

# *Contents*

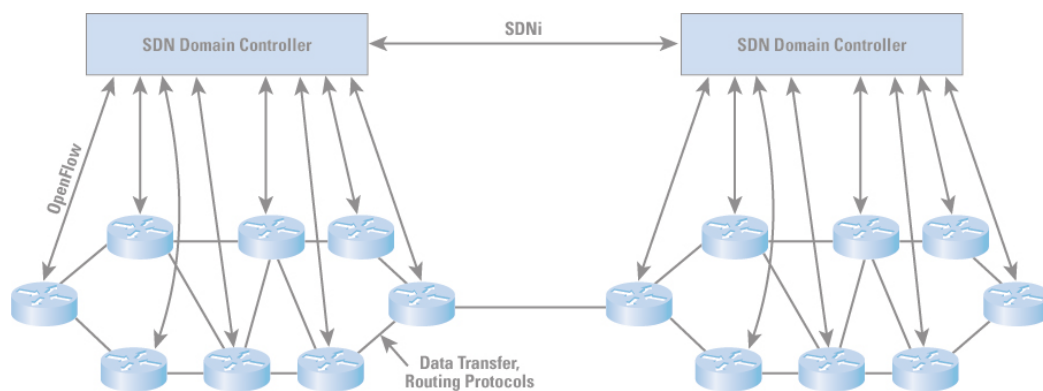
	Page no.
<b>Abstract</b>	<b>1</b>
<b>1 Introduction</b>	<b>1</b>
1.1 SDN . . . . .	1
1.2 A Bit of History . . . . .	2
1.3 Necessity of SDN . . . . .	3
1.4 Software Defined Networking vs. Traditional Networking . . . .	4
1.5 Key Challenges of SDN . . . . .	6
<b>2 Architecture of SDN</b>	<b>9</b>
2.1 The Control Plane . . . . .	10
2.1.1 SDN Controller Functional Components . . . . .	11
2.2 Data Plane . . . . .	12
2.3 Application Plane . . . . .	13
<b>3 Security in SDN</b>	<b>15</b>
3.1 DDoS . . . . .	15
3.2 DoS Attacks . . . . .	16
3.3 Hijacked/Rogue Controller . . . . .	16
3.4 Malicious Applications . . . . .	17
3.5 Control-Data Plane Link Attacks . . . . .	17
3.6 Eavesdropping Attacks . . . . .	18
3.7 Side by Side Comparison of Attacks . . . . .	19
3.8 Literature Survey . . . . .	19
<b>4 Scope of Work</b>	<b>23</b>
4.1 Problem Statement . . . . .	23
4.2 Assumptions . . . . .	23
4.3 Proposed Framework . . . . .	24
4.3.1 Usual and Unusual TCP handshakes . . . . .	24
4.4 Behaviour of TCP Handshakes . . . . .	26
4.5 TCP Latency . . . . .	30
4.6 An Architecture for DoS Detection . . . . .	30
4.6.1 Collection Module . . . . .	31

4.6.2	Management of Data Structure . . . . .	33
4.7	Decision Module . . . . .	35
4.7.1	Detection . . . . .	35
4.8	Experimental Results . . . . .	36
4.9	Conclusion . . . . .	38
<b>5</b>	<b>Appendix</b>	<b>39</b>
5.1	Mininet . . . . .	39
5.2	Installation, Setup and Command . . . . .	39
5.3	command . . . . .	41
5.4	controller . . . . .	43
5.5	Generate Attack using hping3 . . . . .	47
5.6	Python code for proposed framework . . . . .	49
	<b>Bibliography</b>	<b>61</b>

# *Abstract*

Software Defined Networking (SDN) is a new paradigm for developing network management application using OpenFlow framework. The most interesting concept of this paradigm is centralized network management. However, this centralized nature of SDN is also vulnerable in terms of security. The Denial of Service (DoS) attacks pose a major threat to Internet users and services. This document proposes an approach to identify synchronize (SYN) flooding attacks from the victim's side. Firstly a TCP handshake monitoring system is developed which can classify different types of TCP handshakes. This information is used to keep track of the unusual handshakes in the network leading to the detection of such flooding attacks.

# Chapter 1 : Introduction



# *Introduction*

The world has seen rapid and unparalleled changes in technology over the past ten to twenty years. A few year ago, storage, computing, and network resources were kept physically and operationally separate from one another. This rapid increase of mobile devices has made the availability of information location-independent. By 2000, it was invented that the purpose of allowing large number of computers connected over a network to exchange information via services without human interaction. The goal of Software-Defined Networking is to enable cloud and network engineers and administrators to respond quickly to changing business requirements via a centralized control console.

Software Defined Networking (SDN) introduces a new communication network management system that decouple network control and forwarding functions. This feature gives the opportunity to enable network control directly programmable. Abstraction of underlying infrastructure from applications and network services is also maintained.

## **1.1 SDN**

Software Defined Networks (SDN) introduces a new communication network management paradigm and has gained attention from academia and industry. The main concept of SDN is that it decouples the control layer from the data layer.

One primary goal of SDN is to allow a network controller, called the control plane, to manage the entire network by configuring routing mechanisms for underlying switches. The switches, also called the data plane, are responsible for data forwarding according to their forwarding tables. If that specific entry of destination is not present in the data plane, it consults with control plane for routing. Routing computation and network management are handled by the controller. The entries of forwarding tables on switches are determined and managed by the controller through a secured communication protocol between switches and the controller.

Multiple protocols are available for communicate between control plane and data plane. Such as NETCONF, ONF, XMPP, OVSDB, MPLS-TP [1] etc. More specifically, enterprises adopt OpenFlow [2] as communication protocol for secure and efficient communication between switches and the controller. As

specified in the OpenFlow protocol [3], switches must forward a packet-in request to the controller if they receive a new packet (an indication of new flow) that they do not have any matching entry in the flow table to forward the packet. The controller receives the packet-in request and computes a routing path for the new flow before sending packet-out packets to notify corresponding switches to update their forwarding tables, which is an indication of the establishment of a new flow path. Subsequent packets belonging to the same flow will be forwarded accordingly by switches.

Most interesting feature of SDN is its abstraction. Instead of sending specific code to the devices, the machines can talk to the controller in generalized terms. An excellent example of this is our use of the printer. A unique command is used for printing which goes out from the PCs to the actual printer itself. OS sends a generalized message to the driver and it sends the specific messages (code) required to print out the document. From the users' perspective, they simply hit the print button. With SDN, concepts are abstracted and the user makes requests of the network in a similar fashion to hitting the print button from an application. But instead of the application sending print commands, it can send link up/down or bandwidth commands, or pull reports on different aspects of the network. And just as with the computer where the user is printing from an application (like a word processor) which runs on top of the OS, there are applications that run on top of the SDN network controller.

## 1.2 A Bit of History

Although a recent concept, the idea of SDN has a longer history. OpenFlow is a communications protocol and standard created and managed by the Open Networking Foundation. Its inception was founded on the basis of an initiative to promote awareness and adoption of software-defined networking (or SDN). SDN is a computer networking concept that was developed as a joint research venture between UC Berkeley and Stanford University circa 2005. The hype cycle began with Martin Casado's thesis work at Stanford in 2005. Casado worked alongside a number of individuals at Stanford, including Nick McKeown and Scott Shenker. McKeown and Shenker joined with Casado to form Nicira Networks in 2007. Guido Appenzeller later also became part of the OpenFlow effort at Stanford. Data plane programmability has a long history. Active networks represent one of the early attempts on building new network architectures based on this concept. The main idea behind active networks is for each node to have the capability to perform computations on, or modify the content of packets.

## 1.3 Necessity of SDN

The architecture brings the network and networking data closer to the application layer and the applications closer to the networking layer.

Programmability (i.e. the ability to access the network via APIs and open interfaces) is central to SDN. It has its own set of APIs, logic, and the ability for an application to make requests to the network, receive events, and speak the SDN protocols.. Here the key is that programmers don't need to know the SDN protocols because they write to the controller's APIs. Programmers don't need to know the different configuration syntax or semantics of different networking devices because they program to a set of APIs on the controller that can speak to many different devices.

When SDN is applied to a centralized management plane that is separate from the data plane, it can more quickly make decisions on where data traffic can be rerouted, as this can occur programmatically with software interfaces (APIs).

Within the enterprise data center, traffic patterns have changed significantly. In contrast to client-server applications where the bulk of the communication occurs between one client and one server, today's applications access different databases and servers, creating a flurry of "east-west" machine-to-machine traffic before returning data to the end user device in the classic "north-south" traffic pattern. At the same time, users are changing network traffic patterns as they push for access to corporate content and applications from any type of device (including their own), connecting from anywhere, at any time. Finally, many enterprise data centers managers are contemplating a utility computing model, which might include a private cloud, public cloud, or some mix of both, resulting in additional traffic across the wide area network.

Network virtualization is one of cause behind creating a dynamic cloud environment. As network performance improves, so does that of the cloud environment itself. However, many businesses are dealing with the cloud as it is today. Meanwhile, server virtualization has helped businesses better manage their computing resources.

Meanwhile, server virtualization has helped businesses better manage their computing resources. Servers that were once devoted to a single application now handle dozens of virtual machines. The resulting explosion in virtual endpoints has had a direct impact on network infrastructure.

Simplifying the network through abstraction is just one way that SDN provides a more responsive playing field for application deployment. Instead of relying on outdated network implementation details, enterprise IT teams can focus on fostering an IT-friendly approach towards designing and implementing network services that best serve application needs.

In case of the hybrid cloud an application may run in a private cloud or data center yet utilize the public cloud when the demand for computing capacity spikes or cost can be reduced. Historically, cloud bursting was typically used

only in environments with non-mission critical applications or services, but with the network tie-in and software principles applied, the use case shifts. Applications now remain in compliance with the IT organizations' policies and regulations. It also allows for the application to run across different platforms regardless of where the application was built.

## 1.4 Software Defined Networking vs. Traditional Networking

- Traditional Networking

Network Devices have a control plane that provides information used to build a forwarding table. They also consist of a data plane that consults the forwarding table. The forwarding table is used by the network device to make a decision on where to send frames or packets entering the device. Both of these planes exist directly on the networking device. The following figure shows roughly how a network device functions 1.1.

- They are Static and inflexible networks. They are not useful for new business ventures. They possess little agility and flexibility
- They are Hardware appliances.
- They have distributed control plane.
- They use custom ASICs and FPGAs.
- They work using protocols.

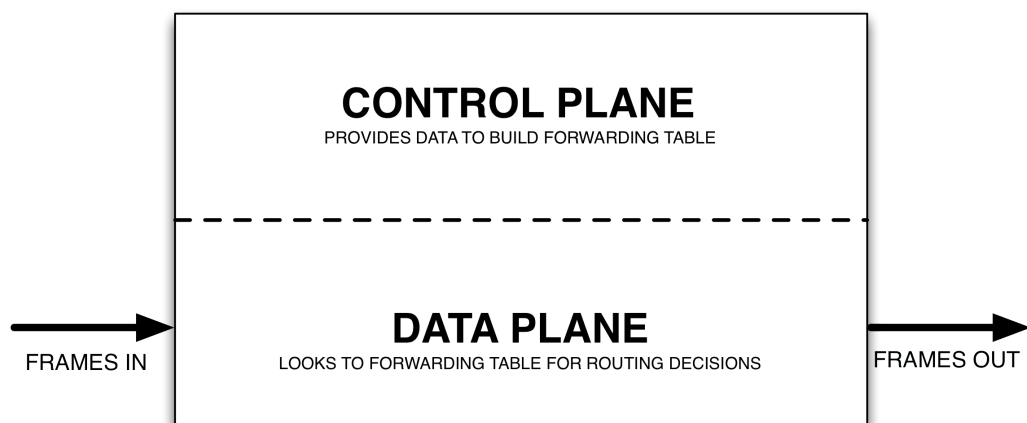


Figure 1.1: Traditional Networking



- **Software Defined Networking**  
In SDN abstracts this concept, and places the Control Plane functions on an SDN controller. The SDN controller can be a server running SDN software. The Controller communicates with a physical or virtual switch Data Plane through a protocol called OpenFlow. OpenFlow conveys the instructions to the data plane on how to forward data. The network device must run the OpenFlow protocol for this to be possible. You can see how SDN differs from traditional networking in the following graphic. The following figure shows roughly how a SDN network device functions 1.2.
- They are programmable networks during deployment time as well as at later stage based on change in the requirements. They help new business ventures through flexibility, agility and virtualization.
- They are configured using open software.
- They have logically centralized control plane.
- They use merchant silicon.
- They use APIs to configure as per need.

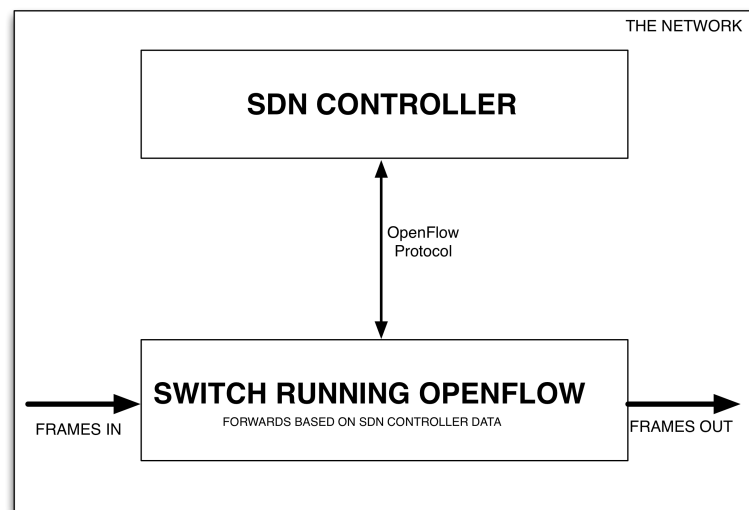


Figure 1.2: SDN Networking

## 1.5 Key Challenges of SDN

SDN holds great promise in terms of simplifying network deployment. It also reduce the total cost of managing enterprise and carrier networks by providing programmable network services. However, a number of challenges remain to be addressed. This section focuses on some specific questions arising from the challenges of SDN.

- Performance and Flexibility

One fundamental challenge of SDN is how to handle high-security high-performance packet processing flows in an efficient manner. There are two elements to consider: performance and programmability/flexibility.

**Performance** refers specifically to the processing speed of the network node considering both throughput and latency[4]. Programmability means the capability to change and/or accept a new set of instructions in order to alter functional behavior. **Flexibility** is the ability to adapt systems to support new unforeseen features (e.g., applications, protocols, security measures).

- Scalability

Assuming that the performance requirements can be achieved within the hybrid programmable architecture, a further issue that has seen some discussion but limited solution is scalability in SDN.

The issue can loosely be split into controller scalability and network node scalability. The focus here is on controller scalability in which three specific challenges are identified. The **first** is the latency introduced by exchanging network information between multiple nodes and a single controller. The **second** is how SDN controllers communicate with other controllers using the east and westbound APIs. The **third** challenge is the size and operation of the controller back-end database.

Considering the first issue, a distributed or peer-to-peer controller infrastructure would share the communication overload of the controller. However, this approach does not eliminate the second challenge of controller-to-controller interactions, for which an overall network view is required.

Within a pure SDN environment, a single controller or group of controllers would provide control plane services for a wider number of data forwarding nodes, thus allowing a system wide view of network resources.

A specific solution to controller scalability is HyperFlow [5]. HyperFlow is a controller application that sits on the NOX controller and works with an event propagation system. The Hyper-Flow application selectively publishes events that change the state of the system, and other controllers replay all the published events to reconstruct the state. By

this means all the controllers share the same consistent network-wide view. Onix [6] is a distributed control platform providing abstractions for partitioning and distributing network state onto multiple distributed controllers. Flexibility of SDN provides an opportunity in terms of network manageability and functional scalability.

- Mobility

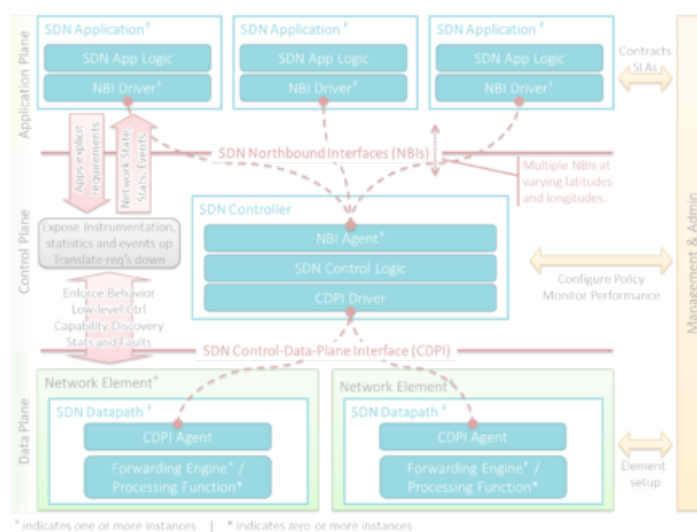
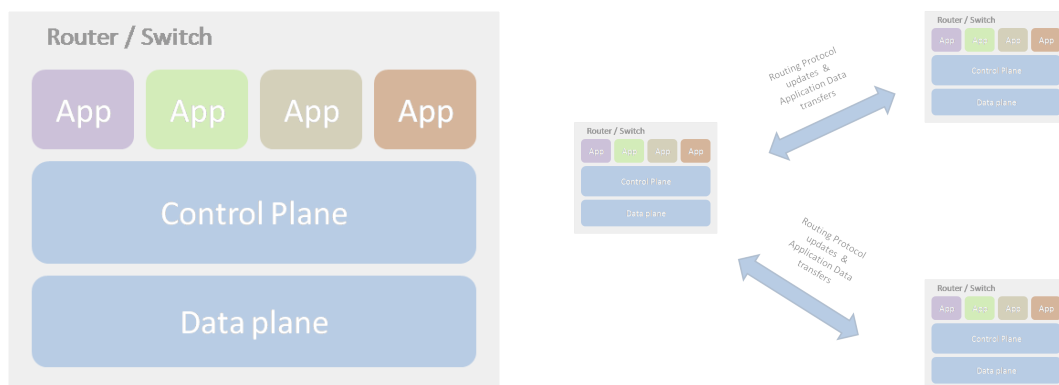
The software-defined networking (SDN) concept is an efficient solution to manage highly complex wired networks, such as large-scaled data centers and fixed Internet network. The basic idea of SDN is to separate the control and data forwarding functions of routers so that the traffic engineering (TE) optimization can be dynamically made by a central controller. However the application of SDN concept in dense wireless networks is not straightforward. Unlike wired networks, the channel capacity of wireless networks always changes, depending on the mobility of users, channel fading and interference. In dense wireless networks with mobile users, the impacts of varying channel capacity are more difficult to handle because of small cell size.

- Security

SDN gives network administrators the ability to collect traffic statistics from the network devices and pass these onto the control plane for processing. This allows for in depth security-analysis without any negative effects on the performance of the data-plane. SDN makes it possible to configure security policies centrally at the controller and push them out network wide. Security policies on every router or switch in the network. SDN allows easy integration of third-party software into the environment via the SDN framework, meaning that plugin-like applications can be deployed to gain certain security and non-security related tasks. As SDN controllers hold a global view of the network, they introduce the possibility of network-wide intrusion detection systems, which utilize the traffic statistics they receive from the network devices. As devices are required to communicate back to the controller at regular intervals, it ensures that compromised devices are found quickly and reduces the chances of false positives, an issue that is still yet to be solved in the context of traditional networks.

On one hand, the increased flexibility and ability to innovate with network applications and network control, gives programmers the ability to better protect against traditional network attacks i.e. TCP-based DoS attacks, eavesdropping, man-in-the-middle attacks. On the other hand, the links between control-plane, data-plane and application-plane bring about new attack platforms for adversaries attempting to illegitimately use network services. Much research has been carried out over the years on traditional security attacks

# Chapter 2 : The SDN Architecture



# *Architecture of SDN*

A software-defined networking (SDN) architecture (or SDN architecture) defines how a networking and computing system can be built using a combination of open, software-based technologies and commodity networking hardware that separate the control plane and the data layer of the networking stack.

Traditionally, both the control and data plane elements of a networking architecture were packaged in proprietary, integrated code distributed by one or a combination of proprietary vendors. The OpenFlow standard, created in 2008, was recognized as the first SDN architecture that defined how the control and data plane elements would be separated and communicate with each other using the OpenFlow protocol. The Open Network Foundation (ONF) is the body in charge of managing OpenFlow standards, which are open source. However, there are other standards and open-source organizations with SDN resources, so OpenFlow is not the only protocol that makes up SDN. The following figure shows an overview of SDN architecture.2.1

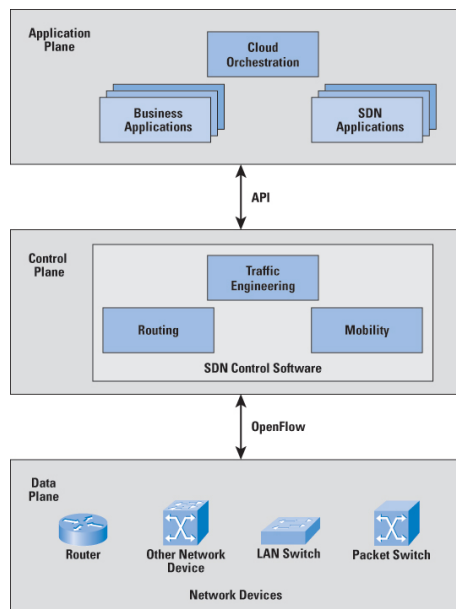


Figure 2.1: SDN Architecture

## 2.1 The Control Plane

At a very high level, the control plane establishes the local data set used to create the forwarding table entries, which are in turn used by the data plane to forward traffic between ingress and egress ports on a device. The data set used to store the network topology is called the routing information base (RIB). The RIB is often kept consistent (i.e. loop-free) through the exchange of information between other instances of control planes within the network. Forwarding table entries are commonly called the forwarding information base (FIB) and are often mirrored between the control and data planes of a typical device. The FIB is programmed once the RIB is deemed consistent and stable. To perform this task, the control entity/program has to develop a view of the network topology that satisfies certain constraints. This view of the network can be programmed manually, learned through observation, or built from pieces of information gathered through discourse with other instances of control planes, which can be through the use of one or many routing protocols, manual programming, or a combination of both. An idealized controller is shown in Figure 2.2.

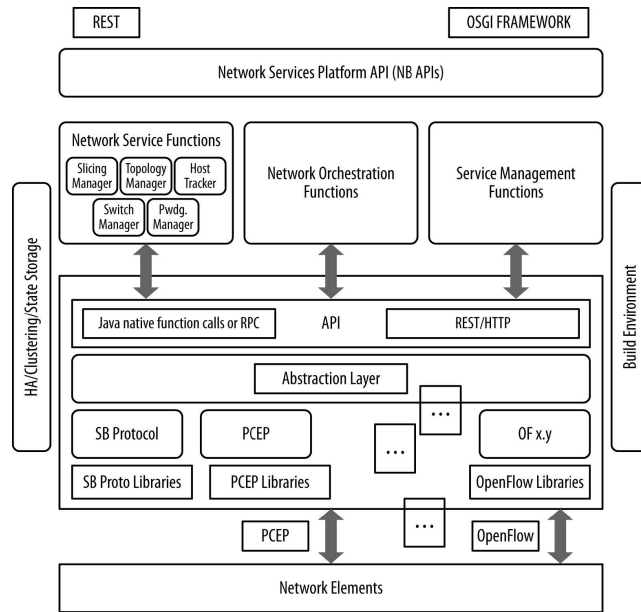


Figure 2.2: SDN Controller

The mechanics of the control and data planes is demonstrated in Figure 2.3

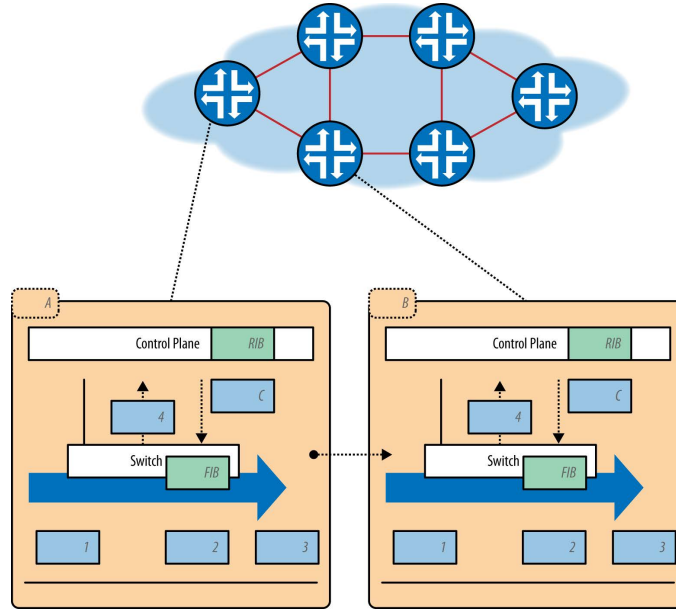


Figure 2.3: Control and data planes of a typical network

### 2.1.1 SDN Controller Functional Components

Having just stated that the SDN controller is a black box, it is nevertheless useful to conceptualize a minimum set of functional components within the SDN controller, namely data plane control function (DPCF), coordinator, virtualizer, and agent. Subject to the logical centralization requirement, an SDN controller may include arbitrary additional functions. A resource data base (RDB) models the current information model instance and the necessary supporting capabilities[7].

- Data plane control function  
The DPCF component effectively owns the subordinate resources available to it, and uses them as instructed by the OSS/coordinator or virtualizer(s) that controls them. These resources take the form of an information model instance accessed through the agent in the subordinate level.
- Coordinator  
To set up both client and server environments, management functionality is required. The coordinator is the functional component of the SDN controller that acts on behalf of the manager.

- **Virtualizer**  
In the SDN architecture, virtualization is the allocation of abstract resources to particular clients or applications; in NFV, the goal is to abstract network functions away from dedicated hardware, for example to allow them to be hosted on server platforms in cloud data centers. An SDN controller offers services to applications by way of an information model instance that is derived from the underlying resources, management-installed policy, and local or externally available support functions.
- **Agent**  
Any protocol must terminate in some kind of functional entity. A controller-agent model is appropriate for the relation between a controlled and a controlling entity, and applies recursively to the SDN architecture. The controlled entity is designated the agent, a functional component that represents the client's resources and capabilities in the server's environment.

## 2.2 Data Plane

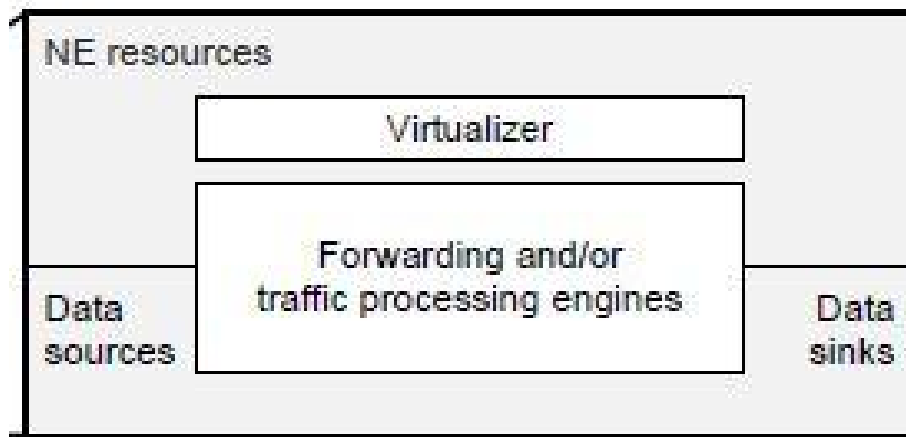


Figure 2.4: Network Elements(NE) resources detail

The data plane incorporates the resources that deal directly with customer traffic, along with the necessary supporting resources to ensure proper virtualization, connectivity, security, availability, and quality. Figure 2.4 expands the NE resources view of figure ?? accordingly. The NE resources block comprises data sources, data sinks and forwarding and/or traffic processing engines.

The data plane handles incoming datagrams (on the wire, fiber, or in wireless media) through a series of link-level operations that collect the datagram and perform basic sanity checks. A well-formed (i.e., correct) datagram is processed in the data plane by performing lookups in the FIB table (or tables,



in some implementations) that are programmed earlier by the control plane. This is sometimes referred to as the fast path for packet processing because it needs no further interrogation other than identifying the packet's destination using the preprogrammed FIB. The one exception to this processing is when packets cannot be matched to those rules, such as when an unknown destination is detected, and these packets are sent to the route processor where the control plane can further process them using the RIB.

## 2.3 Application Plane

SDN principles permit applications to specify the resources and behavior they require from the network, within the context of a business and policy agreement. The interface from the SDN controller to the application plane is called the application-controller plane interface, A-CPI (note). Figure 2.5 shows that an SDN application may itself support an A-CPI agent, which allows for recursive application hierarchies. Different levels of an application hierarchy are described as having various latitudes, depending on their degree of abstraction.

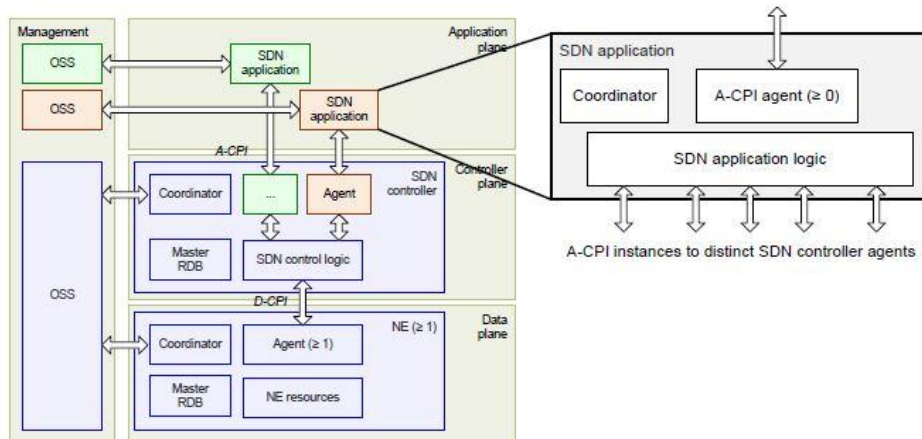
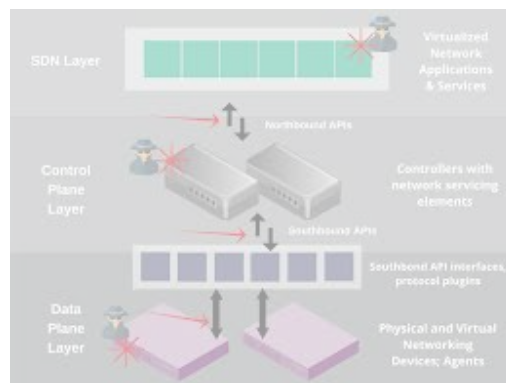
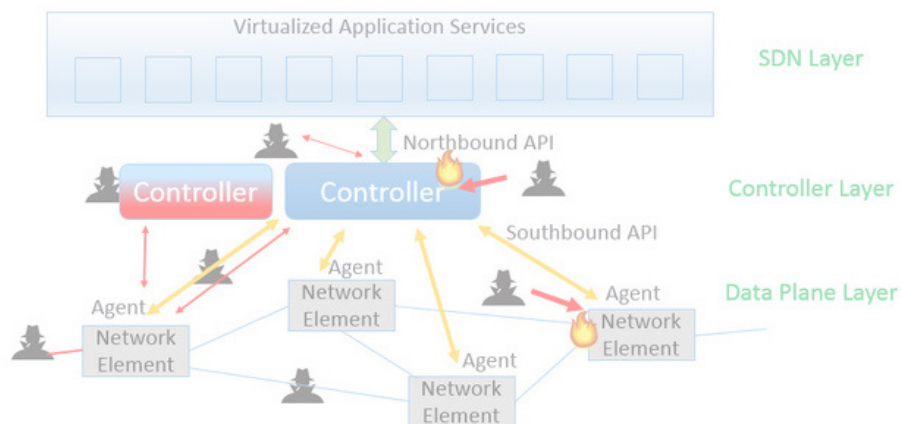


Figure 2.5: Application plane in details

# Chapter 3 : Security in SDN



## *SDN Security Attack Vectors*



# *Security in SDN*

SDN gives network administrators the ability to collect traffic statistics from the network devices and pass these onto the control plane for processing. This allows for in depth security-analysis without any negative effects on the performance of the data-plane. SDN allows easy integration of third-party software into the environment via the SDN framework, meaning that plugin-like applications can be deployed to gain certain security and non-security related tasks.

As SDN controllers hold a global view of the network, they introduce the possibility of network-wide intrusion detection systems, which utilize the traffic statistics they receive from the network devices. As devices are required to communicate back to the controller at regular intervals, it ensures that compromised devices are found quickly and reduces the chances of false positives, an issue that is still yet to be solved in the context of traditional networks [8]

Networks running under the SDN paradigm still have the same security requirements as traditional network settings. SDN completely changes the architecture and the inter-communicative aspects of the components in the network. This introduces a completely new platform for attackers looking to perform security-breaching attacks. This leads to a need for similar levels of security as traditional networks, but to defend against threats of a different nature.

## **3.1 DDoS**

Numerous types of conventional DDoS attacks can be carried out in an SDN environment, but it is a variation utilising flow entries which can be harnessed by an adversary in order to target a controller and compromise its availability. By flooding the controller with requests for a flow-decision, the controllers could become overwhelmed and cannot utilize its resources. So the controller would be rendered unable to deal with any legitimate requests it receives. By targeting the centralized point of control (i.e. the controller) it renders the entire network largely unusable. Meanwhile, data-paths currently in the network may be able to function temporarily with a downed controller. But once the hard timeout of rules in their table has expired they will be required to make contact with the controller again, which will be unable to deal with

requests. If an attacker(s) is able to be persistent with their flooding, this will eventually cause the unavailability of all network functionality.

## 3.2 DoS Attacks

At the data plane level, falsely created flow-entries can be flooded to other devices in order to consume the space in their flow entry tables. Besides that, it also helps to flood the controller. This leaves the forwarding devices unable to add any legitimate flow entries to their tables. One of the key issues with the data-plane devices within Software Defined architectures is that of the switches inability to differentiate between legitimate flow requests and illegitimate ones. This flaw allows for attackers to perform successful DoS attacks at the Data plane level by filling the switches flow-buffer with false requests [9].

Whilst it would be possible for an adversary to target an individual data-path and attempt to halt its availability, it is far more likely that the controller would be targeted, effectively creating and spreading a system-wide lapse in availability.

## 3.3 Hijacked/Rogue Controller

The controller can be thought of as the centralized ‘brain’ of an SDN. It controls the whole network from one point, making it arguably the most vital component of SDN architecture. An attacker that manages to compromise the controller essentially has control over the whole network. Ability to control the actions of the controller would allow the attacker to manipulate flow entries in any way that attacker choose, e.g. stopping certain packet types reaching their destination, re-directing packets to malicious nodes in the infrastructure etc. In conjunction with this, the attacker could aim to compromise a particular forwarding-device in the network and enable it to operate as a ‘man-in-the-middle or black-hole/grey-hole node. This would allow the attacker to potentially drop, alter or inspect the contents of any packet it receives.

Another possibility is that an attacker successfully registers a ‘rogue’/fake controller in the control plane of the network. With this rogue controller in place the adversary may be able to influence/halt the availability of other controllers, change rules installed in data-paths caches and effectively halt/-manipulate the workings of applications in the application layer.

Any attack that targets the controller/controllers in SDN architecture can have potentially devastating effects. Whilst the centralized nature and ability to collate information at one point can be massively advantageous to network administrators/programmers. On the other side, in the wrong hands it could be utilized to spearhead attacks on the integrity of control messages/sensitive application information, the availability of important services to a systems

users, and the confidentiality of sensitive user information utilized by applications in the application layer. Any approach attempting to successfully halt hijacked/rogue controllers should focus on ensuring the authenticity of the controller, before allowing it to make any changes to the network.

### **3.4 Malicious Applications**

Due to the allowance of the SDN framework for integration of third-party applications, the issue of malicious applications arises. Applications exhibiting malicious behavior within an SDN environment can have catastrophic consequences, similar to that of a compromised controller.

Authentication and authorization of an application to operate within an SDN environment is difficult to enforce. Applications relying on deep packet-inspection techniques to operate can pose potential risks to the network. They may be able to indirectly control the entire network through the information they have collected during packet-inspection.

The increased amount of data, and the way in which it is centrally located is what gives malicious applications the ability to threaten the integrity and confidentiality of user/network information that they have access to. Securing the northbound interface is a difficult task, as each application utilizing it may require access to a unique subset of information from the controller. In order to successfully monitor this, some kind of strict, information-access policy need to be enforced. This ensures that an application declares which information it will need and is only able to access these. This could ensure that applications are not covertly stealing or using information from other applications. Authenticity must also be ensured, before an application is able to communicate with the controller.

### **3.5 Control-Data Plane Link Attacks**

Another key area in SDNs which presents opportunities for attackers would be the link between the control plane and the data plane. The OpenFlow specification defines use of TLS (Transport Layer Security) as optional, making this a weak-point and clearly susceptible to various attacks, i.e. man-in-the-middle attacks, black-hole attacks.

#### **Man-in-the-middle Attack**

A man-in-the-middle type attack takes place when a malicious node establishes itself between the controller and the data-paths residing on the data plane. Instead of directly forwarding the messages straight to the controller (or vice-versa), the 'man-in-the-middle' node is able to manipulate/ or inspect the contents of packets.

## Black-hole Attack

A black-hole type attack could also be performed, in which a node establishes itself in between a targeted device and the controller, and simply drops any packets it receives without forwarding them to the controller. This results in a breakdown of network communications and renders the services unavailable to legitimate users.

If an attacker does manage to establish itself as an intermediary between the control plane and data plane, it can potentially be devastating to the entire network. The man-in-the-middle type attack is a direct attack on the integrity of control messages between network devices in the data plane and the controller. An adversary can change control messages and shape the way the network is formed to a way advantageous to them. On the other hand, the black-hole type attack is a direct attack on the availability of the networks services. If all messages between network devices and the controller are not being forwarded by the malicious node, it will inevitably result in a breakdown in communication, with devices in the data plane unable to solicit the controller when necessary. This link between control and data plane is clearly a weak-point, and acts as a ripe attack platform for adversaries. It is therefore extremely important that it is secure before SDNs see widespread usage in production settings.

## 3.6 Eavesdropping Attacks

Adversaries attempting to gain illegitimate access to SDN networks or halt service availability may find it advantageous to eavesdrop (the act of illegitimately capturing and inspecting the packets flowing over a connection) on certain connections in the network. This may allow them to gather meaningful information which can then be used to carry out more intrusive attacks. Eavesdropping attacks have long been carried out in traditional network settings wireless architectures are particularly weak due to their over-the-air transmissions. However, in the context of SDN, eavesdropping can be carried out to inspect the packets traversing the link between control-data plane, and also exclusively at the data plane. At the data plane, discusses a ease-of-use 'listening' mode integrated into OpenFlow switches can be utilized by a malicious adversary (that has been able to compromise the switch) in order to inspect the packets transmitted by surrounding switches, allowing attackers to learn important control information. In a sense, eavesdropping carried out at the control and data plane is more of a passive attack and does not directly affect the availability, confidentiality or integrity of data. It does, however, empower attackers to carry out further attacks which compromise these security aspects.

In the context of eavesdropping carried out at the Application-Control Plane link, however, the confidentiality of sensitive information can be di-

rectly compromised. A malicious adversary can learn information pertaining to a particular user if they manage to eavesdrop on a connection transmitting sensitive application data. This makes eaves-dropping a particularly serious attack confidentiality must be ensured before critical applications carrying sensitive information can be deployed in an SDN-environment.

### 3.7 Side by Side Comparison of Attacks

The following table summarises the above investigation and allows us to see the layers of abstraction that each attack affects in the SDN-architecture, and the specific security aspects that they could potentially compromise.

Attack	Targeted SDN Layer	Affected Security Aspect		
		<i>Availab ility</i>	<i>Confidentia lity</i>	<i>Integrity</i>
Distributed Denial of Service	Control, Data	x		
Denial of Service	Control, Data	x		
Hijacked/ Rogue Controller	Control, Data, App	x	x	x
Malicious Applications	App		x	x
Man-in-the-middle	Control, Data, Control-data link		x	x
Black-hole	Control, Data, Control-data link	x	x	
Eavesdropping	Control, Data, App		x	

Figure 3.1: Comparison of SDN Attack Types

### 3.8 Literature Survey

The previous section defined some of the key threats to both the control and data planes in the context of SDN environments. The separation of these planes leads to highly configurable networks. However it also introduces the possibility of a number of security threats. This section describes some previous works in this field.

Seungwon Shin et al. [10] introduce a solution for TCP based control plane DoS attacks **AVANT-GUARD**. This solution consists of two components; a Connection Migration mechanism used in establishing useful TCP sessions from failed ones, and Actuating Triggers which enable data plane devices to activate flow rules under certain pre-defined conditions. Connection Migration proxies the TCP handshake that takes place when nodes initiate a TCP connection, and ensures that the handshake is successfully completed and the session established before allowing any flow entries pertaining to this session to be forwarded to the controller. This reduces the possibility of TCP-SYN packet flooding attacks on the controller as the handshake will not have been completed for these sessions.

The Actuating Triggers mechanism introduced in [10] also reduces the computational load incurred by the controller, by allowing devices to activate certain flow rules in their tables under predefined conditions. This reduces the number of transmitted flow-requests. Evaluative tests prove that in the presence of a TCP-SYN based DDoS attack, the response time of the controller to legitimate flow requests increases by a negligible amount in the order of milliseconds, along with the percentage of overhead incurred. Whilst the performance of AVANT-GUARD is desirable, the approach is generally limited due to the fact that it targets one particular variation of the DDoS attack (TCP-SYN based attacks). An option would be to deploy this solution alongside other DDoS prevention mechanisms targeting other attack variations, however this configuration could become cumbersome and resource intensive. Instead, a solution defending the control plane from a wide variety of DDoS attack types would be more desirable.

Presented in [11] is **FlowRanger**, a proposal described as a request prioritizing algorithm for control plane-based DoS attacks. At a fundamental level, FlowRanger implements a priority-based scheduling system, with its key metric being that of trust-values held by each node in a network. Controllers implementing FlowRanger are able to evaluate the trust values. of each node they are receiving requests from and buffer them in separate priority queues [11]. This results in the dropping of some legitimate requests. This is clearly an issue; in production environments it is not acceptable for legitimate flow-requests to be dropped. **FlowRanger** challenges this with its priority queueing mechanism; suspected attacking requests will still be served, albeit at a lower priority than others. This works well, for example, if a legitimate switch is having issues and is having to retransmit flow-requests to the controller. It will be initially buffered in a lower priority queue, but eventually the request will be served. This means that once the switch has established itself and replenished its flow-cache with correct rules, its trust value will increase and its requests will return to higher priority queues. FlowRanger differs from previous implementation in that its priority queues are implemented at the controller, instead of in a distributed manner. This provides a further layer of protection (as long as the controller is not compromised).

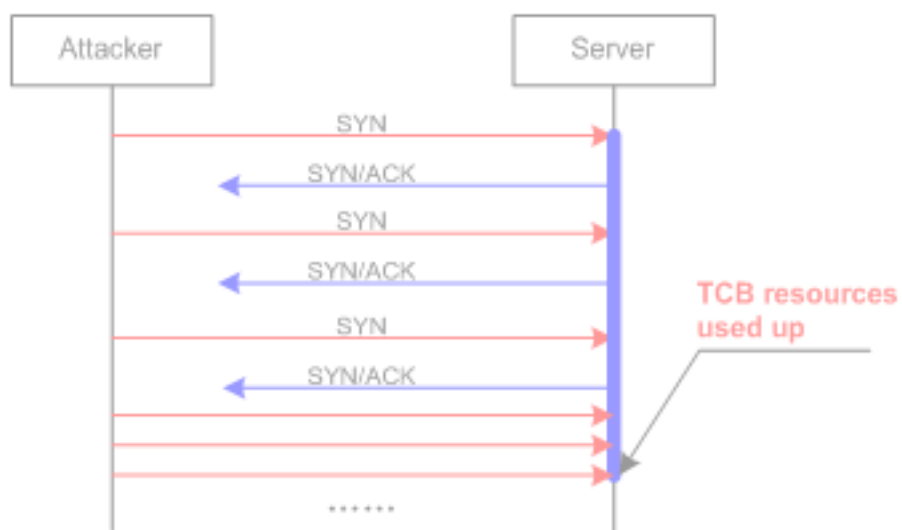
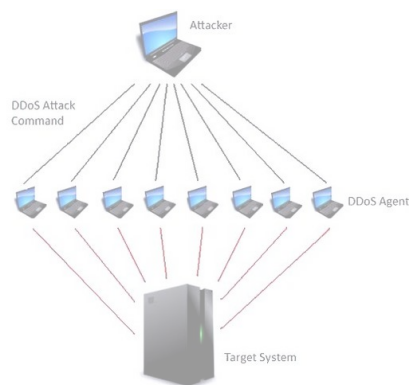


Paulo Fonseca et al. [12] present a novel mechanism to prevent control plane based DoS attacks - **CPRecovery**. This replication based component allows for the handing-over of control from one controller, failing due to the presence of a saturation attack, to a secondary controller. To achieve this, switches check for the presence of a properly operating controller by sending an inactivity probe. If this probe determines inactivity in the controller, a connection is made to a secondary controller which assumes the role of the failed one. To ensure this process is seamless, the initial primary controller sends state update messages to the secondary controllers in the network [12]. This multi-layer approach does provide resilience, and unlike AVANT-GUARD [10], it provides the control plane with protection from a wide variety of attacks. It is, however, noted in that once the secondary controller assumes a primary role, it is then susceptible to DoS attacks itself. One further downside to CPRecovery is its performance. Due to the overhead incurred from the instantiation and connection of recovery controllers, CPRecovery can be quite slow [12], with response times far higher than that of AVANT-GUARD. It would appear that this is a trade-off for the more thorough protection offered in.

References [10], [11], [12] all focus on control plane-based DoS/DDoS attacks. At the current time, there is a small number of published papers having proposed methods on DoS attack detection in SDN.

Indicatively, in [13], the authors present a low-overhead technique for traffic analysis using Self Organizing Maps to classify flows. This mechanism is deployed on an SDN (NOX-controlled [13]) network to enable DoS attack detection.

# Chapter 4 : Scope of Work



## *Scope of Work*

Most internet-based services rely on the Transmission Control Protocol (**TCP**). The **OpenFlow** protocol is also layered on top of the Transmission Control Protocol (TCP), and prescribes the use of Transport Layer Security (**TLS**). ONF defines OpenFlow as the first standard communications interface defined between the control and forwarding layers of an SDN architecture. OpenFlow allows direct access to and manipulation of the forwarding plane of network devices such as switches and routers, both physical and virtual (hypervisor-based).

Establishment of TCP connections is based on a handshake, more specifically a three way handshake. Exchange of three packets, to reserve and announce suitable resources at both ends before data exchange can proceed. This mechanism has however proven to be quite vulnerable to attacks. A denial-of-service (DoS) attack has, as an objective, to stop legitimate users from using a service. A distributed DoS (DDoS) attack extends the concept to a large number of attacking nodes. The synchronize (SYN) flooding attack submerges the victim with traffic pretending to open a new TCP connection, thus abusing the handshake mechanism.

### **4.1 Problem Statement**

Due to the centralized nature of SDN, DOS or DDOS SYN Flooding attack becomes much easier. So, the SYN Flooding attack needs to be detected and mitigated.

### **4.2 Assumptions**

- SDN generally contains single controller. It is the application that acts as strategic control point in the SDN network, manages flow control to the switches/routers and the applications and business logic to deploy intelligent networks.
- Controller manages the network and also saves the states of the network. So in case of network or node failure saved states are analyzed.

- Various types of DOS attack exist like UDP Flood, ICMP (Ping) Flood, TCP SYN Flood, Ping of Death, HTTP Flood etc. Here only TCP SYN-Flooding attack is considered.

## 4.3 Proposed Framework

In order to detect the SYN Flooding attack from victim's side, first classify different forms of TCP handshakes. Then identify the unusual TCP handshake sequences that result from an attack. A spacial data structure is introduced to observe, in real time, the state of the TCP handshake and study its performance. In addition, the data structure is briefly explained for the management of the operations such as initialization, adding and removing flows. Finally, effectiveness of the TCP handshake monitoring system is analyzed to identify the presence of SYN flooding attacks by applying it to real traffic traces. This detection system uses **Exponentially Weighted Moving Average (EWMA)** To compute latency period between server and client, which helps to calculate the threshold value, used to identify attack situation.

### 4.3.1 Usual and Unusual TCP handshakes

To establish a TCP connection, a client sends a SYN packet to the server. The server, in turn, transmits to the client a SYN/ACK packet as an acknowledgement of the reception of the SYN packet. When the client receives the SYN/ ACK packet of the server, it sends to the server an ACK packet in acknowledgement. At that point, the data transfer can begin.

In the handshake, the state in which the server waits for the ACK packet of a client is called **half-open**. In this state, the server has prepared the communication with a client by assigning a buffer for the connection. At a server, the number of half-open connections is limited.

The number of connections that can be maintained in a half-open state is controlled by a **backlog queue**. Packets that arrive at the server and exceed its capacity are simply rejected, and the server sends an **RST** packet to inform the relevant client that the SYN packet was thrown out. A connection that stays too long in a half-open state is timed-out, and an **RST** packet is sent to the client. As most cases of DoS attack are targeted toward servers, we restrict this presentation to a client-server use of TCP. To establish a TCP connection, a client sends a SYN packet to the server. In turn, the server transmits to the client a SYN/ACK packet as an acknowledgement of the reception of the SYN packet. When the client receives the SYN/ACK packet from the server, it replies with an ACK packet. At that point, the data transfer can begin.

The number of connections that can be maintained in a half-open state is controlled by a backlog queue. SYN packets that arrive at the server and exceed its capacity are simply rejected, and the server sends an **RST** packet to inform the relevant client that the SYN packet was thrown out. A connection

that stays too long in a half-open state is timed-out: an RST packet is sent to the client, and the buffer can be reused.

**Unusual** TCP handshakes are like attackers send to the server a SYN packet with a forged (spoofed) address. The server receiving these SYN packets sends a SYN/ACK packet to the spoofed address 4.1. It will then never receive the final ACK packet, which would have completed the handshake, because the SYN/ACK packet was sent to the machine whose address was spoofed, which ignored it. In the best case, the server may receive an RST packet from that machine.

When the backlog queue is filled with such SYN packets, the server cannot accept any SYN packet from legitimate users trying to connect. To terminate these waiting connections, the server has a timer to purge from the backlog queue all half-open connections that exceed the timeout value.

As the attack packets used in the SYN flooding are not different from normal packets, except for the spoofed address source, it is difficult to distinguish them from any normal packet intended for the victim server. This is why a SYN flooding attack is difficult to detect.

Legitimate handshakes and attack handshakes look exactly the same, except for the non-completion of the three steps and the frequent use of a spoofing address. Thus, different forms of TCP handshake are analyzed for SYN Flooding attack detection.

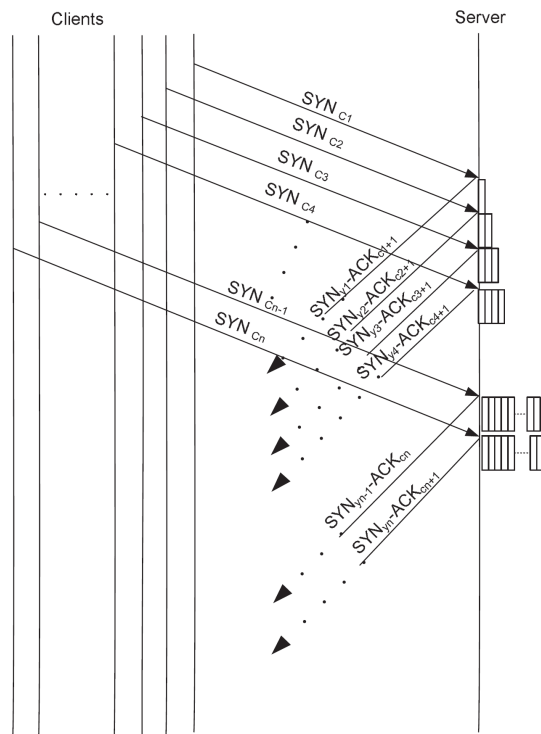


Figure 4.1: SYN Flooding Attack

## 4.4 Behaviour of TCP Handshakes

The analysis of packet traces containing no flooding attack shows that there are TCP handshakes whose sequence does not follow the three standard steps. We will name them unusual handshakes. Normally, those are the result of the network congestion and sometimes router errors and unreachable port; but during a DDoS attack, they can also be the result of the attack. To detect TCP handshake anomalies, it is necessary to identify the various TCP handshake sequences that can be a part of a SYN flooding attack. We look at the handshakes from the server side, observed at the last mile router. In the following list, all the different forms of TCP handshakes are listed, that can be usual or unusual, and analyse the matching message sequences. The unusual handshake sequences can be part of a DoS attack on a victim server.

- **SYN** : The server receives a SYN packet, but it cannot answer any more because it is overwhelmed. This connection will be ended after server time-out, as described earlier. 4.2
- **SYN (Client, Server), RST (Client, Server)** : This handshake sequence cannot be identified as part of a DoS attack because the client decided to terminate the connection request. 4.3
- **SYN (Client, Server), RST (Server, Client)** : This sequence probably means that the server is the victim of a DoS attack because it cannot reply to the legitimate client any more. 4.4
- **SYN, SYN/ACK** : The server waits indefinitely for the ACK packet, either because the IP source address is spoofed or because the ACK packet is rejected because of network congestion. This sequence can correspond to a DoS attack. This connection will be ended after server time-out. 4.5
- **SYN, SYN/ACK, RST** : This handshake sequence can correspond to a DDoS attack. At the reception of the SYN/ACK packet, the client host then transmits an RST packet to the server because it never sent a SYN packet. 4.6
- **SYN, SYN/ACK, ACK** : This connection is a complete three-way handshake sequence. 4.7

At the last mile router, four cases of unusual handshakes can be identified as DDoS attacks: (SYN, d), (SYN (Client, Server), RST (Server, Client)), (SYN, SYN/ACK, d), and (SYN, SYN/ACK, RST) where d corresponds to a (time-out) delay. d could be equal to the time-out of the server on the connection, but as we may not want to wait that long, we show below in Section 3.3 how separately holding **RTT (Round Trip time)** information on flows helps keep a tighter estimate of d.

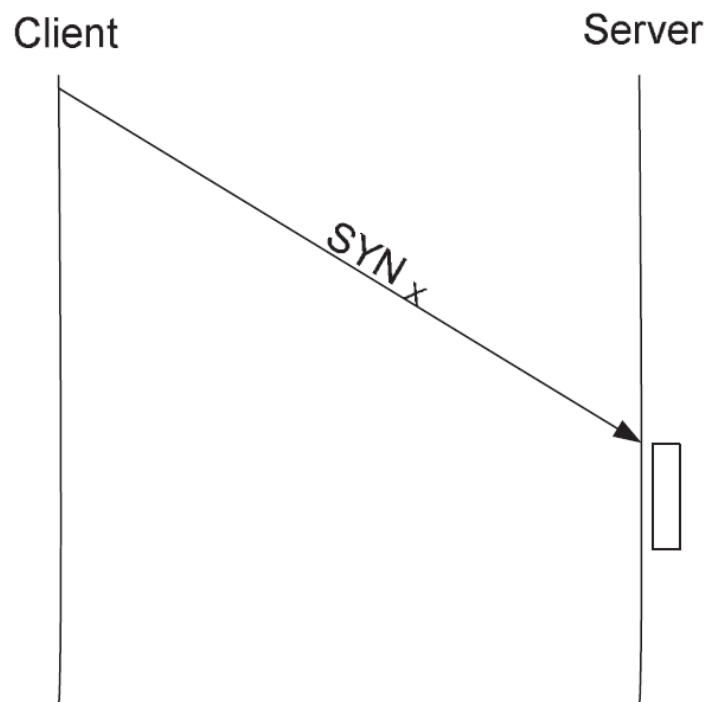


Figure 4.2: [SYN]

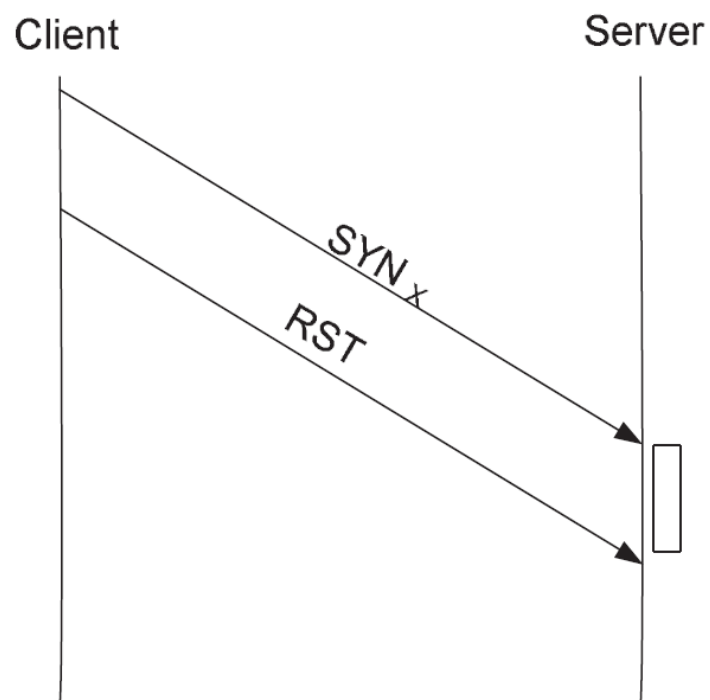


Figure 4.3: [SYN (Client, Server), RST (Client, Server)]

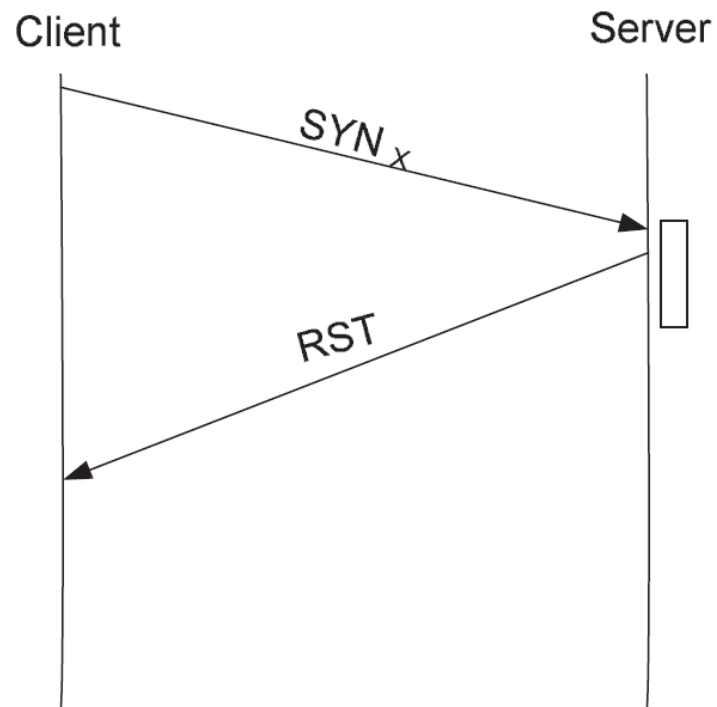


Figure 4.4: [SYN (Client, Server), RST (Server, Client)]

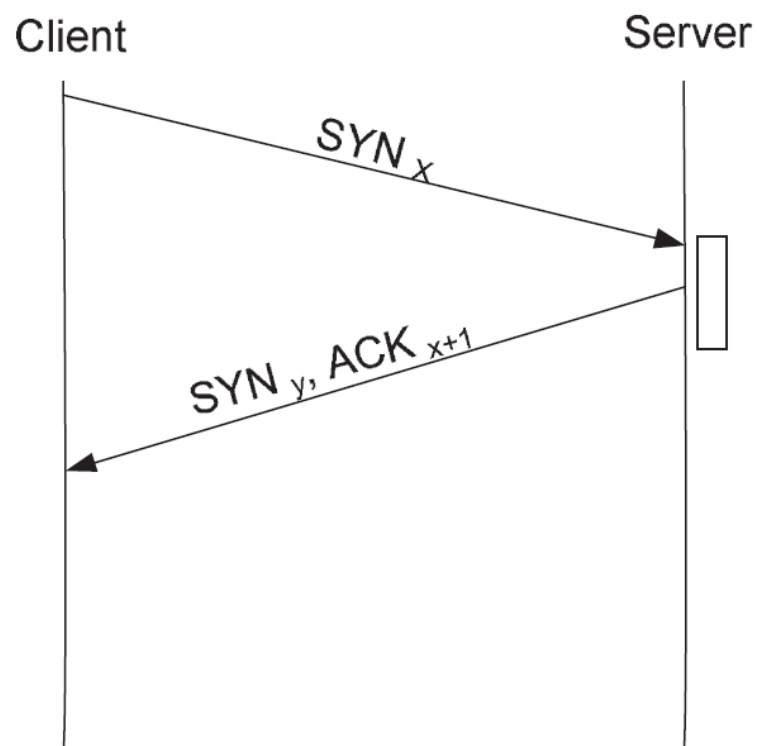


Figure 4.5: [SYN, SYN/ACK]



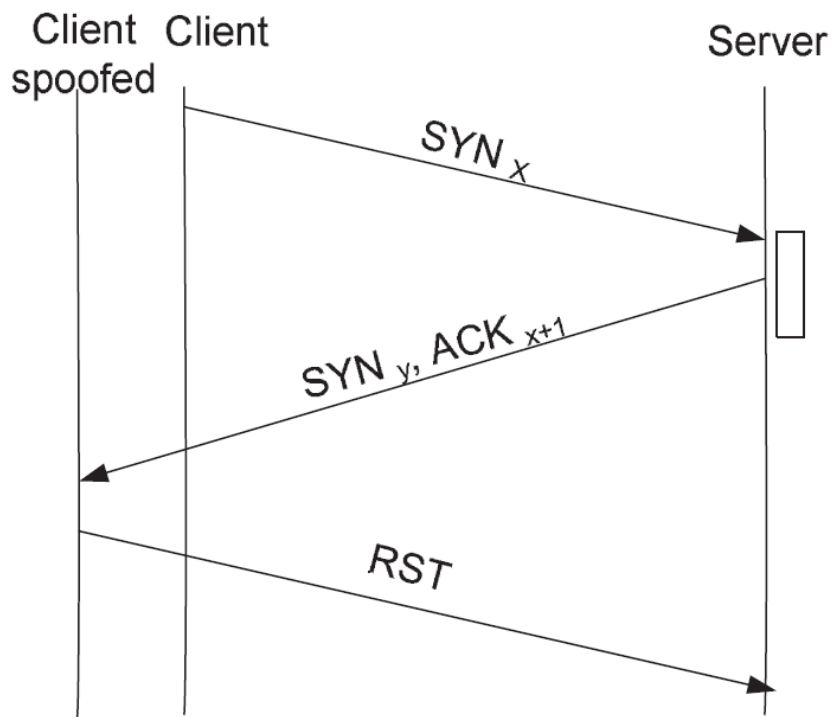


Figure 4.6: [SYN, SYN/ACK, RST]

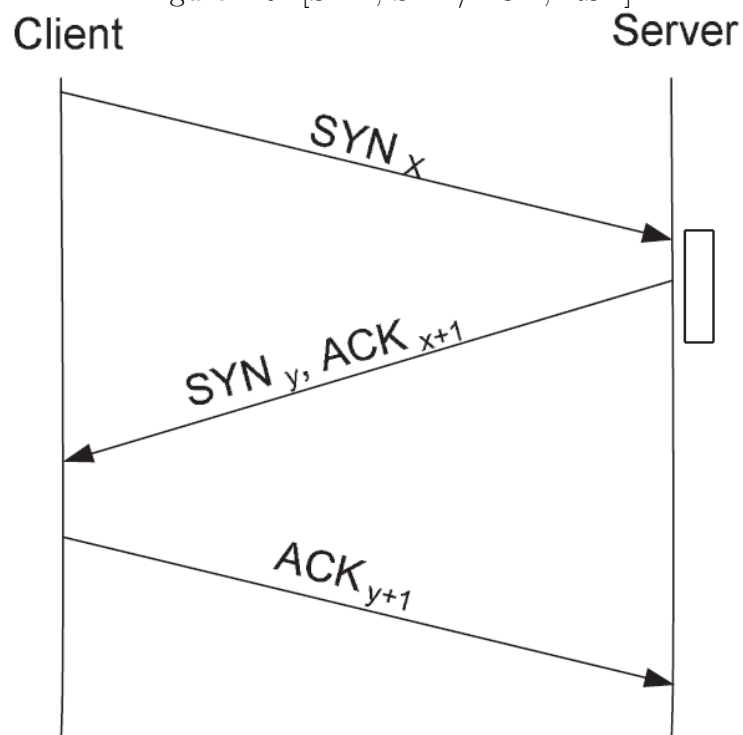


Figure 4.7: [SYN, SYN/ACK, ACK]

## 4.5 TCP Latency

TCP latency value is used to set a time-out. This value is used to signal a step of a TCP connection set-up that was not completed and exposed an unusual handshake. The ideal case would be to monitor each flow to use a value that reflects its RTT. There are many alternatives for computing latency.

- The first alternative is to set it to the time-out of the server. If we choose this alternative, we have fewer chances to make a mistake on the identification of an unusual handshake. This, however, takes a long time recall that TCP uses an exponential backoff scheme for retransmissions.
- The second alternative is the average connection time, measured between the reception of the SYN packet and the reception of the matching ACK packet, over usual handshakes and filtered through an EWMA. This connection time  $T_{\text{connection}}$  represents two different elements: the processing time of the server and the transmission time. In fact, the connection time is the sum of the time for reading the packet, allocating the TCP data structure and sending the SYN/ACK  $T_m$  and the RTT time  $TRTT$  between the transmission of the SYN/ACK packet and the reception of the ACK packet (see Figure 2). Update of this value is performed after a normal handshake is completed, with a proviso for retransmissions. Note that  $T_m$  reflects the load of the server, which varies independently of the RTT, and a more precise estimate could be built by monitoring  $T_m$  and  $TRTT$  separately.
- The third alternative improves on the previous ones. We keep and update the connection time for all source IP addresses of completed handshakes, rather than keeping a global connection time as in the second alternative. This connection time is stored together with the TCP information, and this allows us to take into account the varying distances and degrees of congestion between source networks and the server network. The update is performed in real time after a normal handshake. Furthermore, because we can safely assume that all machines in the same network will encounter similar network conditions, we can build a single estimate for all flows per source network, rather than per flow.

Here the third alternatives is used to compute latency value. 4.8

## 4.6 An Architecture for DoS Detection

This detection method supervises the TCP handshakes behaviour. The detection technique architecture is broken up into three modules: the collection module, the decision module and the monitoring module. Again, we assume

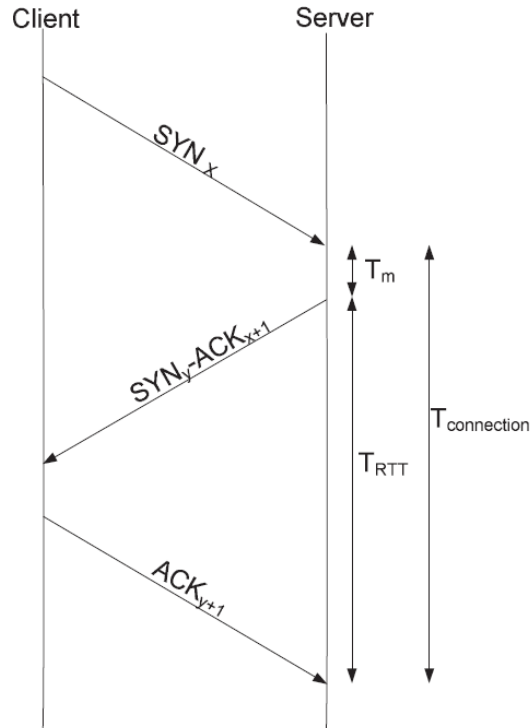


Figure 4.8: TCP Connection Setup Latency

that detection is done at the edge of the victims network or possibly at the last mile router.

The collection module gathers all the information necessary in a data structure for the detection technique and organizes them according to the detection tactics. The decision module has the methods necessary for the identification of an attack. It consults the information collected by the collection process and, according to the method chosen, makes the decision to report an attack if necessary. The monitoring module executes the decision module, typically after the expiration of the observation period.

#### 4.6.1 Collection Module

This module keep information relative to the beginning of a new TCP connection to distinguish normal from abnormal situations. Two important pieces of information are considered for identifying and counting unusual handshake sequences:

- Per source network, an estimate of the TCP connection latency (RTT, plus delay for memory allocation for the TCP data structure), according to the third alternative the most complex to keep a lower bound on the detection delay for the unusual handshake.

- The TCP handshake information: At the edge router, this can collect all SYN, SYN/ACK, ACK and RST packets and install a collection module that monitors the behaviour of the TCP handshake at the victims network. To monitor the TCP handshakes, we classify all TCP sequences after the expiration observation period. The value of this timer depends on the SYN rate of the traffic.

We keep this information in a single table, indexed, by the 24-bit prefix of source IP addresses, that is the source network address. The use of such an index is legitimate because 24-bit network addresses dominate other addresses in Internet routing. In earlier Section, already explained the alternatives for the TCP connection latency estimation and presented a metric to help us make the better choice. It is described here a structure for the third alternative, as it is the most resource demanding. In next Section, we show how handshake information is kept and analyse its memory requirements.

#### 4.6.1.1 TCP Handshake Information

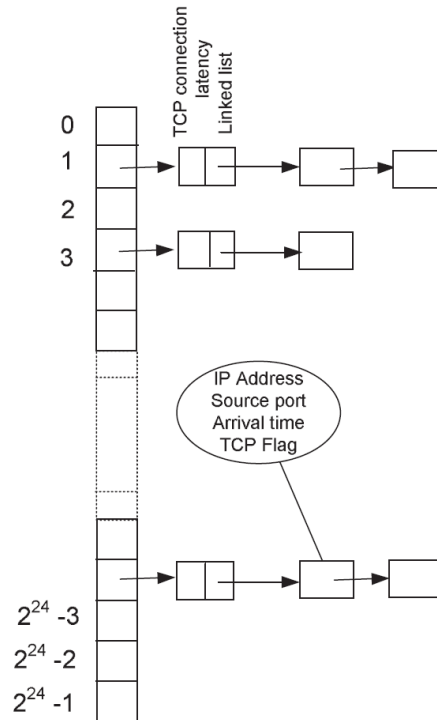


Figure 4.9: Data Structure for TCP Handshake Information

Classify IP packets by filtering them to select only TCP SYN, TCP ACK and TCP RST packets. Each entry in the table contains a 4-byte field for a pointer. When a new 24-bit network address appears, we allocate an 8-byte

structure. The first 4-byte field is the estimate measured in milliseconds of TCP connection latency, and the second 4-byte field is a pointer to a linked list of the flows originating from the source network. As we know which service we want to protect against flooding, it is not necessary to keep the destination address and the port. For each new TCP flow in the linked list, we store the following information, which occupies 8 bytes of memory:

- One byte for the last byte of IP source address and 2 bytes for the source port.
- Four bytes for arrival time of the last SYN packet of a new TCP flow.
- One byte for the flag of the TCP packet: bit 0: SYN, bit 1: SYN/ACK, bit 2: ACK, bit 4: RST, (server, client), bit 8: RST (client, server).

Each entry of the table points to a linked list of all the TCP flows originating from the network matching the entry's index. Each record of the linked list occupies 12 bytes (8 bytes of data as shown earlier plus 4 bytes for the pointer).

Certainly, the detection time depends on the duration of an observation period. If the evaluation period is very small, we can detect the attack more quickly and apply appropriate defence, albeit at the cost of a greater overhead. We have chosen time observations of 10 s in the analysis presented later in this paper. Note that this duration guarantees that most flows will be categorized as unusual or usual within a single period.

Certainly, the attacker can try to send a flood of SYN packets followed by ACK packets to keep a reasonable balance of SYN versus non-SYN packets. Thus, the number of unusual handshakes will be wrong. To avoid that, it could also keep track of TCP sequence numbers. The server, however, would quite likely reset the connection because of wrong sequence numbers.

## 4.6.2 Management of Data Structure

In this section, we explain the mechanism of the data structure update process, with TCP handshake information.

### 4.6.2.1 Initialization

First, create a table of size  $2^{24}$  slots, each initialized with a null pointer. At the first time that an entry is created for a slot, the latency is initialized with a negative value. When a legitimate handshake is completed for an entry in the slot and the latency is still negative, we update the latency of this slot with the legitimate handshake latency. The alternative (to a negative value) could be the time-out value of the server, but this is actually not adequate because this value is quite high and would create a significant bias in the computation of the **EWMA**. When the structure is reset between evaluation periods, then keep the time-out estimates from the previous round.

#### 4.6.2.2 Adding Packets

Now four packets types are processed: SYN, SYN/ACK, ACK and RST and compute the index from the source address.

- In case of a **SYN** packet, we find the matching slot in the table. We have two alternatives:
  1. If the slot is empty, we allocate the structure to hold the latency and a pointer for the (empty) linked list of the TCP handshake information. Then, add the new flow to the linked list.
  2. If the slot is not empty, scan the linked list to find the matching handshake information. If it is found, just update the SYN packet information; if not, just add a new flow in the linked list of the matching slot.
- In case of a **SYN/ACK** packet, try to find the corresponding TCP handshake and update the information. If no matching entry found, just reject the packet.
- In case of an ACK packet, find the corresponding TCP handshake and update the TCP handshake information. Thus far, we are in presence of a normal handshake, so update the latency for this slot. The **latency** of this normal handshake is the difference between the arrival time of the SYN and ACK packets. If the latency is still negative, that is it is the first time update it, simply replace the previous value with the computed latency. Otherwise, update the field with the **EWMA**-computed latency between the current value and the newly computed one.
- In case of an RST packet, we have to verify whether it originates from the client side or the server side to update the status of the flow accordingly.

#### 4.6.2.3 Removing a Flow

At the expiration of an observation period, we have to identify in the data structure the usual and unusual TCP handshakes. Scan all non-empty entries in the table. We have three cases to consider.

1. **Usual handshakes:** Remove sequences of the form [SYN, RST (client, server)] and [SYN, SYN/ ACK, ACK].
2. **Non time-out unusual handshakes:** Delete packets without any latency verification the sequences [SYN, RST(server, client)] and [SYN,SYN/ACK, RST] because these sequences were properly closed with the last RST packet.

3. **Time-out detected unusual handshakes:** For the sequences [SYN] and [SYN, SYN/ACK], if the arrival time of the SYN packet added to the latency of the corresponding table entry is less than the current time, that is a time-out, just delete this unusual handshake; otherwise, we must wait for the next period and check again. Note that if the latency is still negative, we can use the time-out value of the server as a substitute.

## 4.7 Decision Module

The decision module uses an **ratio** measure derived from the number of usual and unusual handshake sequences for detecting attacks. Under normal traffic, the ratio value runs lesser than the threshold, but during SYN flooding attacks, it is shown that the change in the number of unusual handshakes will cause a variation in this value, bringing it higher. Note that, beyond SYN flooding attacks, another possible cause of variation in the ratio measure would be a simultaneous increase in the congestion of the different transit networks. It is however extremely unlikely that such events would occur simultaneously.

To calculate the this ratio, four forms of unusual handshakes are chosen:

- p0 corresponds to the fraction of normal usual packets.
- p1 corresponds to the fraction of the (SYN, d) handshake sequences over all unusual handshakes.
- p2 corresponds to the fraction of the (SYN (Client, Server), RST (Server, Client)) handshake sequences over all unusual handshakes.
- p3 corresponds to the fraction of the (SYN, SYN/ACK, d) handshake sequences over all unusual handshakes.
- p4 corresponds to the fraction of the (SYN, SYN/ACK, RST (Client, Server)) handshake sequences over all unusual handshakes.

### 4.7.1 Detection

First a threshold value is evaluated, that is, the bound T of the ratio to distinguish packet without attack. For the detection, we could simply observe **threshold violations** but, it is preferred to use **Repeated Threshold Violations (RTV)** which are better to avoid false detections.

- **Offline Analysis** First this data structure is applied on several normal, and attack traffics packet to measure the threshold value. For every case it is observed that, in case of usual traffic this value is very low. But for unusual traffic this value is higher. Depending on these scenario a threshold value is evaluated. After that this threshold value is used to differentiate usual and unusual traffics.

- Online Analysis Packets are processed in a time interval. These data are fed into this data structure, and depending on the threshold value it classifies the scenario. For attack scenario it works on safe mode. It is assumed that in safe mode it will allow only specific trustable IPs.

## 4.8 Experimental Results

### Normal Traffic

File size : 154.4 kB

Execution Time: — 0.04425787925720215 seconds —

Threshold Violation = 0 times

Total No of packets : 1277

### Attack Traffic File size : 1.3 MB

Execution Time: — 7.427768230438232 seconds —

Threshold Violation = 7 times

Total No of packets : 9200

2

Ratio = 0.0

4

Ratio = 0.5936826992103375

Potential Attack Detected

Switching into safe mode

6

Ratio = 0.6223456386801699

Potential Attack Detected

Switching into safe mode

8

Ratio = 0.9166332665330661

Potential Attack Detected

Switching into safe mode

10

Ratio = 1.0

Potential Attack Detected

Switching into safe mode

12

Ratio = 0.3286794648051193



Potential Attack Detected  
Switching into safe mode  
14

Ratio = 0.9154308617234469  
Potential Attack Detected  
Switching into safe mode  
16

Ratio = 1.0  
Potential Attack Detected  
Switching into safe mode  
18  
20  
'192.168.0': [-1.0][8]  
22  
7  
— 7.427768230438232 seconds —

### Normal Traffic

File size : 4.2 kB  
Execution Time: — 0.003081083297729492 —  
Threshold Violation = 1 times  
Total No of packets : 33

Ratio = 0.0  
2  
4  
Ratio = 0.0  
6  
'192.168.0' : [0.26559679525, 8, 35660, 6.501399743, *SYN*, *SYN\_ACK*, *ACK*,  
9, 35660, 6.003735017, *SYN*]  
Ratio = 0.5  
Potential Attack Detected  
Switching into safe mode  
8  
1  
— 0.003081083297729492 seconds —

These files are captured by **wireshark**, is a free and open source packet analyzer. In order to get normal packets a web crawling code is used. In order to generate attack a attacking tool **hping3** is used.

## 4.9 Conclusion

SDN is a promising platform which has been extensively used in research settings due to its highly configurable nature. In this document a SYN Flooding attack detection framework is proposed. It can detect only SYN Flooding attacks in a proper way. The threshold value for this framework must be chosen carefully. Since the data structure is organized by source networks , after detecting the attack the unusual source IPs can be blocked.

This model can be improved for other types of attack scenario. The prevention part can be improved by blocking the unusual source IPs. The memory model of this framework can be improved.

This framework takes reasonable time to detect attack. So it can be used in controller as a plug-in module. Another good feature of this framework is, not only SDN, in can also detect SYN Flooding attack in traditional network.

# *Appendix*

## 5.1 Mininet

Platforms for Network/Systems: Network Emulator Architecture

Mininet: Basic Usage, CLI, API

Mininet already has the capability of creating containers as well as the network from a single, simple Python API. Using Mininet avoids the need for installing, configuring and administering multiple orchestration systems. Moreover, Mininet hosts do not run unnecessary extra software such as multiple kernels or daemons, and they do not require unwieldy, bulky virtual file system images. To support configurations that exceed the resources of a single server, Mininets experimental cluster support may be used to easily and seamlessly distribute the virtual testbed across multiple physical (or virtual) servers. By default, Mininet hosts are simply process groups connected to virtual Ethernet interfaces using Linuxs network namespace feature. (Since Mininet 2.2, private mount namespaces have also been enabled by default, though this may become optional in the future.)

## 5.2 Installation, Setup and Command

- Install Mininet
- Run the in VirtualBox
- For creating sample Mininet cluster:
  - Mininet Cluster Python Code:

```
topo = Tree(depth=3, fanout=3)
servers = ['localhost', 'server2', 'server3']
net = MininetCluster(topo=topo, servers=servers)
net.start()
CLI(net)
net.stop()
Shell Command
mn -topo tree,depth=3,fanout=3 -cluster localhost,server2,server3
```

- For simple Mininet without cluster the shell command is:  
mn -topo tree,depth=3,fanout=3

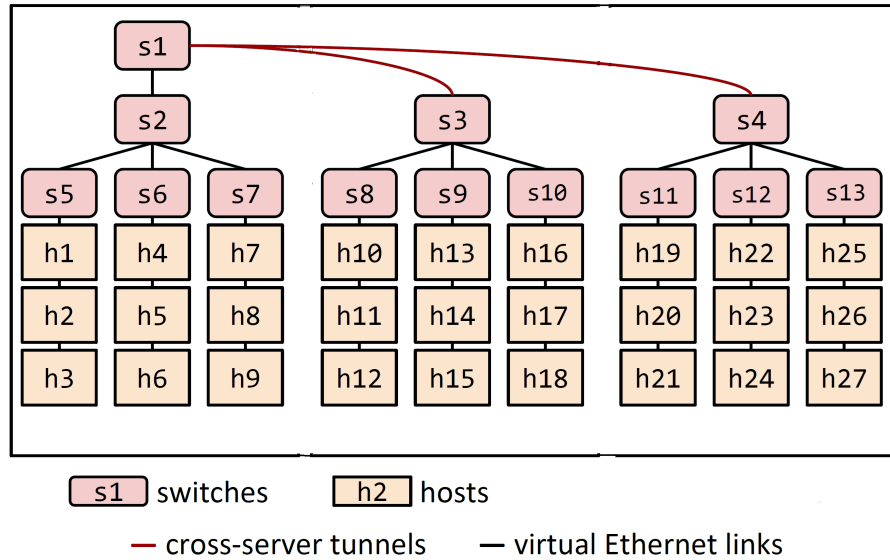


Figure 5.1:

```

[ Read 1 line ]
mininet@mininet-vm:~$ sudo mn --topo tree,depth=3,fanout=3
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25 h26 h27
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13
*** Adding links:
(s1, s2) (s1, s6) (s1, s10) (s2, s3) (s2, s4) (s2, s5) (s3, h1) (s3, h2) (s3, h3)
(s4, h4) (s4, h5) (s4, h6) (s5, h7) (s5, h8) (s5, h9) (s6, s7) (s6, s8) (s6, s9)
(s7, h10) (s7, h11) (s7, h12) (s8, h13) (s8, h14) (s8, h15) (s9, h16) (s9, h17)
(s9, h18) (s10, s11) (s10, s12) (s10, s13) (s11, h19) (s11, h20) (s11, h21) (s12, h22)
(s12, h23) (s12, h24) (s13, h25) (s13, h26) (s13, h27)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25 h26 h27
*** Starting controller
c0
*** Starting 13 switches
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 ...
*** Starting CLI:
mininet>

```

Figure 5.2: Screenshot of Mininet terminal

```

cloud4@cloud4: ~
cloud4@cloud4:~$ clear

cloud4@cloud4:~$ sudo mn
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
mininet> dump

```

Figure 5.3: Screenshot of default Mininet topology

## 5.3 command

- Dump information about all nodes

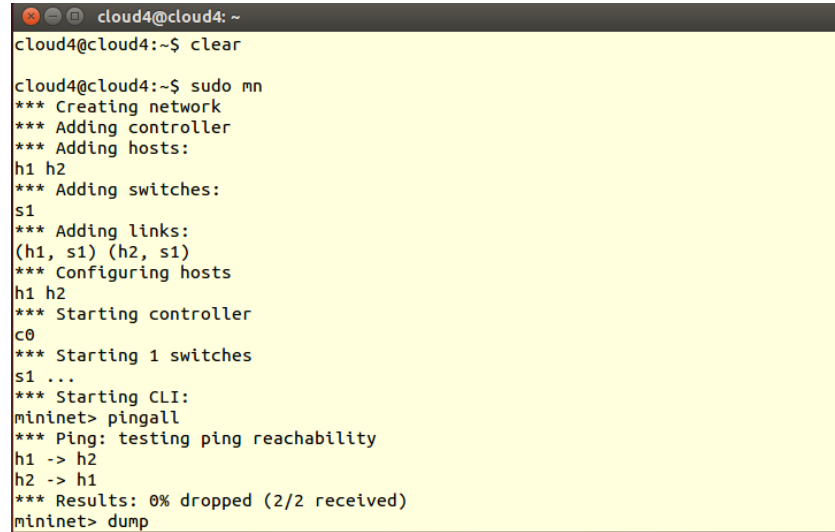
```

cloud4@cloud4: ~
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=4052>
<Host h2: h2-eth0:10.0.0.2 pid=4057>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=4062>
<OVSController c0: 127.0.0.1:6633 pid=4045>
mininet> miniedit
*** Unknown command: miniedit
mininet> exit
*** Stopping 1 controllers
c0
*** Stopping 2 links
..
*** Stopping 1 switches
s1
*** Stopping 2 hosts
h1 h2
*** Done
completed in 185.011 seconds
cloud4@cloud4:~$

```

Figure 5.4: Mininet dump command

- Display links : `minineti net`
- Ping : `ping : minineti h1 ping -c 1 h2` : `h1 ping h2 pingall` command : which does an all-pairs ping.

A terminal window titled 'cloud4@cloud4: ~' with a yellow background. The user enters 'clear' to reset the terminal. Then they run 'sudo mn' to start Mininet. The output shows the creation of a network with two hosts (h1, h2) and one switch (s1). The user then enters 'mininet> pingall' to test connectivity between all hosts. The output shows that the ping is successful for both directions (h1 to h2 and h2 to h1) with 0% dropped packets. Finally, the user enters 'mininet> dump' to show the network state.

```
cloud4@cloud4: ~  
cloud4@cloud4:~$ clear  
  
cloud4@cloud4:~$ sudo mn  
*** Creating network  
*** Adding controller  
*** Adding hosts:  
h1 h2  
*** Adding switches:  
s1  
*** Adding links:  
(h1, s1) (h2, s1)  
*** Configuring hosts  
h1 h2  
*** Starting controller  
c0  
*** Starting 1 switches  
s1 ...  
*** Starting CLI:  
mininet> pingall  
*** Ping: testing ping reachability  
h1 -> h2  
h2 -> h1  
*** Results: 0% dropped (2/2 received)  
mininet> dump
```

Figure 5.5: Mininet pingall command

## 5.4 controller

- Floodlight Controller The Floodlight Open SDN Controller is an enterprise-class, Apache-licensed, Java-based OpenFlow Controller. It is supported by a community of developers including a number of engineers from Big Switch Networks.

simulate Floodlight from terminal : `java -jar target/floodlight.jar`

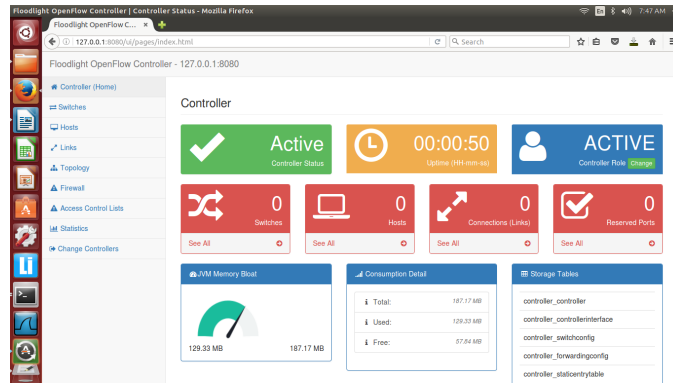


Figure 5.6: Floodlight controller without any topology

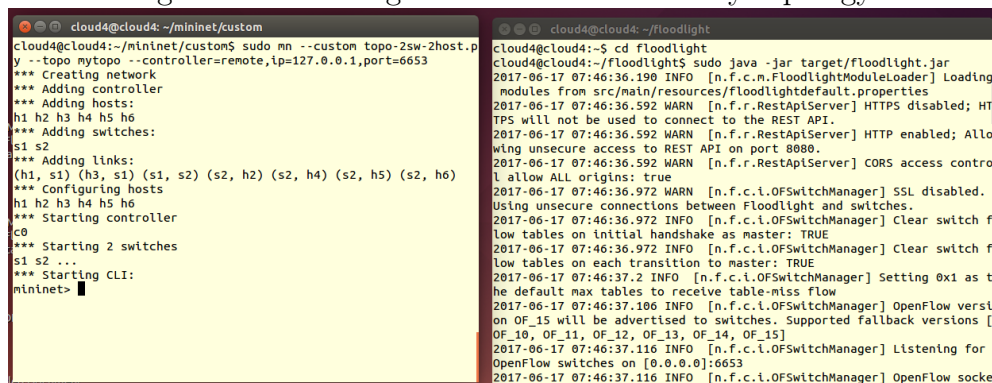


Figure 5.7: simulate a topology from floodlight controller

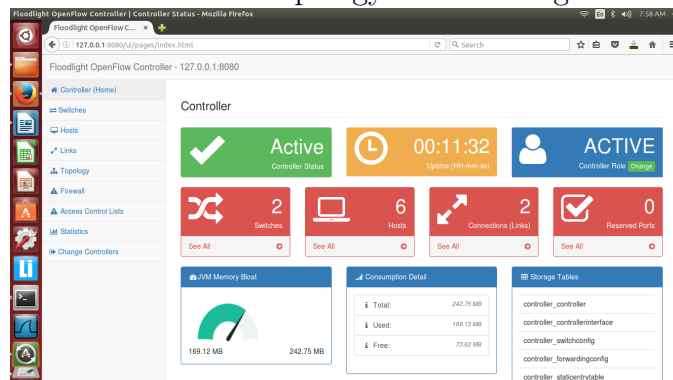


Figure 5.8: Floodlight controller with topology topo-2sw-2host

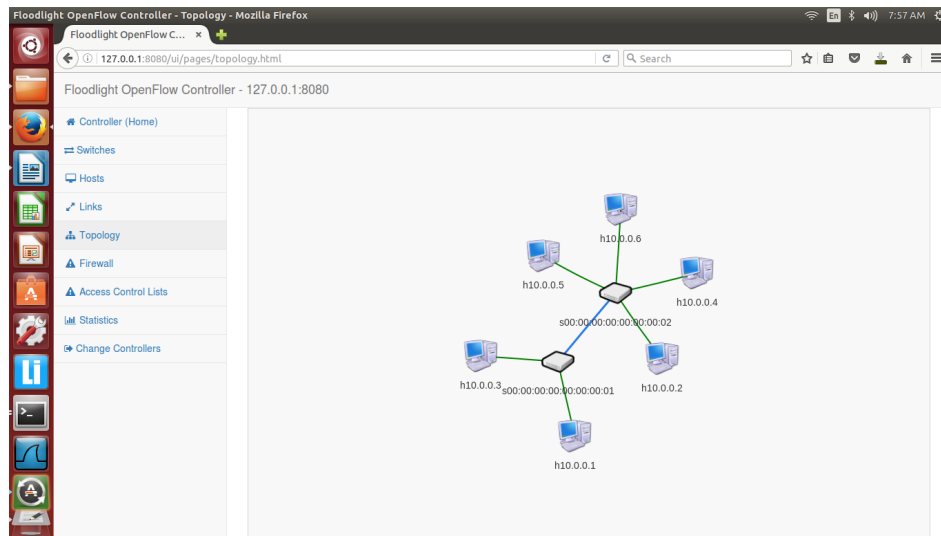


Figure 5.9: Floodlight controller with topology topo-2sw-2host

### topo-2sw-2host.py

Custom topology example

Two directly connected switches plus a host for each switch:

host — switch — switch — host

Adding the 'topos' dict with a key/value pair to generate our newly defined topology enables one to pass in '-topo=mytopo' from the command line.

```
from mininet.topo import Topo

class MyTopo( Topo ):
    "Simple topology example."

    def __init__( self ):
        "Create custom topo."

        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        leftHost1 = self.addHost( 'h1' )
        leftHost2 = self.addHost( 'h3' )
        rightHost1 = self.addHost( 'h2' )
        rightHost2 = self.addHost( 'h4' )
        rightHost3 = self.addHost( 'h5' )
        rightHost4 = self.addHost( 'h6' )
```



```
leftSwitch = self.addSwitch( 's3' )
rightSwitch = self.addSwitch( 's4' )

# Add links
self.addLink( leftHost1, leftSwitch )
            self.addLink( leftHost2,
                leftSwitch )
self.addLink( leftSwitch, rightSwitch )
self.addLink( rightSwitch, rightHost1 )
            self.addLink( rightSwitch,
                rightHost2 )
            self.addLink( rightSwitch,
                rightHost3 )
            self.addLink( rightSwitch,
                rightHost4 )

topos = { 'mytopo': ( lambda: MyTopo() ) }
```

- POX Controller POX is an open source development platform for Python-based software-defined networking (SDN) control applications, such as OpenFlow SDN controllers.

```
cloud4@cloud4: ~/pox-carp
cloud4@cloud4:~$ cd pox-carp/
cloud4@cloud4:~/pox-carp$ sudo python pox.py forwarding.l2_pairs info.
packet_dump samples.pretty_log log.level --DEBUG
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
Arjaita
INFO:forwarding.l2_pairs:Pair-Learning switch running.
INFO:info.packet_dump:Packet dumper running
[core] POX 0.2.0 (carp) going up...
[core] Running on CPython (2.7.6/Jun 22 2015 17:58:13)
[core] Platform is Linux-3.13.0-85-generic-x86_64-w
ith-Ubuntu-14.04-trusty
[core] POX 0.2.0 (carp) is up.
[openflow.of_01] Listening on 0.0.0.0:6633
```

Figure 5.10: POX controller with topology topo-2sw-2host

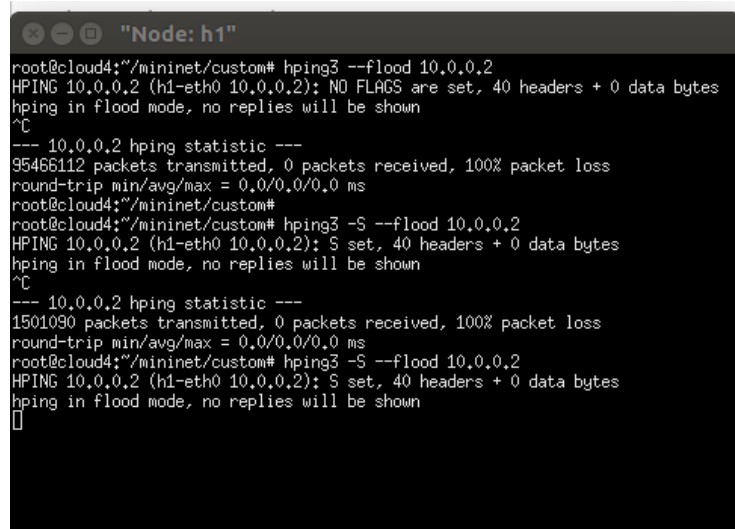
```
cloud4@cloud4: ~
cloud4@cloud4:~$ clear

cloud4@cloud4:~$ sudo mn --topo tree,2,2 --controller remote
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2 s3
*** Adding links:
(s1, s2) (s1, s3) (s2, h1) (s2, h2) (s3, h3) (s3, h4)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=13013>
<Host h2: h2-eth0:10.0.0.2 pid=13018>
<Host h3: h3-eth0:10.0.0.3 pid=13020>
<Host h4: h4-eth0:10.0.0.4 pid=13022>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=13027>
```

Figure 5.11: POX controller with topology

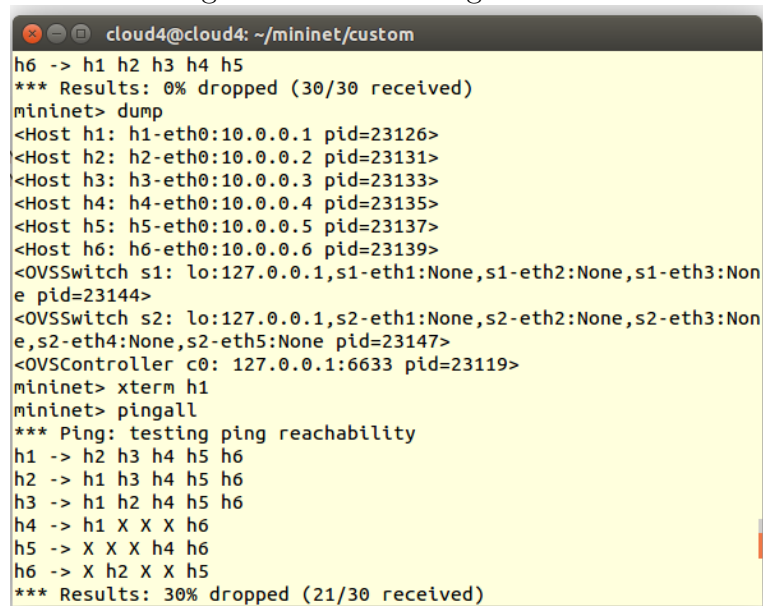
## 5.5 Generate Attack using hping3

hping3 is a free packet generator and analyzer for the TCP/IP protocol. Hping is one of the de-facto tools for security auditing and testing of firewalls and networks. [14]



```
root@cloud4:~/mininet/custom# hping3 --flood 10.0.0.2
HPING 10.0.0.2 (h1-eth0 10.0.0.2): NO FLAGS are set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
^C
--- 10.0.0.2 hping statistic ---
95466112 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
root@cloud4:~/mininet/custom# hping3 -S --flood 10.0.0.2
HPING 10.0.0.2 (h1-eth0 10.0.0.2): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
^C
--- 10.0.0.2 hping statistic ---
1501090 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
root@cloud4:~/mininet/custom# hping3 -S --flood 10.0.0.2
HPING 10.0.0.2 (h1-eth0 10.0.0.2): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
^
```

Figure 5.12: Attacking Node h1



```
cloud4@cloud4: ~/mininet/custom
h6 -> h1 h2 h3 h4 h5
*** Results: 0% dropped (30/30 received)
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=23126>
<Host h2: h2-eth0:10.0.0.2 pid=23131>
<Host h3: h3-eth0:10.0.0.3 pid=23133>
<Host h4: h4-eth0:10.0.0.4 pid=23135>
<Host h5: h5-eth0:10.0.0.5 pid=23137>
<Host h6: h6-eth0:10.0.0.6 pid=23139>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None pid=23144>
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None,s2-eth3:None,s2-eth4:None,s2-eth5:None pid=23147>
<OVSController c0: 127.0.0.1:6633 pid=23119>
mininet> xterm h1
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6
h2 -> h1 h3 h4 h5 h6
h3 -> h1 h2 h4 h5 h6
h4 -> h1 X X X h6
h5 -> X X X h4 h6
h6 -> X h2 X X h5
*** Results: 30% dropped (21/30 received)
```

Figure 5.13: Attack scenario in SDN

```
Terminal Terminal File Edit View Search Terminal Help
len=46 ip=192.168.0.1 ttl=30 id=11718 tos=0 iplen=40
sport=0 flags=RA seq=1 win=0 rtt=5.7 ms
seq=0 ack=810957402 sum=a1bc urp=0

len=46 ip=192.168.0.1 ttl=30 id=12524 tos=0 iplen=40
sport=0 flags=RA seq=13 win=0 rtt=2.7 ms
seq=0 ack=1662418959 sum=2044 urp=0

len=46 ip=192.168.0.1 ttl=30 id=12528 tos=0 iplen=40
sport=0 flags=RA seq=15 win=0 rtt=5.8 ms
seq=0 ack=1208052885 sum=ad72 urp=0

len=46 ip=192.168.0.1 ttl=30 id=12532 tos=0 iplen=40
sport=0 flags=RA seq=17 win=0 rtt=5.3 ms
seq=0 ack=239918316 sum=93c urp=0

len=46 ip=192.168.0.1 ttl=30 id=12288 tos=0 iplen=40
sport=0 flags=RA seq=22 win=0 rtt=3.5 ms
seq=0 ack=486149268 sum=aefd urp=0

len=46 ip=192.168.0.1 ttl=30 id=12546 tos=0 iplen=40
sport=0 flags=RA seq=23 win=0 rtt=3.2 ms
seq=0 ack=481969281 sum=c6 urp=0

len=46 ip=192.168.0.1 ttl=30 id=12548 tos=0 iplen=40
sport=0 flags=RA seq=24 win=0 rtt=2.9 ms
seq=0 ack=499787017 sum=67e4 urp=0

len=46 ip=192.168.0.1 ttl=30 id=12544 tos=0 iplen=40
sport=0 flags=RA seq=25 win=0 rtt=6.8 ms
seq=0 ack=1993777963 sum=380d urp=0

^C
-- 192.168.0.1 hping statistic --
26 packets transmitted, 9 packets received, 66% packet loss
round-trip min/avg/max = 2.7/4.7/6.8 ms
arjaita@arjaita-VirtualBox:~$

arjaita@arjaita-VirtualBox:~/Desktop
round-trip min/avg/max = 0.0/0.0/0.0 ms
arjaita@arjaita-VirtualBox:~/Desktop$
arjaita@arjaita-VirtualBox:~/Desktop$ sudo hping3 -S -p 80 --flood --rand-s
ource 192.168.0.1
HPING 192.168.0.1 (enp0s3 192.168.0.1): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
^C
-- 192.168.0.1 hping statistic --
245622 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
arjaita@arjaita-VirtualBox:~/Desktop$ clear
arjaita@arjaita-VirtualBox:~/Desktop$ sudo hping3 -S -p 80 --flood --rand-s
ource 192.168.0.1
HPING 192.168.0.1 (enp0s3 192.168.0.1): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
^C
-- 192.168.0.1 hping statistic --
631984 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
arjaita@arjaita-VirtualBox:~/Desktop$
```

Figure 5.14: Attack scenario in Normal Network

## 5.6 Python code for proposed framework

### Detect.py

---

```
from IPlist import IPlist
from math import log
import time
start_time = time.time()

detectionDelay = 2
attackCount = 0

def init():
    IPlist = IPlist()
    for currentTime in AddFlow(IPlist):
        CountAndDelete(IPlist, currentTime)
        print(currentTime)
    CountAndDelete(IPlist, currentTime+detectionDelay)
    print(currentTime+detectionDelay)

def AddFlow(IPlist):

    currentTime = 0

    try:
        file = open('test3.csv', 'r')
        for line in file.readlines():
            added = IPlist.add(line.strip(),
                               currentTime, detectionDelay)
            while(not added):
                currentTime += detectionDelay
                yield currentTime
            added = IPlist.add(line.strip(),
                               currentTime, detectionDelay)
    except IOError:
        print("not found")
    # print(SYNcount, PcktCount, IPlist)
    print(IPlist)

def CountAndDelete(IPlist, currentTime):
    removedSockets = IPlist.remove(currentTime)
```

```

    #calculate ratio
    ratio = 0
    sum = IPlist.p1 + IPlist.p2 + IPlist.p3 + IPlist.
        p4
    if IPlist.pcktCount > 0 :
        ratio = (sum/IPlist.pcktCount)
    #    print(IPlist.pcktCount, sum, percentage)
        print("Ratio_=", ratio)
    if ratio > 0.3 :
        print("Potential_Attack_Detected_\\nSwitching_
            into_safe_mode")
        #print(removedSockets)
        global attackCount
        attackCount += 1

init()
print(attackCount)
print("---_s_seconds_---" % (time.time() -
    start_time))

```

## IPlist.py

```
from PcktTypes import PcktTypes
from Servers import Servers
from TCPflow import TCPflow
from TCPLatency import TCPLatency

class IPlist:
    def __init__(self):
        self.servers = Servers()
        self.IP = dict() #2^24
        self.p0 = self.p1 = self.p2 = self.p3 = self.
            p4 = 0
        self.serverTimeOut = 1
        self.pcktCount = 0

    def add(self, packet, currentTime, detectionDelay
):
        tokens = packet.split(',')
        tokens[6] = ','.join(str for str in tokens
            [6:])
        if tokens[6].find('>') < 0: #no port = not
            TPC packet, skip
            return True
        srcIP = tokens[2][1:-1]
        destIP = tokens[3][1:-1]
        srcPort = tokens[6].split('>')[0].strip()[1:]
        destPort = tokens[6].split('>')[1].strip().
            split(' ')[0]
        arrivalTime = float(tokens[1][1:-1])
        if(arrivalTime > currentTime+detectionDelay):
            return False
        # self.pcktCount += 1
        prefix = ','.join(str for str in srcIP.split(
            '.')[0:3])
        suffix = srcIP.split('.')[3]
        if tokens[6].find('[SYN]') >= 0:
            #add to the list
            if prefix in self.IP:
                #search for src socket
                for flow in self.IP[prefix][1:]:
```

```

        if flow.srcIPsuffix == suffix and
            flow.srcPort == srcPort:
            #update the flow (overwrite)
            flow.flags = PcktTypes.SYN
            flow.arrivalTime =
                arrivalTime
            break
    else:
        flow = TCPflow(suffix, srcPort,
            arrivalTime, PcktTypes.SYN)
        self.IP[prefix].append(flow)
    else: #create a header and enter the
        flow
        header = TCPLatency()
        flow = TCPflow(suffix, srcPort,
            arrivalTime, PcktTypes.SYN)
        self.IP[prefix] = [header, flow]
elif tokens[6].find('[ACK]') >= 0:
    #search for handshake
    if prefix in self.IP:
        for flow in self.IP[prefix][1:]:
            if flow.srcIPsuffix == suffix and
                flow.srcPort == srcPort:
                self.IP[prefix][0].
                    update_Latency(arrivalTime,
                        flow.arrivalTime)
                #update the flow (overwrite)
                flow.flags |= PcktTypes.ACK
                flow.arrivalTime =
                    arrivalTime
elif tokens[6].find('[SYN,ACK]') >= 0:
    #find the handshake (entry is in reverse
        direction)
    prefix = '.'.join(str for str in destIP.
        split('.')[0:3])
    suffix = destIP.split('.')[3]
    if prefix in self.IP:
        for flow in self.IP[prefix][1:]:
            if flow.srcIPsuffix == suffix and
                flow.srcPort == destPort:
                flow.flags = flow.flags |
                    PcktTypes.SYN_ACK
elif tokens[6].find('RST') >= 0: #
    #####

```



```

        if srcIP in self.servers.serverIPs:
            prefix = '.'.join(str for str in
                               destIP.split('.')[:3])
            suffix = destIP.split('.')[3]
            if prefix in self.IP:
                for flow in self.IP[prefix][1:]:
                    if flow.srcIPsuffix == suffix
                       and flow.srcPort ==
                           destPort:
                        flow.flags |= PcktTypes.
                            RST_s2c
            else:
                if prefix in self.IP:
                    for flow in self.IP[prefix][1:]:
                        if flow.srcIPsuffix == suffix
                           and flow.srcPort ==
                               srcPort:
                            flow.flags |= PcktTypes.
                                RST_c2s

    return True

def remove(self, currentTime):
    self.p0 = self.p1 = self.p2 = self.p3 = self.
        p4 = 0
    self.pcktCount = 0
    sockets = []
    for prefix, row in self.IP.items():
        for flow in row[1:]:
            socket = prefix+'.'+flow.srcIPsuffix+
                ':'+flow.srcPort
            self.pcktCount += 1
            if flow.flags == PcktTypes.RST_c2s: #
                usual (only RST_c2s)
                row.remove(flow)
                #
                sockets.append(socket)
                self.p0 += 1
            elif flow.flags & PcktTypes.ACK != 0:
                #usual SYN & ACK => SYN_ACK
                row.remove(flow)
                #
                sockets.append(socket)
                self.p0 += 1
            elif flow.flags == PcktTypes.RST_s2c:
                #unusual
                self.p2 += 1

```

```

        row.remove(flow)
        sockets.append(socket)
    elif flow.flags & PcktTypes.RST_c2s
        != 0: #unusual (since 1st condition
        failed we must have SYN_ACK)
        self.p4 += 1
        row.remove(flow)
        sockets.append(socket)
    elif flow.flags == PcktTypes.SYN_ACK:
        #potential unusual due to timeout
        if row[0].get_Latency() < 0:
            time = flow.arrivalTime +
                self.serverTimeOut
        else:
            time = flow.arrivalTime + row
                [0].get_Latency()
        if time < currentTime:
            self.p3 += 1
            row.remove(flow)
            sockets.append(socket)
    elif flow.flags == PcktTypes.SYN: #
    potential unusual due to timeout
        #self.p1 += 1
        if row[0].get_Latency() < 0:
            time = flow.arrivalTime +
                self.serverTimeOut
        else:
            time = flow.arrivalTime + row
                [0].get_Latency()
        if time < currentTime:
            self.p1 += 1
            row.remove(flow)
            sockets.append(socket)

    return sockets
print(self.p0, self.p1, self.p2, self.p3,
      self.p4, end='\n')

def __repr__(self):
    return self.IP.__str__()

```

### PcktTypes.py

---

```
# from enum import Flag
class PcktTypes():
    SYN = 0
    SYN_ACK = 1
    ACK = 2
    RST_s2c = 4
    RST_c2s = 8
```

### Servers.py

---

```
class Servers:
    def __init__(self):
        self.serverIPs = ['54.175.190.37', '
                           54.175.190.38']
#         self.serverIPs = []
#         for i in range(0,10):
#             self.serverIPs.append('192.168.0.'+str(
i))
# #         print(self.serverIPs)
```

### srcIP.py

---

```
class srcIP:
    def __init__(self, IP, arrivalTime, flag):
        self.IP = IP
        self.arrivalTime = arrivalTime
        self.flag = flag

    def get_IP(self):
        return self.IP

    def get_arrivalTime(self):
        return self.arrivalTime

    def get_flag(self):
        return self.flag

    def update_flag(self, flag):
        self.flag = flag

    def remove_IP(self):
        del(self)
```

### TCPflow.py

---

```
from PcktTypes import PcktTypes
class TCPflow:
    def __init__(self, srcIPsuffix, srcPort,
        arrivalTime, flags):
        self.srcIPsuffix = srcIPsuffix
        self.srcPort = srcPort
        self.arrivalTime = arrivalTime
        self.flags = flags

    def __repr__(self):
        s = self.srcIPsuffix+', '+self.srcPort+', '+str
            (self.arrivalTime)+',SYN';
        if self.flags & PcktTypes.SYN_ACK:
            s += ',SYN_ACK'
        if self.flags & PcktTypes.ACK:
            s += ',ACK'
        if self.flags & PcktTypes.RST_c2s:
            s += ',RST_c2s'
        if self.flags & PcktTypes.RST_s2c:
            s += ',RST_s2c'
        return s
```

### TCPlatency.py

---

```
class TCPlatency:
    def __init__(self, latency=-1, link=None):
        self.latency = float(latency)
        self.link = link
        self.__alpha__ = 0.5

    def get_Latency(self):
        return self.latency

    def update_Latency(self, completion, arrival):
        l = float(completion) - float(arrival)
        if self.latency < 0:
            self.latency = l
        else:
            self.latency = (self.__alpha__ * l) + ((1
                - self.__alpha__) * self.latency)
            # EWMA:  $L_{n+1} = \alpha l_n + (1-\alpha) L_n$ 
#            print(self.latency)

    def get_Link(self):
        return self.link

    def __repr__(self):
        return str(self.latency)
```

## Bibliography

- [1] *SDN Protocols*, Std. [Online]. Available: <http://searchsdn.techtarget.com/news/2240227714/Five-SDN-protocols-other-than-OpenFlow>
- [2] *OpenFlow Switch Specification*, Std. Version 1.3.1 (Wire Protocol 0x04), September 6, 2012.
- [3] *OpenFlow Switch Specification*, Std. Version 1.3.2 (Wire Protocol 0x04), April 25, 2013. [Online]. Available: <https://www.cs.princeton.edu/courses/archive/fall13/cos597E/papers/openflow-specv1.3.2.pdf>
- [4] S. Sezer and D. L. and. (12 July 2013) Implementation challenges for software-defined networks. [Online]. Available: <http://ieeexplore.ieee.org/document/6553676/>
- [5] A. Tootoonchian and Y. Ganjali. (2010) Hyperflow: A distributed control plane for open-flow. [Online]. Available: <https://pdfs.semanticscholar.org/f7bd/dc08b9d9e2993b363972b89e08e67dd8518b.pdf>
- [6] T. Koponen and M. Casado. Onix: A distributed control platform for large-scale production networks. [Online]. Available: [https://www.usenix.org/legacy/event/osdi10/tech/full\\_papers/Koponen.pdf](https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Koponen.pdf)
- [7] (June, 2014) Open networking foundation sdn architecture. [Online]. Available: [https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR\\_SDN\\_ARCH\\_1.0\\_06062014.pdf](https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf)
- [8] M. G. Mehiar Dabbagh, Bechir Hamdaoui and A. Rayes, “Software-defined networking security: Pros and cons,” Tech. Rep. [Online]. Available: <https://pdfs.semanticscholar.org/2bb7/d94cde907934ed12613c169966067da23c7e.pdf>
- [9] M. Y. Ijaz Ahmad, Suneth Namaly and A. Gurtov. Security in software defined networks: A survey. [Online]. Available: <http://ieeexplore.ieee.org/document/7226783/>



- [10] S. Shiny, V. Yegneswaran, P. Porras, and G. Guy. (2013) Avant-guard: Scalable and vigilant switch flow management in software-defined networks. [Online]. Available: <http://dx.doi.org/10.1145/2508859.2516684>
- [11] L. Wei and C. Fung. Flowranger: A request prioritizing algorithm for controller dos attacks in software defined networks.
- [12] P. Fonseca, E. Mota, and A. Passito. (April 2012) A replication component for resilient openflowbased networking.
- [13] R. Braga, E. Mota, and A. Passito. (2010) Lightweight ddos flooding attack detection using nox/openflow. [Online]. Available: <https://pdfs.semanticscholar.org/deee/c803eaceec8cd5254c64e73b67959c5e7670.pdf>
- [14] [Online]. Available: [https:// www.blackmoreops.com/2015/04/21/denial-of-service-attack-dos-using-hping3-with-spoofed-ip-in-kali-linux/](https://www.blackmoreops.com/2015/04/21/denial-of-service-attack-dos-using-hping3-with-spoofed-ip-in-kali-linux/)