCTORALE

Title : Deep Learning for Internet of Things (IoT) Network Security

Keywords : Deep Learning, Internet of Things, Network Security, Autoencoder, GAN

Abstract : The growing Internet of Things (IoT) introduces new security challenges for network activity monitoring. Most IoT devices are vulnerable because of a lack of security awareness from device manufacturers and end users. As a consequence, they have become prime targets for malware developers who want to turn them into bots and use them to perform large scale attacks.
Contrary to general-purpose devices, an IoT device is designed to perform very specific tasks. Hence, its networking behavior is very stable and predictable making it well suited for data analysis techniques. Therefore, the first part of this thesis focuses on leveraging recent advances in the field of deep learning to develop network monitoring tools for the IoT. Two types of network monitoring tools are explored : IoT device type recognition systems and IoT network Intrusion Detection Systems (NIDS). For IoT device type recognition, supervised machine learning algorithms are trained to perform network traffic classification and determine what IoT device the traffic belongs to. The IoT NIDS consists of a set of autoencoders, each trained for a different IoT device type. The autoencoders learn the legitimate networking behavior profile and detect any deviation from it. Experiments using network traffic data produced by a smart home show that the proposed models achieve high performance.
Despite yielding promising results, training and testing machine learning based network monitoring systems requires tremendous amount of IoT network traffic data. But, very few IoT network traffic datasets are publicly available. Physically operating thousands of real IoT devices can be very costly and can rise privacy concerns. In the second part of this thesis, we propose to leverage Generative Adversarial Networks (GAN) to generate bidirectional flows that look like they were produced by a real IoT device. Generated bidirectional flows consist of a sequence of individual packet sizes along with a duration value. Hence, in addition to generating packet-level features which are the sizes of individual packets, our developed generator implicitly learns to comply with flow-level characteristics, such as the ordering of the packets, the total number of packets and bytes in a bidirectional flow or the total duration of the flow. Experimental results using data produced by a smart speaker show that our method allows us to generate high quality and realistic looking synthetic bidirectional flows.

Titre : Apprentissage profond (*deep learning*) pour la sécurité des réseaux d'objets connectés (IoT)

Mots clés : Deep Learning, Internet des Objets, Sécurité Réseau, Autoencoder, GAN

Résumé : L'internet des objets (IoT) introduit de nouveaux défis de sécurité pour la surveillance des réseaux. La plupart des appareils IoT sont vulnérables en raison d'un manque de sensibilisation à la sécurité des fabricants d'appareils et des consommateurs. En conséquence, ces appareils sont devenus des cibles privilégiées pour les développeurs de malware qui veulent les transformer en bots pour ensuite pouvoir les utiliser pour mener des attaques à grandes échelles.
Contrairement à un ordinateur de bureau, un objet IoT est conçu pour accomplir des tâches très spécifiques. Par conséquent, son comportement réseau est très stable et prévisible, ce qui le rend bien adapté aux techniques d'analyse de données. Ainsi, la première partie de cette thèse tire profit des algorithmes de *deep learning* pour développer des outils de surveillance des réseaux IoT. Deux types d'outils de surveillance réseau sont explorés : Les systèmes de reconnaissance de type d'objets IoT et les systèmes de détection d'intrusion réseau IoT. Pour la reconnaissance des types d'objets IoT, des algorithmes d'apprentissage supervisés sont entraînés pour classifier le trafic réseau et déterminer à quel objet IoT le trafic appartient. Le système de détection d'intrusion réseau IoT consiste en un ensemble d'*autoencoders*, chacun étant entraîné pour un type d'objet IoT différent. Les *autoencoders* apprennent le profil du comportement réseau légitime et détectent tout écart par rapport à celui-ci. Les résultats expérimentaux en utilisant des données réseau produites par une maison connectée montrent que les modèles proposés atteignent des performances élevées.
Bien que permettant d'obtenir des résultats prometteurs, l'entraînement et l'évaluation des modèles de *deep learning* nécessitent une quantité énorme de données réseaux IoT. Or, très peu de jeux de données de trafic réseau IoT sont accessibles au public. Le déploiement physique de milliers d'objets IoT réels peut être très coûteux et peut poser problème quant au respect de la vie privée. Ainsi, dans la deuxième partie de cette thèse, nous proposons d'exploiter des *GAN (Generative Adversarial Networks)* pour générer des flux bidirectionnels qui ressemblent à ceux produits par un véritable objet IoT. Un flux bidirectionnel est représenté par la séquence des tailles des paquets ainsi que de la durée du flux. Par conséquent, en plus de générer des caractéristiques au niveau des paquets, tel que la taille de chaque paquet, notre générateur apprend implicitement à se conformer aux caractéristiques au niveau du flux, comme le nombre total de paquets et d'octets dans un flux ou sa durée totale. Des résultats expérimentaux utilisant des données produites par un haut-parleur intelligent montrent que notre méthode permet de générer des flux bidirectionnels synthétiques réalistes et de haute qualité.

Institut Polytechnique de Paris
91120 Palaiseau, France