Bachelor Thesis

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Performance and Security Influence of Augmenting DDoS Protection Systems using SDN and NFV

Lukas Beierlieb
Department of Computer Science
Chair for Computer Science II (Software Engineering)

Prof. Dr.-Ing. Samuel Kounev
First Reviewer

M.Sc. Lukas Iffländer
First Advisor

Submission
16. January 2018

www.uni-wuerzburg.de

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Wuerzburg, 16.01.2018

.....

(**Lukas Beierlieb**)

Deutsche Zusammenfassung

Cyberangriffe stellen eine große Gefahr für die moderne, technologisierte Gesellschaft dar. Distributed Denial of Service-Angriffe sind in der Lage die Erreichbarkeit beliebiger Dienste im Internet zu verhindern ohne auf Sicherheitslücken im Zielsystem angewiesen zu sein. Heutzutage sind, unter anderem wegen dem Internet of Things, mehr Geräte an das Internet angeschlossen als je zuvor, was die Gefahr großer DDoS-Attacken noch erhöht.

Teilweise belasten existierende Verteidigungsansätze den Server, der den zu schützenden Dienst anbietet, direkt, teilweise ist der Schutzmechanismus auf einem anderen System untergebracht. In diesem Fall muss aber der ganze Netzwerkverkehr des Anwendungs-Servers von diesem verarbeitet und weitergeleitet werden.

Es gibt diverse Möglichkeiten diese Abschwächungstaktiken zu verbessern, im einfachsten Fall mit schnellerer Hardware. Ein anderer Ansatz ist die Verlagerung in den Bereich der Internetanbieter, sodass Attacken erkannt und verhindert werden können, bevor sie ihr Ziel überhaupt erreichen. Diese Arbeit präsentiert einen neues Konzept zur Bekämpfung von SYN-Flut-Angriffen, das durch Verwendung der modernen Netzwerktechnologien SDN und NFV einerseits den Anwendungsserver nicht belastet und andererseits das Verteidungssystem nicht mit Paketen akzeptierter TCP-Verbindung unnötig stört, indem diese direkt zum Dienst durchgeleitet werden. Darauf aufbauend wurde zusätzlich eine Netzwerküberwachung entwickelt, die Nutzer mit ungewöhnlich hoher Bandbreitennutzung einschränkt.

Der Inhalt dieser Arbeit besteht aus einer Einführung in die technischen und konzeptionellen Grundlagen, einem Überblick über existierende Lösungen, der Vorstellung des entwickelten Ansatzes, sowie dessen Implementierung, Validierung und Evaluation.

Contents

1. Introduction

Cyber attacks are a major threat to our society. Lloyd's CEO Inga Beale values the cost of cyber attacks at $400 billion every year [llo15]. One group of such cyber attacks are the so called Distributed Denial of Service attacks, consisting of up to thousands and millions of attackers. These attacks have been a problem in IT security for a long time and still are an ubiquitous threat today.

The main problem with DDoS attacks lies in the fact that there is no definite defense against them, as they do not necessarily depend on security vulnerabilities, but can rely on brute force - which is in this case network bandwidth. An arms race has developed between attackers, trying to increase their bot nets in size to generate higher bandwidth attacks, and service providers, improving their mitigation countermeasures to withstand those attacks.

Existing work presents approaches that have realized ways to counter attacks locally on the defended server [Edd07], as well as using a dedicated protection server [syn]. However, this either creates additional load for the used server or all traffic has to be routed via the additional server, creating a new target for attack and thus an additional potential bottle neck.

There a multiple ways to achieve better protection results, e.g. by running existing methods, but with faster hardware or by moving the defense systems from local networks to the internet service provider space, mitigating attacks even before they reach their target. This bachelor thesis introduces a defense mechanism operating locally in the victim's network and utilizing the modern network architecture paradigms Software Defined Networking and Network Function Virtualization. Since this thesis is embedded in the context of security of cloud services for Internet of Things (IoT) devices, where communication via REST APIs is common, the developed defense mechanism is oriented towards HTTP and TCP based attacks. We propose a new approach against SYN flood attacks, that puts no additional load on the application server, but also takes burden of the defense system by rerouting packets of established connections directly to the server. To the best of our knowledge, this is the first work to introduce this concept, from us named TCP Handshake Remote Establishment And Dynamic rerouting using SDN (THREADS). Along with that, a network monitoring mechanism that throttles clients with abusive bandwidth usage has been developed.

For the evaluation of our proposed mechanism we lay out a testbed consisting of attackers, valid clients, a server, as well as the introduced security VNF and a SDN controller in a

realistic network structure. The server and the security VNF are deployed inside an SDN enabled network. We perform behavior validation and performance tests under various conditions.

The remainder of this thesis is structured as follows: After the introduction in this chapter we introduce the technical background for our approach in Chapter 2. In Chapter 3 a brief overview of the related work is given. Then, our approach and its enhancements over existing solutions are explained in Chapter 4. The actual implementation is described in Chapter 5. Validation and performance evaluation are described in Chapter 6 and Chapter 7 concludes this thesis.

2. Background

2.1 Distributed Denial of Service Attacks

DoS stands for Denial of Service [HR06], an IT security threat that eliminates or reduces a service's availability to its clients, typically only as long as it is active. DDoS attacks are Distributed DoS attacks, i.e. there is not a single attacker but many. The following sections introduce common forms of DDoS attacks and discuss their respective relevance for this thesis.

2.1.1 Service Crashing

An obvious way of preventing a service from serving clients is to stop it from running. In this case, the service will not be accessible even after the end of the active attack until the service is restarted. While this type of attack is very effective, it is not always applicable, because there has to be a vulnerability that allows crashing the service partially or completely and that has to be exploitable for the attacker, who is usually limited to reaching the service with network packets.

The Ping of Death attack is a classical implementation of this type. RFC 791 defines the maximum size of an IPv4 packet as 65535 bytes [P+81a]. A malformed ping packet exceeding this size sent from the attacker usually will not be sent as a whole (e.g. because of Ethernet's 1500 maximum transmission unit) but as multiple fragments. If the kernel of the system receiving the packets assumes everybody else respects the standard and never checks actual packet sizes, a buffer overflow can happen by assembling a oversized, fragmented packet, effectively crashing the whole operating system.

This type of attack is not considered further, as protection either happens by removing the bugs or by using an Intrusion Prevention System (IPS), which scans for harmful packets and prevents them from reaching their actual target.

2.1.2 Network Occupation

Network occupying attacks on the other hand can be successful on every target, independent of the software they are running. As the name implies, the attacker or - most of the time - the attackers send so many packets to the victim that his link to the Internet is saturated. Consequently, there is no link capacity left to transmit packets from legitimate clients to the target and the Internet Service Provider (ISP) has to drop the packets before they even reach the target network. Which types of packets are used for this does not

matter, there just have to be enough. Also, with this attack not only single services are in danger. If spam packets saturate important links inside the Internet infrastructure, whole parts of it can be rendered unreachable. "Reddit, Twitter, Etsy, Github, SoundCloud, Spotify and many others were all reported as being hard to reach by users throughout the attack, which lasted about two hours." [dyn]

The defense system developed in this work is positioned in the local network, thus it cannot offer protection against link saturation.

2.1.3 Other Resource Occupation

Besides network link capacity, there are other resources necessary to serve client requests, e.g. processing time, memory or storage capacity. This type classifies attacks which do not exhaust network links - therefore packets sent from legitimate clients can actually reach their destination - but allocate most part of another resource, preventing the client requests from being processed and answered.

That such an attack does not have to involve a flood of packets is demonstrated by the Slow Loris attack. In this case, the amount of open connections a web server can handle at the same time is the resource that is consumed. The attack consists of opening up many connections and keeping them open for as long as possible by only sending packets just before the connection times out. If the server doesn't configure limits for concurrent connections from a single IP or minimum bandwidth requirements, a single attacker can easily occupy a server, even with small bandwidth.

HTTP floods aim to send a server so many requests that not all of them - together with legitimate requests - cannot be processed. Therefore, attacks usually do not request easy tasks like static web pages but rather requests that are expensive for the service to answer, e.g. those with dynamic content creation or many database requests. With IoT cloud services as the context of this thesis and HTTP based REST services being common in this field, the defense system proposed is designed to mitigate HTTP floods.

The other attack type we focus on is the TCP SYN flood [Edd07]. To understand it, some information about the Transmission Control Protocol (TCP), as defined in RFC 793, is required. "TCP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols" [P+81b]. For applications using TCP as an underlying transport protocol, this means they do not have to work with individual packets, but have to explicitly open and close connections and data is perceived as a stream of bytes flowing from one connection end to the other. Those byte streams have two important properties: Everything that has been sent has to arrive on the other side and it has to arrive exactly in order. This behavior is not automatically fulfilled, because the protocol splits the information stream into segments and sends them individually. On the way, packet loss might occur or some packets might overtake earlier sent ones. To solve these problems, each packet also contains a sequence number in its packet header, which increases with every byte sent so far. Together with the sequence number, every packet has an acknowledgement number, telling the other connection end up to which sequence number it its received packets already. When a packet is not acknowledged after some timeout, it is resent, preventing packet loss. Also, packets received in the wrong order can easily be reordered using their sequence numbers. Though, it would be problematic to send the first packet like this, because it would not be determinable whether it really is the first packet, carrying an initial sequence number (ISN), or if it is not and the sequence number should actually follow up previously sent, lost packets. Therefore, before any data is sent over a TCP connection, both sides transmit their ISN before any data is sent. This is called the TCP three-way handshake, which is realized as shown in Figure 2.1. One connection end (from now on called the server) is actively listing for new connection requests. The other

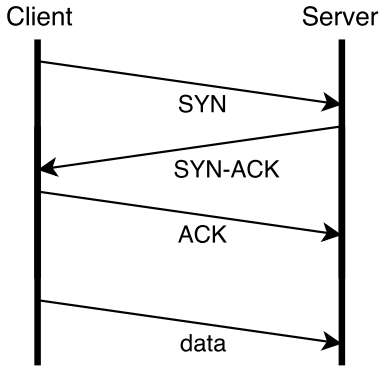

Figure 2.1: TCP handshake

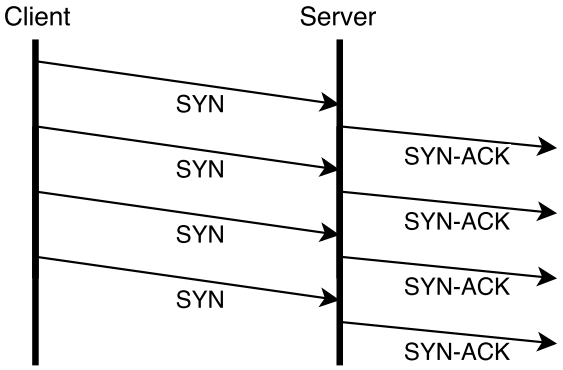

Figure 2.2: SYN flood

party (further on called the client) requests a connection by sending a SYN packet to the server. SYN is an abbreviation for "synchronize", i.e. the sequence number of the packet is the client's ISN for this connection. The server responds to this with an SYN-ACK packet. "synchronize", because it tells the client which sequence number the server is going to use and "acknowledge", because the server acknowledges having received the previous SYN packet. In the third step, the client confirms the server's choice of ISN with an ACK packet. With both ISNs known to both ends, data packets can now be safely transmitted, increasing the sequence number for every byte sent. TCP SYN floods exploit the fact that all the information about a connection has to be stored in a transmission control block (TCB) in a data structure referred to as the backlog. While it is possible to fill up the backlog with established connections, similar to what Slow Loris attack does, this requires the attacker to use his actual IP address as the source address of the packets, because he has to be able to receive the SYN-ACK packet in order to fully establish the connection with an ACK packet. With that, a simple limit for concurrent connections that one source can open can prevent an attack if the attacker count is low. Also, the attacker can be identified if he is using his actual IP address. However, not only fully established connections' TCB are stored in the backlog, already half-open connections' are. These are created when a SYN packet is received and they become established connections when the ACK packet arrives later. This allows the attacker to send SYN packets with arbitrary source addresses, thus hiding his identity and avoiding connection limits, while still occupying the servers backlog, resulting in legitimate SYN packets not being able to be handled. Figure 2.2 illustrates this for one attacker who sends SYN packets with spoofed source address, to which the server's SYN-ACK responses are transmitted. Multiple existing countermeasures are described in the Related Work chapter, sections 3.1.1, 3.1.2 and 3.2.1.

2.2 Software Defined Networking

Traditional networks consist of interconnected packet forwarding devices like routers and switches. All the control logic that is necessary for a device's operation is built into the device itself and configuration changes have to be executed via proprietary interfaces. The significant drawback of such a design is its inflexibility. With all the control logic distributed between all the network devices, configuration changes require access to every involved instance, each with possibly and different interface and access protocol. Given, for the time when network participants also were very static that was not much of a problem, but with today's virtualized cloud infrastructure, which is highly dynamic, networks have to be flexible as well.

Software Defined Networking (SDN) [JP13] is a network paradigm, achieving exactly that. The distributed controlling intelligence is removed from individual devices and centralized

in SDN controllers, which are forming the network's control plane. The traditional devices are replaced with "dumb" counterparts which have a standardized interface to allow being remote controlled from the control plane. As now their only task is forwarding packets as they are told to do, they are called the data plane. Protocols designed for controller-device communication are called Southbound APIs. While this design already allows on the fly modification and cooperation between devices, SDN goes even further. Northbound APIs define communication protocols between the control plane and the applications that are using the network.Thus, SDN controllers are not limited to using information collected from the forwarding devices to decide on network configuration, but they are also able to dynamically react to the demands the applications using the network have.

An example utilizing those functionalities is dynamic function chaining. Instead of having a fixed order in which packets traverse some network functions like firewalls or intrusion detection systems, SDN enables the possibility to have the functions report runtime statistics to a SDN controller, who then in real time evaluates the information, decides on a traverse order and configures the network to realize the planned packet flow. It is also possible to add and remove functions on demand.

Not important for this thesis and thus not further discussed are the west- and eastbound APIs, which are interfaces for communication between SDN control planes or between SDN and traditional networks.

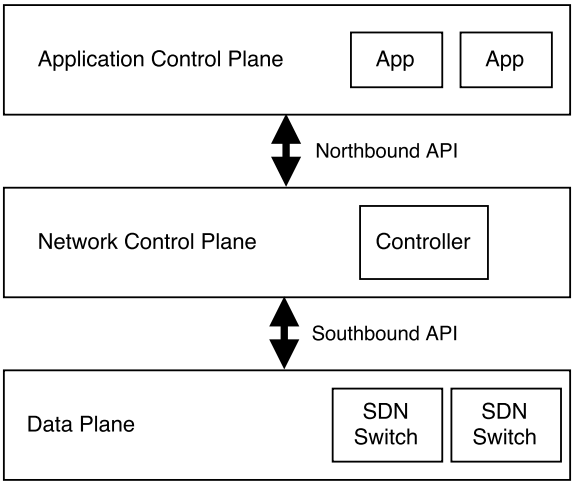

Figure 2.3: SDN Layers

2.3 OpenFlow

OpenFlow [MAB+08] is a southbound API, a protocol defining communication between OpenFlow enabled switches and OpenFlow controllers. An OpenFlow switch's behavior is defined by its flow table. Each flow in this table consists of three parts:

- **Match:** Here, multiple criteria can be used to define which packets belong to the flow. Criteria are IPv4 source and destination address, protocol type, TCP/UDP source and destination port, MAC addresses, etc.
- **Action:** This defines what happens with an incoming packet that match the flow's criteria. Possible actions are sending the packet as is out on one ore more ports, packet modification before forwarding, sending the packet to the controller, etc.
- **Stats:** The stats contain information about the amount of packets and bytes that were matched by the flow.

Alongside these, each flow has additional properties, e.g. to resolve what happens when multiple flows match a packet, each flow has a priority, so the action of the highest prioritized flow is executed. Hard timeouts define a maximum life time, after which flows are removed, while soft timeouts specify a duration in which no matches have to happen in order to remove the flow.

2.4 Network Function Virtualization

Network functions work with packets that are not addressed to them but that have to pass them in order to reach their actual destination. Their functionalities differ greatly - switches, routers, load balancers, firewalls, intrusion detection and protection systems and network monitoring devices are network functions. Because high throughput, high efficiency and low latency are often demanded, they were usually implemented as Application Specific Integrated Circuits (ASIC). While application tailored hardware devices meet previously mentioned demands, they have some drawbacks rendering them undesirable for modern, dynamic network infrastructure. Virtual Network Functions (VNF) [JP13] have the same functionality as their hardware counterparts but are implemented as software running in Virtual Machines (VM) on Commercial off-the-shelf (COTS) hardware.

This has several advantages: ASICs are expensive to develop and produce, especially because they are usually low production volume and have to be tested and validated as much as possible because mistakes cannot be fixed afterwards. Software for COTS hardware is easier and faster to develop, no production is necessary, keeping costs down. For production use they have be well tested, too, but fixes and updates are possible later. Another advantage is the flexibility in deployment. ASICs, as physical devices, have to placed in a physical location and connected to power and data cables. If the devices functionality is not needed, the hardware cannot be used for other purposes - or if the devices cannot satisfy demands, even only temporarily, more devices have to be bought. VNFs on the other hand are very flexible, they can be started by spinning up a VM on a server with available resources, can be shut down when they are not needed, allowing the resources to be used by other programs or VMs, can be migrated to another server for improved positioning, and if one instance is not enough to handle demand, more can be spun up.

While VNFs are not capable of the same performance figures as dedicated hardware, some developments improved virtualization and packet processing to a more competitive level. One of those key technology is PCI passthrough. Usually hypervisors either emulate hardware devices for their hosted VMs or rely on paravirtualization techniques. Both approaches induces a processing overhead and with possible throughputs of multiple million packets per second every slowdown is noticeable. PCI passthrough allows to unbind actual devices from the host and provide VMs direct access to them. Apart from Network Interface Cards (NIC), this technology is used to supply computational heavy VMs with GPUs [WYK+14]. However, as hardware devices are usually not built to be used by multiple machines, i.e. one device is either controlled by the host or by a single VM. This problem is solved by Single Root - I/O Virtualization (SR-IOV) [DYL+12]. Devices supporting this technology combined with the correct driver can pretend to be multiple devices. This way it can be directly used by multiple VMs, sharing its resources between them. Still, if the VNF is not a purpose built operation system like clickOS [MAR+14], the VM's kernel remains between the VNF application and the NIC. The kernel can be bypassed by giving the application's process direct access to the device. This is similar to PCI passthrough in case of a Type 2 hypervisor, where the VM is a userspace process and has direct device access. With VMs, the driver is running in the virtualized OS. Standalone applications with hardware access have to implement their own device driver.

Parts of this thesis are implemented using userspace drivers, thus more information is contained in Section 5.2. Even with fast hardware access from virtualized environments, actual processing is still happening on COTS hardware with general purpose CPUs. Field Programmable Gate Arrays (FPGA) are freely configurable hardware. They are almost as fast ASICs, but their behavior can be changed by overwriting their configuration, which allows VNFs to dynamically offload suited tasks to hardware processing [KSS14].

3. Related Work

This chapter will start with explaining two mechanisms from the Linux kernel that exist to mitigate TCP SYN flood attacks and then follow it up with scientific papers about DDoS defence strategies.

3.1 Existing Implementations in the Linux Kernel

3.1.1 SYN Cookies

Section 2.1.3 points out that the critical resource in SYN flood attacks are connection entries in the TCP backlog. SYN cookies [Edd07] are a fully TCP standard compatible way of eliminating the need for backlog entries for half-open connections. Half-open connections store source and destination addresses and ports, the clients ISN, as well as the own ISN and requested TCP options like Maximum Segment Size (MSS), Selective ACK (SACK) or window scaling [Bor12]. This is necessary to check if a received ACK packet belongs to previous SYN and SYN-ACK packets and if the client received the server's ISN correctly. The idea of SYN cookies is to store this information not locally but encode it into the SYN-ACK packet and retrieve the information from the ACK response. Figure 3.1 shows the TCP header's to content.

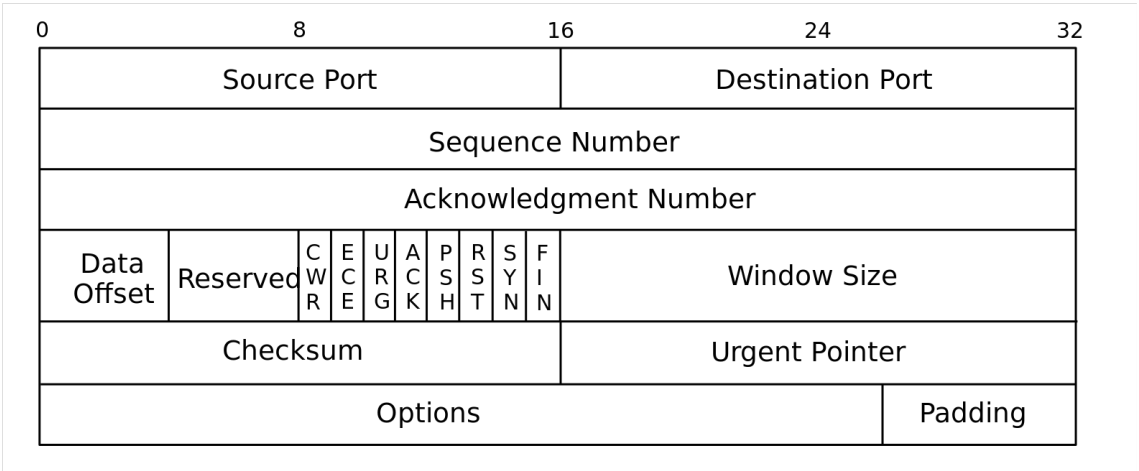

Figure 3.1: TCP header [tcp]

Source and destination port values are determined by connection. The acknowledgement number has to be the client's sequence number from the SYN packet plus 1. Data offset has to describe the headers size and flags have to be SYN and ACK. While window size can be chosen freely, its value has a big impact on following connection throuhgput, thus should be set as usual. The checksum is calculated from the rest of the header's fields. Any value can be used for the urgent pointer as it is only interpreted when the URG flag is set. One one hand, this allows to encode 16 bit of information into it, on the other this is of no use because the client ignores the data and does not transmit it back in his ACK packet. The same is true for the padding at the header's end, which additionally is supposed to consist of zeros. This leaves the sequence number and possibly options as potential information storage.

SYN cookies use the 32 bit sequence number. There are no regulations for the choice of ISN except that it should increase over time. For security concerns, it also should not be predictable. The default Linux kernel ISN generation routine is described in Section 5.1 and shown in Listing 5.1. With SYN cookies, sequence numbers are composed as follows:

Bits	Content
31 to 27	Time counter
26 to 24	Client's MSS size
23 to 0	Hash value of connection properties

The time counter is used to fulfill the increasing ISN requirement and calculated by $unix\_time >> 6 \pmod{32}$, resulting in a 5 bit number increasing every 64 seconds.

MSS stands for Maximum Segment Size [Pos83] and amounts to 536 bytes by default. The MSS option allows the choice of different values. The client uses the option in the SYN packet to tell the server the maximum size of TCP segments he wishes to receive. The server replies with his MSS choice in the SYN-ACK response and usually saves the client's value in the newly created TCB. Because SYN cookies want to prevent to local memory to be used, it is efficiently stored in the ISN by choosing eight MSS values together with an 3 bit encoding beforehand, choosing the biggest size which is smaller than the client's choice and putting the size's code into the ISN.

The previously discussed 8 bits are predictable, so the leftover 24 bits have to ensure that the sequence number cannot be guessed by anybody else, while allowing the server to verify with the acknowledge number of an ACK packet whether the packet is a valid response to a SYN-ACK packet or if it is spoofed. This is realized by calculating a hash of client's and server's IP addresses and ports, as well as the time counter and another, secret number. The secret prevents predictability. When an ACK packet is received, the server calculates the hashes for the last few values of the time counter. If one of them matches, a TCB is created. All the necessary (addresses, ports, client sequence number, client MSS) can be extracted from the ACK packet.

3.1.2 SYNPROXY

SYNPROXY [syn] is a Netfilter[1] module and supposed to mitigate TCP SYN floods, too. Netfilter is "a set of hooks inside the Linux kernel that allows kernel modules to register callback functions with the network stack" (from their website). When a packet passes a hook, all registered modules are executed. SYNPROXY utilizes this functionality to prevent SYN packets from directly reaching the networking stack, where a TCB would be allocated and a SYN-ACK would be sent as response. Instead, the module drops

[1] `netfilter.org`

the SYN packet after it impersonated the kernel and sent a manually crafted SYN-ACK packet. Rather than using local memory, the connection state is stored in the packet itself, similar to previously mentioned SYN Cookies. On arrival of an ACK packet, it is checked if it belongs to a valid handshake. If this is the case, for the client it looks now like the connection is successfully established. The server on the other hand does not even know yet that a connection was requested. Therefore, the SYNPROXY module has to impersonate the client and execute a handshake with the server, opening the connection for him.

While both ends know about the connection by now, they cannot directly communicate with each other, because the ISN the module chose when impersonating the server is only in 1 of 2^{32} cases the same as the one the server will choose in the second handshake. To fix this, the module has to modify the server's sequence and the client's acknowledge numbers in every subsequent packet.

The overhead of intercepting and modifying every packet is reduced in the synsanity[2] module by matching the ISN choices of the module and the network stack. As it cannot influence the ISN the kernel will use, the only way is to predict in the module which number the network stack would use. This is done by exactly copying the kernel function and using the same secret for calculating SYN Cookie sequence numbers.

3.2 Related Papers

3.2.1 VFence

The same principle as the SYNPROXY iptables module, but implemented with VNFs outside of the target's kernel is VFence [JYR+16]. Figure 3.2 displays how a network topology with integrated VFence can look like. The agents are responsible of filtering ingress traffic based on a whitelist containing allowed address/port combinations. When a SYN packet is received, the agent use a mechanism similar to SYN cookies to generate a sequence number and sends out back SYN-ACK packet. On arrival of a valid ACK packet, the agent performs a handshake with the server and adds the connection information to its whitelist, allowing following traffic being forwarded. Whitelist entries are removed when a FIN packet is detected or a timeout expires. To enable scalability, not a single VNF has to handle all the traffic, but traffic is distributed over multiple VNFs via a dispatcher. As the whitelist of each agent is different, packets of one connection have to always be forwarded to the same. It is the dispatcher's job to balance the load as equal as possible across the agents, as well as handle addition and removal of agents when scaling. VFence's evaluation proved its effectiveness, maintaining low packet dropping rate and low response delay, while with no defense mechanism in place almost all packets are dropped and end-to-end increases drastically.

3.2.2 VGuard

Another NFV based DDoS protection approach is VGuard [FM15], however, it does not focus on mitigating SYN floods, but on attacks based on large traffic that occupies network or CPU resources. The VGuard system consists of two VNFs arranged in a service chain. Incoming traffic is filtered by a firewall VNF first and then passed to the *DDOS VNF*. Its purpose is distributing the traffic across two tunnels, both ending at the protected service. One is meant to transport traffic of trusted sources reliably, the other to separate suspicious traffic. To assign connections to the tunnels correctly, every connection has a priority value between 0 and 1. Connections known to be malicious have a priority

[2] `https://github.com/github/synsanity`

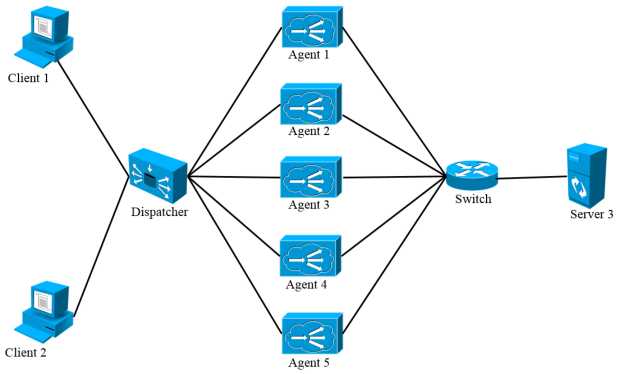

Figure 3.2: VFence example topology [JYR+16]

value of 0 and are not forwarded at all. Trusted connection have priority 1 and are sent to the high-priority tunnel. For all other, the priority value reflects how suspicious the connection is. They tunnel assignment depends on the current utilization, as well as the other connections and their priorities. Experimental testing with the system yielded the result that even under DDoS attack trusted connections' quality of service is guaranteed.

3.2.3 StateSec

StateSec [BNR+17] utilizes an existing, entropy-based DDoS detection algorithm and implements it with stateful SDN [BBCC14], resulting in a more efficient and scalable solution. As described in Section 2.2, SDN is about having simple data plane switches based on modifiable forwarding rules. The intelligence managing the rules resides completely in the controller. Practically though, this separation causes communication overhead and possibly congestion of the link between switch and controller when the controller monitors a lot of traffic information. An introduced mechanism to offload functionality from the controller into the SDN switch and thus unburden controller and the connecting network is stateful SDN. Figure 3.3 shows how incoming packets are processed inside the switch.

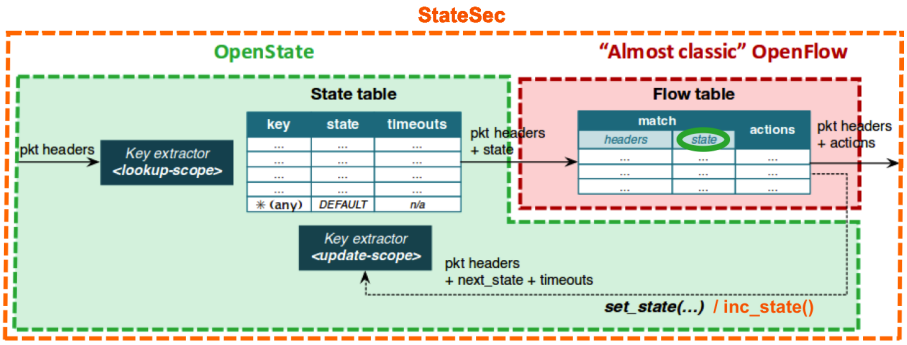

Figure 3.3: StateSec switch architecture [BNR+17]

Firstly, packet information is compared against keys from the State table. A key could be the packet's source IP address, for example. For every key that matches, the state associated with the key is added to the packet's meta information. The next step is pretty much the normal SDN behavior, with the difference that it is possible for flows to match to states saved in the meta data and that a *set_state* action exists, that can modify a key's associated state. With this, the controller has less monitoring work to do, while it remains in full control over the switch's behavior. Packets that are found to be suspicious by the monitoring state machine are matched with high priority flows which apply actions

like dropping, queuing or forwarding to other devices like Intrusion Detection Systems. Evaluation showed monitoring precision and detection accuracy increased, while control plane occupation decreased.

4. Approach

The context of this thesis is the development of a SDN/NFV security framework for protecting cloud services of IoT devices. The attack types which the DDoS protection system defends against are chosen accordingly. A common way for IoT devices to communicate with their cloud services is via Representational State Transfer (REST) APIs, which rely on HTTP requests. HTTP in turn typically uses TCP as its transport layer protocol. Therefore, the two attacks picked for this work are the TCP SYN flood as well as the HTTP flood. Detailed information about the mitigation strategies follow in the next sections.

4.1 SYN Flood Defense

With SYNPROXY (Section 3.1.2) a "Man in the Middle" technique which prevents malicious SYN packets from reaching their actual target was introduced. Variations have been described with synsanity and VFence. The key differences of the variants are:

- **Type of middleman:** The middleman has to be able to see, modify and drop packets of the connection and should remain unnoticeable for the endpoints. SYNPROXY and synsanity are modules in the server's kernel, while VFence utilizes VNFs.
- **Server ISN handling:** The middleman has to decide which ISN it sends to the client before the server knows about the connection at all. Using a number different from the server's later, actual choice requires translating sequence and acknowledge numbers of every packet. One way to match the numbers is predetermining the server's choice, which limits the middleman to something inside of the server's kernel, since from outside it ISNs should not be predictable.

Because this thesis is about enhancing DDoS protection with the SDN and NFV paradigms, the main functionality will be implemented in a VNF. This is shown in the network topology in Figure 4.1 As said before, being outside of the server's kernel means there is no possibility of predicting which ISN the server will be using. With translating numbers in every packet not offering an interesting alternative, a different method has been chosen: When impersonating the client to open the connection at the server, the middleman simply tells the server which ISN it chose to sent to the client. The server then does not calculate its own, unpredictable ISN but adopts the given one. Matching sequence numbers established, client and server could communicate without any on-the-way packet modification - if the network configuration allowed for that. A typical, static network can not by default

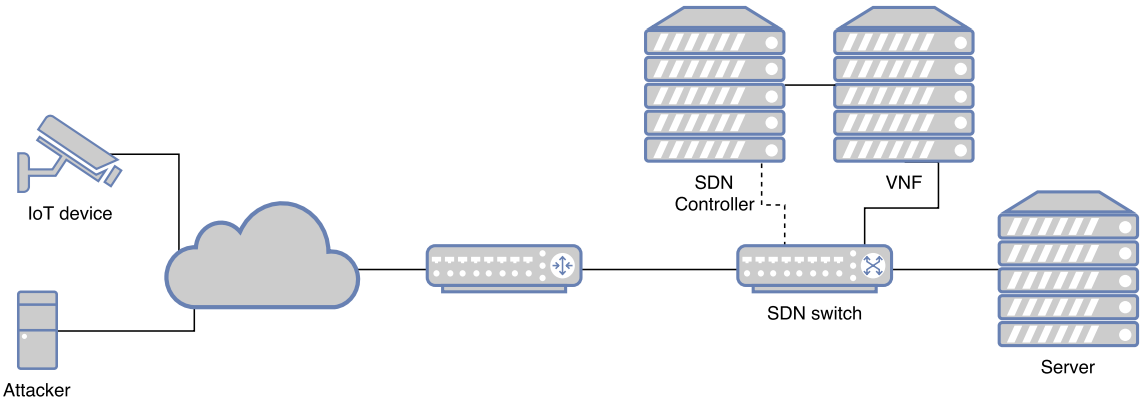

Figure 4.1: Abstract network topology

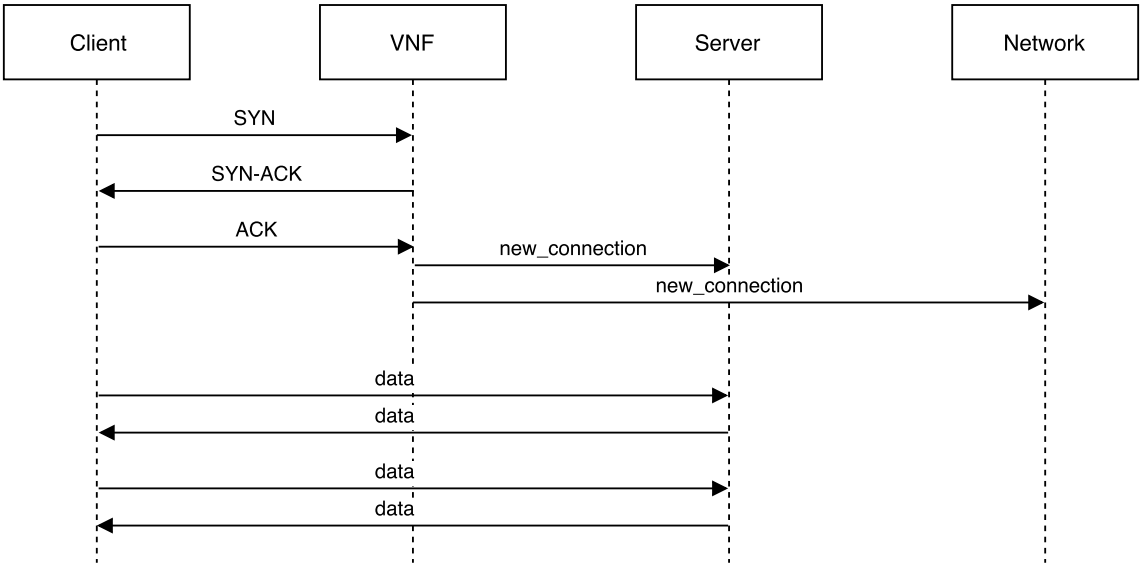

Figure 4.2: Sequence diagram of connection establishment

route packets to the VNF and suddenly change its behavior and send some packets directly to the server. Though, Software Defined Networking enables such network capabilities.

In Figure 4.2 the connection establishment process is visualized in a sequence diagram. It starts off, of course, with the client sending a SYN packet, addressed to the server. The network does not deliver to the server, though, instead the VNF receives the packet. It replies with a manually crafted SYN-ACK packet that looks like it comes from the server itself. The VNF is free to choose any sequence number for the server ISN. The packet is received by the client, who acknowledges it with an ACK packet. Again, the network routes it to the VNF. At this point, the client assumes the connection is established, but the server does not know anything about the connection and the network does not allow direct client-server communication. The order in which those two are informed about the new connection matters. If the server is uninformed but the client sends data packets, which can happen since for the client the connection looks established, and the network does already route them to the server, it will reply with TCP reset packets, probably closing the connection on client side. Thus, the client transmits all necessary connection details to the server, e.g. client IP address and port, client ISN, the server ISN chosen by the VNF. After that, the network is configured to allow direct communication for packets which are originating from or addressed to the client's address and port.

All of that informing about established connections and dynamic network route changes would not be very useful, if they only moved the bottleneck of too many half-open connections from the server to the VNF. The idea is that a network function dedicated to handling many fake SYN packets can handle an attack better than a portion of the kernel that is not necessarily hardened against specific attack types. More details about the performance advantages are explained in detail in the implementation section.

4.2 HTTP Flood Defense

The SYN flood protection mechanism makes sure only fully established connections can reach the server. This does not prevent an attacker from occupying other necessary resources by opening many legitimate connections and consuming bandwidth and processing time. To defend against at least some subtypes of this attack type, the second defence mechanism makes use of the fact that each individual connection to the server is known in the network and that it also can monitor statistics like packet or byte count. By requesting those statistics in frequent intervals, aggregating the byte count for each IP address and comparing it to the values of the last interval, it can be determined how much traffic was caused by each IP address. Not yet suspicious addresses that exceed a configured threshold are put on a blacklist. Addresses on the list that fall back below the threshold are removed from the list. The threshold has to be set dependent on what bandwidth is considered unusual and abusive for the kind of application running on the server. Connections of as evil considered addresses are forcefully traffic limited by the network in order to leave bandwidth remaining for not suspicious clients. This is realized with a Quality of Service (QoS) mechanism called queuing. The inner working of the SDN switch is displayed in Figure 4.3. Queues are used for egress traffic shaping. Instead of sending outgoing packets out directly, they are enqueued in a queue of the output port. Now, packets can be retrieved from the queue and sent while meeting configured policies like maximum bandwidth usage. To prevent clients with excessive data rates from using up all available bandwidth, for

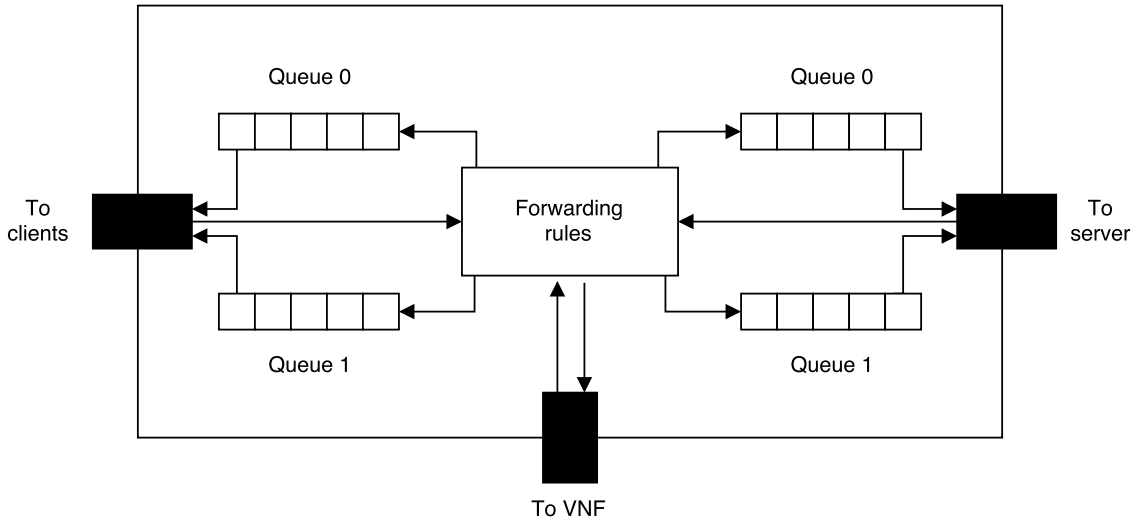

Figure 4.3: SDN switch

both involved switch ports - the one to the clients and the one to the server - two queues exist. Queue 0 does not impose any constraints, while queue 1 has a maximum bandwidth limitation that is lower than the overall transmission rate. By default the forwarding rules that allow direct client-server communication append the packets to queue 0 of the port that has to output the packet. When the VNF puts an IP address on its blacklist it also prompts the switch to modify all forwarding rules associated with that address to enqueue

them to queue 1. Similarly, an address removed from the blacklist triggers a modification of its rules to append to queue 0, again.

5. Implementation

The implementation consists of three parts. The first to be described will be a modification of the Linux kernel, that has to be running on the server. The other two are applications running on the VNF. The reason for two communicating applications realizing the VNF's behavior instead of single program is that two different tasks have to be taken care of. One job is low-level processing of SYN, SYN-ACK and ACK packets, which is very performance critical, because the more SYN packets can be handled per time interval, the more effective is the protection against SYN floods. The language chosen for this is C, making use of the DPDK library for low-level networking. The other task is mostly communication with the SDN controller over its REST API, thus via HTTP requests. In C, this is rather laborious and since it is not as performance critical, Python3 has been chosen, because it offers the simple to use *requests* library for HTTP requests. The only communication happening between the two programs running on the VNF is the DPDK application informing the Python application about new connections, for which a named pipe is sufficient.

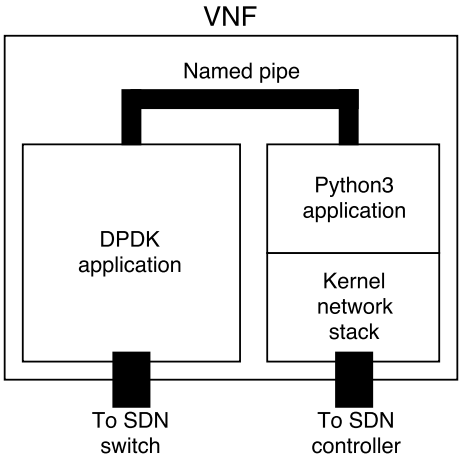

Figure 5.1: VNF overview

5.1 Kernel Modification

The goal of the kernel modification is to allow to remotely open TCP connections at the server with the possibility to tell the kernel which ISN it has to use. This allows the VNF

to inform the server about connections it already established with clients. The efficient way to implement this would be to introduce a new protocol and add packet handler functions for packets of this protocol. On arrival of such a "New TCP Connection" packet, that contains all necessary information about the connection, the handler would execute everything that also happens in the normal three way handshake, e.g. TCB and socket allocation, resulting in a TCP connection. While all the code to do this should be able to be looked up in the SYN and ACK packet handlers, it would be too much effort for this thesis to gather everything together. The approach taken modifies the existing code responsible for generating ISNs for SYN-ACK packets. Before the modification itself is explained, the default kernel behavior is introduced.

Listing 5.1: from `net/core/secure_seq.c`

```
u32 secure_tcp_seq(__be32 saddr, __be32 daddr,
                   __be16 sport, __be16 dport)
{
        u32 hash;

        net_secret_init();
        hash = siphash_3u32((__force u32)saddr, (__force u32)daddr,
                            (__force u32)sport << 16 | (__force u32)dport,
                            &net_secret);
        return seq_scale(hash);
}
```

Listing 5.1 shows the function that is used to calculate ISNs. First, it calculates a hash of the given source address, destination address, source port and destination port, together with a secret. According to RFC 793, the choice of ISN should grow over time and "cycle ... approximately every 4.55 hours"[P+81b], which is what the **seq_scale(u32)** function takes care of. This function is called from two places, both located in `net/ipv4/tcp_ipv4.c`. Once it is called in the function **tcp_v4_connect(...)** to calculate the ISN for SYN packets. The other, for this work interesting call, happens in the function **tcp_v4_init_seq(...)**, shown in Listing 5.2.

Listing 5.2: from `net/ipv4/tcp_ipv4.c`

```
static u32 tcp_v4_init_seq(const struct sk_buff *skb)
{
        return secure_tcp_seq(ip_hdr(skb)->daddr,
                              ip_hdr(skb)->saddr,
                              tcp_hdr(skb)->dest,
                              tcp_hdr(skb)->source);
}
```

It is called when the kernel is waiting for connection requests and a SYN packet is received, to calculate the ISN that is going to be sent in the SYN-ACK response. The function's parameter is a pointer to a **sk_buff**, the data structure used in the kernel to store all information about a packet. When the function is actually called, this will point to the SYN packet that the SYN-ACK using the returned ISN will be a response to. The only thing happening here is calling the **secure_tcp_seq(...)** function with the necessary parameters extracted from the SYN packet's **sk_buff** and forwarding the returned value.

This function is ideal to implement the desired behavior, as it has full access to the contents of the SYN packet and the returned value will be used directly as the ISN. The approach taken is that SYN packets containing exactly four bytes of payload will cause the ISN to be those four bytes. Normal SYN packets, which do not have any payload, are still causing the default ISN calculation to be used. Listing 5.3 displays the modified version of the **tcp_v4_init_seq(...)** function

Listing 5.3: from modified `net/ipv4/tcp_ipv4.c`

```
static u32 tcp_v4_init_seq(const struct sk_buff *skb)
{
        struct tcphdr *tcph;
        unsigned char *payload_start, *payload_end;
        tcph = tcp_hdr(skb);
        payload_start =
            (unsigned char *)((unsigned char *)tcph + (tcph->doff * 4));
        payload_end = skb_tail_pointer(skb);

        if (payload_end - payload_start == 4)
                return cpu_to_be32(*((u32 *)payload_start));

        return secure_tcp_seq(ip_hdr(skb)->daddr,
                              ip_hdr(skb)->saddr,
                              tcp_hdr(skb)->dest,
                              tcp_hdr(skb)->source);
}
```

After declaring the local variables that are going to be used, the function **tcp_hdr** translates the given **sk_buff** pointer to a pointer to start of the TCP header of the packet. The pointer to the start of the payload is calculated by offsetting the pointer to the TCP header by the size of the TCP header. The size is not fixed, but can be looked up in the header's **doff** field (data offset). This value is multiplied by 4 because it specifies the header size as the amount of 32 bit words and the used **unsigned char** pointer is only moved by 8 bit per increment. With that, the address where the payload starts is known. The address to the payload end is stored directly in the **sk_buff**, because it also is the end of the whole packet. Subtracting the addresses from each other reveals the payload size in bytes. As describe before, if the payload is four bytes long, it will be used as the sequence number. The bytes cannot necessarily be used directly, because the network byte order is Big-Endian and for example every x86 and x86_64 processor uses Little-Endian byte order. The **cpu_to_be32** macro ensures that, independent of the platform, the sequence number will be in Big-Endian order, which is used in the network. For payload lengths different from four bytes, the default routine for ISN generation is executed instead.

The amount of kernel code changed and added is very low, but this comes with a drawback. The connection creation still relies on the TCP handshake implementation of the kernel, thus a full handshake is necessary between VNF and server. The VNF cannot get away with sending a single packet to the server after it completed a handshake with a client, it can only send a SYN packet with the server's ISN in the payload and has to react the SYN-ACK response packet with another ACK packet to finally open the connection at the server. The duration of the time slice in which the client thinks the connection is open and possibly already sends data and the server cannot accept these packets is increased, though, latency between VNF and server usually should be very low.

Running with this modification is only safe when all SYN packets come from trusted sources, as it allows for predictable sequence numbers. With the defense system in place, the only arriving SYN packets are those generated by the DDoS protection VNF or potentially from clients who established a connection through the VNF beforehand and have their flow not removed yet.

5.2 DPDK Application

As mentioned in Section 2.4, network functions operate on packets that are not addressed to them and they are not only interested in the packet data but also the protocol headers. This means, even though the VNF is only needs to processes TCP packets, it cannot use

sockets of the type *SOCK_STREAM*, like the applications which send and receive TCP packets do. For explanation, the main events on the in-kernel way of a received TCP to the process that owns the socket the packet is addressed to is described: The network interface card (NIC) receives the packet, transfers it to main memory and notifies the CPU via an interrupt about the arrival. The interrupt handler checks the *ethertype*, recognizes it as an IPv4 packet and calls the IPv4 receive handler. There is checked whether the destination address matches with the local address. If this was not the case, depending on configuration, the packet would either be dropped or forwarded elsewhere. Since the addresses for the example packet match and the next inner protocol is TCP, the TCP receive handler is called. It implements the protocol rules like sending an ACK packet as response or reordering out of order received packets. Also, the target socket is determined via the destination port number and the packet's payload is copied to its receive buffer. When the application issues, for example, a *recv* system call on this socket, data from the socket buffer is copied into the process's memory. This shows clearly that the VNF cannot use stream sockets for its purpose. There is however a different kind of socket family, called the packet sockets.

Listing 5.4: Opening packet sockets

```
int fd1 = socket(AF_PACKET,SOCK_DGRAM, htons(ETH_P_IP))
int fd2 = socket(AF_PACKET,SOCK_RAW, htons(ETH_P_ALL))
```

Listing 5.4 demonstrates the creation of two packet sockets. The difference between them is that the file descriptor **fd1** would represent a socket that receives packets without the Ethernet header (because of **SOCK_DGRAM**) and would only receive IP packets (packet type **ETH_P_IP**). The socket that **fd2** represents receives the whole packet, viz. including the Ethernet header (**SOCK_RAW**), of every packet that is sent or received (**ETH_P_ALL**). Packet sockets are commonly used in packet sniffing applications like *tcpdump*[1] or *Wireshark*[2].

However, packet sockets and the packet processing system of the Linux kernel they rely on start to struggle with packet rates beyond a few hundred thousand packets per second. One reason is the interrupt based communication with the NIC. Every time an interrupt signals a packet arrival, one CPU core has to save its current state, execute the handler function and restore back to its previous state. To reduce this overhead under heavy load, the New API (NAPI) [Sal05] was added to the Linux kernel. With NAPI, interrupts are enabled by default. When a packet is received and the interrupt is handled, further interrupts are disabled. Meanwhile, the kernel polls the network card for new packets until either there are no more packets or a time limit exceeds. Then, interrupts are re-enabled.

There is another source of overhead when using packet sockets. The process retrieves packets from the kernel by system calls like the *recv* system call. This already requires a switch from user mode to kernel mode and the switch back on the system call return, but additionally every single byte of the packet has to be copied from the socket buffer in memory into the process memory. The same goes for sending, the packets have to be copied from user space to kernel space, before they can be sent by the NIC driver. At the time of writing the *Meltdown* bug became known. The Kernel Page Table Isolation (KPTI) implemented in the Linux kernel to prevent it from being exploitable, puts additional cost to executing system calls.

While there are multiple approaches to get rid of those overhead causes - Snabb [PNFR15], netmap [Riz12], PF_RING [D$^+$04] to name just a few[GEW$^+$15] - the Intel Data Plane

[1] `http://www.tcpdump.org/`
[2] `https://www.wireshark.org/`

Development Kit (DPDK) [Int15] has been chosen for this work. DPDK is a C library providing utilies for fast packet processing. In the Programmer's Guide [Int14] the principles of implemented features are explained, while the sample applications and the documentation[3] provide information how to utilize them. DPDK's capabilities are demonstrated by multiple projects, e.g. DPDKStat [TMMR16], which is able of analysing traffic at $40Gbit/s$ on commodity hardware. In the following, only the subset of DPDK that was used in this work will be explained.

The most important feature are the userspace, poll-mode drivers DPDK provides for many NICs[4]. To understand how this works, it is necessary to know how a driver running on the CPU communicates with the NIC (in case of a PCI/PCIe device). With I/O mapped peripherals, there is a device address space, separate from the memory address space. This means one address can refer to a valid memory cell and a valid device register at the same time, thus the CPU needs distinct instructions for making it clear which address space is referred to. In contrast, with memory mapped I/O, memory and device registers share the same address space, both can be accessed with the same set of read and write instructions. An abstract architecture is displayed in Figure 5.2. The CPU uses logical addresses in its instructions, these are translated into physical addresses in the Memory Management Unit (MMU). Depending on whether the address is mapped as I/O or as memory, the associated hardware handles the respective read or write request. In this case, the CPU

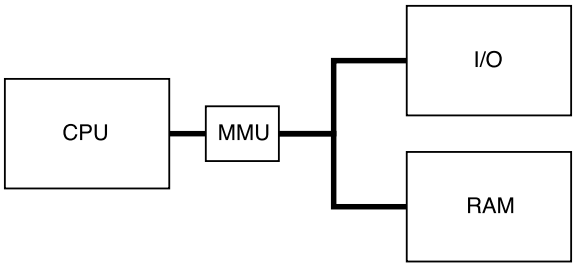

Figure 5.2: Memory and I/O access

always acts as the bus master, instructing somebody to else to do something. If this was the only mode of operation, transporting larger chunks of data - e.g. a network packet - from a device into main memory would require a CPU core to actively copy the data. This is not the case though, a peripheral device can also become the bus master and transfer data directly from or to main memory, which is called Direct Memory Access (DMA). Not only are DMAs faster than CPU driven copying, the CPU can also execute other instructions concurrently. A NIC driver's job now generally is allocating memory; part of it as an area where the NIC can write received packets to via DMA. Another part is used to setup the RX queue, a ring buffer containing pointers into the previously mentioned area, telling the NIC where it should actually copy each packet to. If the queue is empty, the network card has to drop received packets. There also exists a TX queue, containing addresses at which packets are stored in memory, that are supposed to be sent out. When the NIC retrieves an address from the queue, it copies the referenced packet into its device memory and physically sends it. The driver also has to inform the NIC where the queues are located in memory, as well as register an interrupt handler that reacts to the interrupts generated by the device - when a packet was successfully copied to RAM and now has to be processed by the network stack, as well as when a packet has been successfully sent, so the memory containing the packet can be freed and used for other purposes. So, in order to have a driver run in userspace, it has to be able to write to the addresses the device

[3] https://dpdk.org/doc
[4] http://dpdk.org/doc/nics

is mapped to. The MMU, as well as logical and physical addresses have been mentioned previously, this is now followed up with an explanation about virtual memory.

With virtual memory, each process can use the full address space. This means each address can be used by multiple processes at the same time and an address is only unambiguous when it is clear which process it belongs to. To project the virtual address spaces into the actual, physical address space, the virtual address spaces are divided into pages, typically with a size of 4 KB. Each process has a page table, where for every page is saved whether it is present in physical memory and if, at which physical address. Handling memory access like this has many advantages. Processes are separated from each other, which, for one, provides increased security, since one process cannot modify another process's memory, but also allows each process to freely choose the addresses it uses without checking if the memory area is already in use by somebody else. However, this does not restrict shared memory between cooperating processes, as pages from different pages can be mapped to same physical addresses. Also, a page does not necessarily have to be present in memory when it is not needed. Therefore, when more pages are present than can be fitted in the RAM, they can be swapped out to storage devices and swapped back in when a process tries to access them. This even enables programs to use more memory than the machine has RAM. The disadvantage is performance. For every memory access, the physical location has to be determined and if it the page is swapped out, the whole page has to be fetched from a slower storage device before it can be accessed. To prevent having to querying a page table for every memory access and making use of the principle of locality, the MMU has a cache for page translations, called the Translation Lookaside Buffer (TLB).

Now, to give a process access to a NIC's memory mapped I/O registers, one or multiple pages have to be mapped to the registers' physical addresses. Also, there should not be a conventional kernel driver in place, also trying to access the device. For this reason, NICs that are supposed to be used by DPDK applications have to be bound to either the *uio_pci_generic*, *uio_igb* or *vfio* driver. UIO stands for Userspace I/O system, developed explicitly for userspace drivers. The *uio_igb* driver is a DPDK-compatible kernel driver for Intel NICs, while VFIO, short for Virtual Function IO, is used when a IOMMU is used. Without IOMMU, DMAs are issued by the devices with physical addresses, allowing them access to the full RAM. IOMMUs realize virtual memory for peripheral devices, providing security and isolation on performance cost. All these drivers provide functionality for mapping the devices registers to userspace and disable or remap interrupts. In case of DPDK userspace drivers, interrupts are not needed because - as said before - they are poll-mode drivers. This means, device communication happens only on request from the application. The driver change can be done manually or with DPDK's dpdk-devbind utility. Listing 5.5 shows the process of loading the *uio_pci_generic* driver module, unbinding an Intel I211 NIC from its default igb driver and binding it to UIO.

Listing 5.5: Binding NIC to *uio_pci_generic*

```
# ./usertools/dpdk-devbind.py --status

Network devices using DPDK-compatible driver
============================================
<none>

Network devices using kernel driver
===================================
0000:26:00.0 'I211 Gigabit Network Connection 1539' if=enp38s0 drv=igb

Other Network devices
=====================
<none>
```

```
# modprobe uio_pci_generic
# ./usertools/dpdk-devbind.py --force -b uio_pci_generic 26:00.0
# ./usertools/dpdk-devbind.py --status

Network devices using DPDK-compatible driver
============================================
0000:26:00.0 'I211 Gigabit Network Connection 1539' drv=uio_pci_generic

Network devices using kernel driver
===================================
<none>

Other Network devices
=====================
<none>
```

Additionally to binding NICs to a compatible kernel driver, DPDK has another requirement. To minimize page table lookups, hugepages are used. Compare to the default 4 KB pages, hugepages with 2 MB or even 1 GB page sizes require smaller page tables and the TLB will usually have a higher hit-rate. To allocate hugepages, the kernel has to support them. 1 GB pages have to be allocated at boot time, but 2 MB pages can be allocated in run time. To make hugepages accessible, they also have to be mounted into the filesystem. However, systemd automatically mounts them at `/dev/hugepages`.

Listing 5.6: Allocating hugepages

```
# echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
# mkdir /mnt/huge
# mount -t hugetlbfs nodev /mnt/huge
```

When a DPDK application is started, it has to be specified which CPU cores it is supposed to run on. Of course, when the application starts, it consists of only a single thread. The core this thread is executed on is typically called the master core. The main thread can start additional threads and specify on which of the other cores, the so called slave cores, the thread is supposed to run. Usually, the Linux scheduler can move threads from one core to another. This however comes with a performance penalty, as the thread loses its content in the core-specific L1 and L2 caches. To prevent this, the main thread is bound to the master core and each thread started on a slave core is bound to the respective core. Bound to a core means that the Linux scheduler will schedule a thread always on the that core.

For thread communication, DPDK offers an efficient ringbuffer implementation, that can handle multi-consumer and multi-producer accesses without use of a lock. A ringbuffer is a FIFO data structure. Another common FIFO data structure is the linked list, which has the advantage of allocating as much memory as needed and grow theoretically infinitely, whereas a ringbuffer has a fixed size and always allocates the memory it needs when it is full. However, because everything is already allocated, operations on a ring are typically faster than on a list, where memory has to be dynamically allocated for new elements or freed when a element is removed. Furthermore, the ringbuffer is a continuous chunk of memory, while a linked list theoretically can be scattered across the whole address space, possible affecting caching efficiency.

With the prerequisites covered, the following will explain the implementation of the TCP handshake man-in-the-middle application. First of all, to remember to which client SYN-ACK packets have been sent, a variation of SYN cookies are used to store all necessary connection information in the sequence number of the SYN-ACK packet. To clarify that no state has to be saved in the application at all, the Mealy machine in Figure 5.3 shows that the virtual state a connection is in can be determined with only the content of the packet

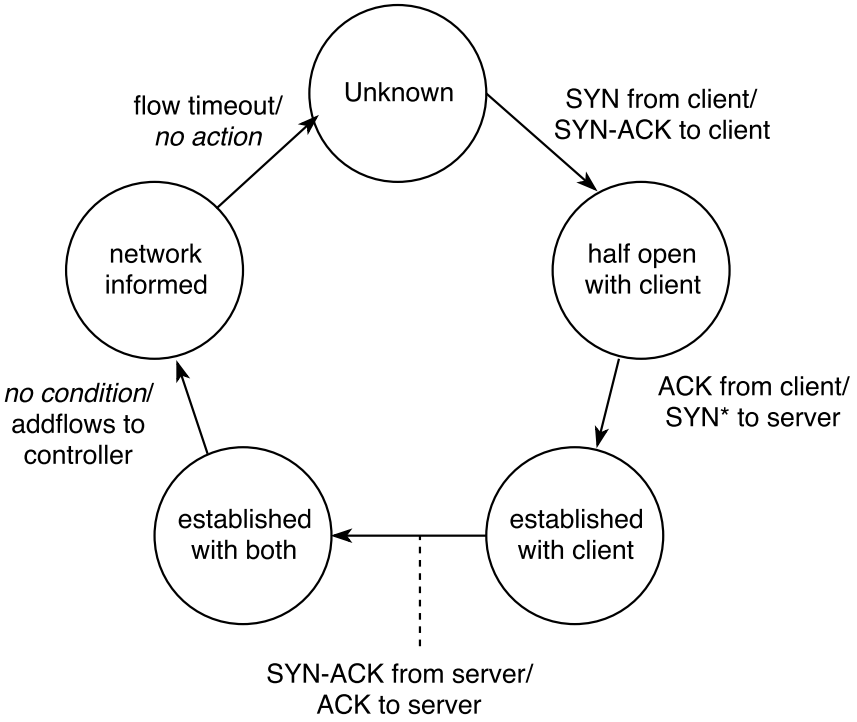

Figure 5.3: Mealy machine of the virtual state of a connection

that is processed right now. A connection is identified by the client's IP address/port combination. As long as no SYN packet with this combination is received, the connection remains in the *unknown* state. The arrival of the SYN packet triggers transition into the next state, which expresses that the connection is half opened with the client, while server and network are still unaware. The transition also executes an action, which is the transmission of a SYN-ACK packet back to the client. The next transition happens on arrival of an ACK packet with the IP/port combination and the correct SYN cookie acknowledgement. At this point the connection with the client is established. Thus, the next action has to be establishing the connection with the server. This can not happen in one step because of how the kernel modification from Section 5.1 works. The first step is to send a SYN packet to the server, with the ISN that the server has to use in the payload. All necessary information to create this SYN* packet can be extracted from the ACK packet. The server replies with a SYN-ACK packet, triggering a transition with the action of sending an ACK packet to fully establish the connection at the server. Accordingly, the new state is called *established with both*, client and server. Now, without any condition - so technically this is still is part of handling the server's SYN-ACK packet - the network gets informed about the new connection. Until now, no state had to be saved anywhere, every information necessary was extractable from the packet that was currently processed. And from now on, the DPDK application does not receive any more packets from the connection, as the network is reconfigured to route the packets directly between server and client. After some time the TCP connection will be closed, or the client just stops sending packets. Then, the server will delete the TCB it has for the connection, the flows in the network reach their timeout and the connection state switches back to unknown. This finite state machine points out that multiple cores can process packets in parallel without having to worry about any saved state that has to be accessed synchronized. One way to achieve this would be to run the same loop on every available core: Try to receive packet from NIC, if packet successfully received, process the packet. If a response packet has to be sent, instruct the NIC to send it, and continue asking for more received packets.

This approach has the advantage that there is absolutely no communication between the processing threads. However, it is not always possible to realize this. Previously, when explaining NIC driver functionality, it was mentioned that the driver and NIC communicate information about received and supposed to be sent packets via the RX queue and TX queue. It is not safe to call a DPDK driver to, for example, poll the RX queue of a NIC for new packets from multiple cores simultaneously. Some NICs do support having multiple RX and TX queues and distributing received packets between them, as well as fetching packets to sent from multiple TX queues. In the kernel, this can eliminate a single core limiting network performance, which can happen, because the interrupts for a queue have to be executed on the same core every time. Since every queue has its own interrupt, the different interrupt handlers can be distributed over multiple cores. This is call Receive Side Scaling (RSS). Not all NICs support this, though, like the *virtio* virtualized NIC, that was used for developing this application. Thus, an approach with only one responsible thread per nic was necessary.

The chosen architecture is visualized in Figure 5.4. The RX core is responsible for con-

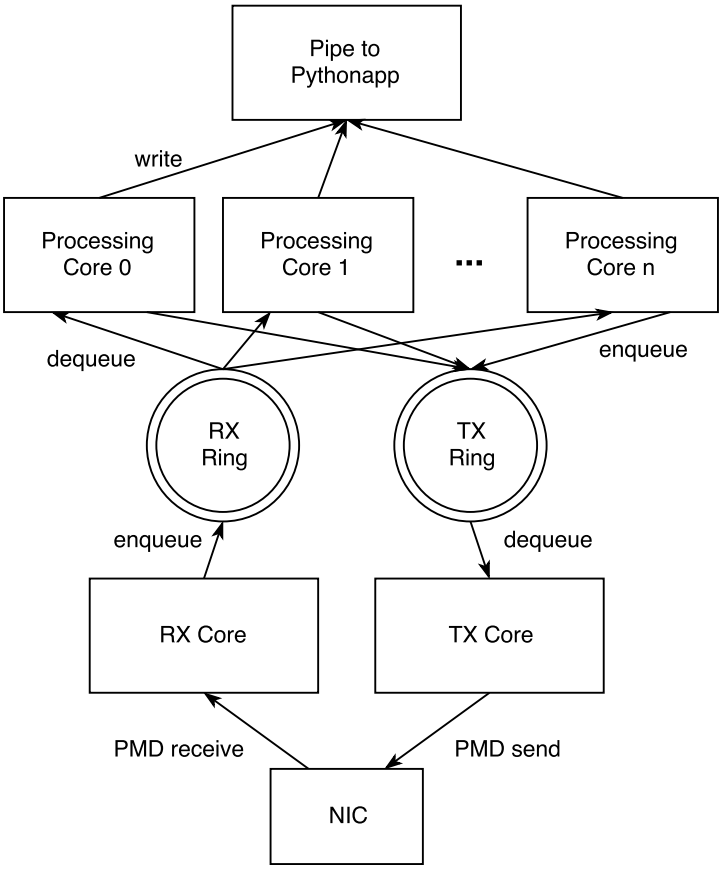

Figure 5.4: Architecture of the DPDK Application

tinuously using the DPDK driver to poll the NIC's RX queue. Every packet fetched liked this is enqueued to the RX ring. Similarly, the TX core always checks if the TX ring is empty and if it is not, dequeues packets and instructs the driver to send them. All other cores are processing cores. They try to dequeue packets from the RX ring - due to the DPDK's ring buffer implementation this is threadsafe - and processes them. Whenever they have to transmit packets, they just enqueue them to the TX ring. As described in the Approach (4.1, the DPDK application offloads the REST requests to the SDN controller to a Python3 application over a named pipe. Luckily, writing to a pipe is an atomic operation as long the pipe's buffer is not full, so the processing threads can issue writes

without synchronizing with other threads.

The first thing to do in the main function of a DPDK application is to initialize the Environment Abstraction Layer.

```
int ret = rte_eal_init(argc, argv);
```

The parameters specified are the parameters of the main function. This way the function can parse DPDK specific command line arguments, e.g. which cores should be used, how much memory should be claimed or if virtual network interfaces should be created. If the return value is -1, something in the initialization failed. Otherwise, the return value specifies how many of the arguments have been DPDK arguments. The application can then parse the remaining, however this application does not support any additional parameters. With the EAL set up, the **rte_lcore_count()** allows to check if enough cores are available. DPDK calls them more accurately *lcores*, logical cores, since one physical core appears as multiple logical cores when utilizing Simultaneous Multithreading (SMT) techniques like Intel's Hyperthreading. Another detail that was expressed inaccurate until now is that packets are stored in RX queues or ring buffers, when they actually reside somewhere in memory and are never copied around. The only thing put on any ring buffers are pointers to packets. With that said, the next thing to be done is allocating a memory pool in which the actual packet contents are stored. This is done with the **rte_pktmbuf_pool_create(...)** function. *mbuf* is an abbreviation for message buffer, the struct DPDK uses to store packets, similar how the Linux kernel uses *sk_buff*. Besides some other arguments, it takes as arguments how many *mbufs* should fit into the pool and how large one *mbuf* is allowed to be. Also one parameter specifies on which NUMA socket the pool should be created. NUMA stands for Non-Uniform Memory Access, indicating that not all memory accesses are equal. This is the case on systems with multiple processor sockets. For a core accessing memory it is faster when talking to RAM directly connected to the local CPU socket than when accessing memory connected to a remote socket, as the data additionally has to be transported through processor interconnects. This application is not developed with NUMA support in mind, therefore the constant **SOCKET_ID_ANY** is used.

Next step is setting up the NIC. With **rte_eth_dev_count()** the amount of available NICs is queried and only if it is exactly one the application continues with setting this one up, which requires multiple function calls.

```
rte_eth_dev_configure(0, 1, 1, &port_conf);

uint16_t nb_rxd = RX_QUEUE_SIZE;
uint16_t nb_txd = TX_QUEUE_SIZE;
rte_eth_dev_adjust_nb_rx_tx_desc(0, &nb_rxd, &nb_txd);
rte_eth_rx_queue_setup(0, 0, nb_rxd, rte_eth_dev_socket_id(0), NULL,
                       mbuf_pool);
rte_eth_tx_queue_setup(0, 0, nb_txd, rte_eth_dev_socket_id(0), NULL);

rte_eth_dev_start(0);

rte_eth_promiscuous_enable(0);
```

The **rte_eth_dev_configure(...)** call configures the NIC with ID 0 - which has to be the only existing NIC - to have one RX queue and one TX queue. The additional port configuration contains information like Maximum Transmission Unit (MTU). **RX_QUEUE_SIZE** and **TX_QUEUE_SIZE** are previously defined values, like 256, representing the the RX and TX queue sizes preferred by the application. These values might not be supported by the NIC, though. The **rte_eth_dev_adjust_nb_rx_tx_desc(...)** call makes sure these values are within NIC specification. The following two lines create RX and TX queue

for device ID 0 and queue ID 0. **rte_eth_dev_socket_id(0)** returns the NUMA socket the NIC is connected to, ensuring it has fast access to the queues. Finally, because the driver needs to fill the RX queue with empty *mbufs*, which the NIC can write received packets to, it needs a pointer to a *mbuf* pool. The last two lines enable the device for the poll-mode driver calls and put it in promiscuous mode, which means the NIC does not ignore packets with a destination MAC address different from the card's address, but also delivers them to the driver. This is necessary, because none of the packets that will be received is specifically targeted for the application, they are only received because of the switch's configuration.

Packets can now be sent and received, but the communication between networking cores and processing cores is still missing. For this purpose two rings are created.

```
ring_rx = rte_ring_create("ring_rx", RX_RING_SIZE, SOCKET_ID_ANY, 0);
ring_tx = rte_ring_create("ring_tx", TX_RING_SIZE, SOCKET_ID_ANY, 0);
```

This is straight forward, the function needs a unique name for the ring, the size, the NUMA socket and some flags, which can change the default behavior for single or concurrent accesses. Since this will be done manually anyway, no flags are used. The return value is used to reference the ring in enqueue and dequeue operations and thus has to be stored. Another communication device is the named pipe that is used to transmit information to the Python application, therefore has to be opened with *WRITE* permission.

```
pipe_new_conn = open(pipe_new_conn_name, O_WRONLY);
```

The only thing left now is to call a function, that generates a cryptographically secure number, which will be used for sequence number generation. The *getrandom* system call, which delivers such numbers, has been added to the Linux kernel in version 3.17 and a wrapper function was added to glibc 2.25. If those are availbable at compile time, they are used. However, the testing environment offered only older versions, in which case a fix number is used, as this does not affect performance, but the predictablity of ISNs. Finally, with everything else set up, the each lcore is told which function it has to execute.

```
unsigned lcore_id;
RTE_LCORE_FOREACH_SLAVE(lcore_id) {
        if (lcore_id != 1)
                rte_eal_remote_launch(processing_core_main, NULL, lcore_id);
}

rte_eal_remote_launch(tx_core_main, NULL, 1);
rx_core_main();
```

DPDK provides a macro to iterate over the lcore IDs of all slave cores. This is used to start the **processing_core_main(...)** on every slave lcore, except for one, on which afterwards the **tx_core_main(...)** is launched. The master core does not have any initialization left to do and executes the **rx_core_main()**, which is shown in the next listing.

```
struct rte_mbuf *bufs[NIC_BURST_SIZE];
uint16_t nb_rx, nb_rx_toring;
for (;;) {
        nb_rx = rte_eth_rx_burst(0, 0, bufs, NIC_BURST_SIZE);
        if (nb_rx > 0) {
                nb_rx_toring = rte_ring_sp_enqueue_burst(ring_rx,
                                        (void *const *)bufs, nb_rx, NULL);
                int i;
                for(i=nb_rx_toring; i<nb_rx; i++)
                        rte_pktmbuf_free(bufs[i]);
        }
}
```

In an endless loop, the function polls the RX queue 0 of NIC with ID 0 with the function **rte_eth_rx_burst(...)**. For efficiency reasons, the driver will not try to fetch only a single packet from the RX queue, but up to **NIC_BURST_SIZE**. This increases throughput by reducing per packet overhead, but also increases latency. The pointers to the packets are written into the **bufs** array and the amount of received packets is returned. If the number is greater than zero, the packets have to be transferred to processing cores via the RX ring. The **rte_ring_sp_enqueue_burst(...** function puts the received packets on the ring. The *sp* portion of the name stands for *Single Producer*. This function can only be used because no other thread also enqueues packets to the ring. The return value is the number of packets actually enqueued - this can be lower than the specified amount when the ring becomes full. To avoid loosing pointers to still allocated packets and thus leaking memory, it is essential to free the not enqueued *mbufs* in their packet pool. The function of the TX core looks similar to the RX core's one:

```
struct rte_mbuf *bufs[NIC_BURST_SIZE];
uint16_t nb_tx, nb_tx_fromring;
for (;;) {
        nb_tx_fromring = rte_ring_sc_dequeue_burst(ring_tx,
                                (void **)bufs, NIC_BURST_SIZE, NULL);
        if (nb_tx_fromring > 0) {
                nb_tx = rte_eth_tx_burst(0, 0, bufs, nb_tx_fromring);
                int i;
                for(i=nb_tx; i<nb_tx_fromring; i++)
                        rte_pktmbuf_free(bufs[i]);
        }
}
```

Here, the TX ring is constantly checked for enqueued packets. Again, the *Single Consumer* variant can be used, as no other thread will consume packets from this ring. If more than zero packets where dequeued, they are sent using the driver function **rte_eth_tx_burst(...)**. If packets are taken from the ring, but failed to sent, they have to be freed in their pool, too.

The function that is running on the processing cores looks even simpler.

```
struct rte_mbuf *bufs[RING_BURST_SIZE];
int nb_rx;
for(;;) {
        nb_rx = rte_ring_mc_dequeue_burst(ring_rx,
                                (void **)bufs, RING_BURST_SIZE, NULL);
        if (nb_rx > 0) {
                process_packets(bufs, nb_rx);
        }
}
```

In an endless loop, the RX ring is polled for packets. Because there are potentially more than one processing cores trying to consume packets simultaneously, the dequeue function that is safe for *Multi Consumer* scenarios has to be used. Whenever packets are retrieved, they are processed in the **process_packets(...)** function. This function iterates over each provided packet pointer. To access packet contents, a pointer to the begin of the packet could be used together with pointer arithmetic to access any location in the packet. However, this approach is not very intuitive and thus error prone and difficult to comprehend. A more comfortable way is shown in the next listing.

```
struct ether_hdr *ether_h = rte_pktmbuf_mtod(bufs[i], struct ether_hdr *);
struct ipv4_hdr *ip_h = (struct ipv4_hdr *)(ether_h+1);
struct tcp_hdr *tcp_h = (struct tcp_hdr *)(ip_h+1);
```

First, the **rte_pktmbuf_mtod(...)** macro is used to retrieve the address at which the packet contents begin for the current packet *mbuf*. It also directly casts it to an pointer to

a **struct ether_hdr**. This struct allows easy access to the fields of the Ethernet header and looks like:

```
struct ether_hdr {
    struct ether_addr d_addr;
    struct ether_addr s_addr;
    uint16_t ether_type;
} __attribute__((__packed__));
```

The **packed** attribute makes sure that the struct is not optimized in the compiler, but is layed out exactly as defined in the source code. This is necessary in order for the struct fields to exactly match the byte positions of the packet content. The first six bytes of the Ethernet header are the destination MAC address, here realized with **struct ether_addr** which is a byte array of size six. It is followed by the source address, also 6 bytes. Then the two byte **ether_type** defines what the next inner protocol is. The next lines make IPv4 and TCP header accessible. To find the starting address of the IP header, to the starting address of the Ethernet header the size of the Ethernet header is added. This address is the first byte that is not part of the Ethernet header, thus the first byte of the next header. This address is then casted to a **struct ipv4_hdr** pointer. Same thing for the TCP header, starting address of IP header plus size of it is the starting address of the next header. Right now, the only thing known about the packet is that it has to be an Ethernet packet. The VNF only has to handle TCP packets, thus the header structs are used to make sure the packet has to be processed further.

```
if (    rte_be_to_cpu_16(ether_h->ether_type) != ETHER_TYPE_IPv4
   ||   (ip_h->version_ihl & 0x0f) != 5
   ||   ip_h->next_proto_id != IPPROTO_TCP
        ) {
        rte_pktmbuf_free(bufs[i]);
        continue;
}
```

First the ether type has to have the value that indicates an IPv4 packet. the macro **rte_cpu_to_be_16(...)** is used to make sure the byte order of the two values match. There are two ways of ordering bytes. With little-endian order, the first byte is the least significant one, with big-endian order, the most significant comes first. The byte order used in network packets is big-endian. Some processors, including every x86 CPU, use little-endian, though. Therefore, the two byte ether type read from the packet has to be converted into the CPU's byte order to correctly compare with a value in the CPU's byte order. The advantage of using a macro for this is that it can be differently defined for different CPU architectures. On CPU using little-endian the macro flips the bytes, on systems with big-endian, nothing is done. After ensuring the packet is an IP packet, the IP header length is inspected. If the IP header includes options, the TCP header would start at the wrong address, as the actual IP header is bigger than the **struct ipv4_hdr**. In the current code, packets with IP options are not processed. The header length is stored in the **version_ihl** field, because version number and IP header length (ihl) are only four bit each, but the smallest addressable unit is one byte. As we read only a single byte from the header, byte order is irrelevant. To extract the header length the logical AND operator is applied with 0x0f, setting the four higher bits, which contain the version number, to zero. The value left is the header size, in 4 byte words. Header length without options is 5*4 byte=20 byte. Finally, the next protocol inside the IP header is encoded in the **next_proto_id** field and should indicate a TCP packet. If one of these tests failed, the packet will not be further processed. Instead its *mbuf* is freed and the next packet is inspected. If the tests passed, the packet is a TCP packet without IP options, which means the header structs casted on earlier are valid to use.

The TCP packets can be differentiated by their flags. The two for this work relevant ones were already introduced, those are SYN and ACK. Together with others, e.g. the FIN and RST flags, they are represent by a single bit each in the **tcp_flags** field. Following code shows how a SYN-ACK packet is checked for.

```
if (tcp_h->tcp_flags == (TCP_FLAG_SYN | TCP_FLAG_ACK)
        && mac_addr_equal(ether_h->s_addr, server_mac_addr)) {
        resp = process_synack_packet(bufs[i], pipe_new_conn);
        bufs[resp_i++] = resp;
}
```

Additionally to checking if SYN and ACK are set and all other FLAGS are unset, it is necessary to make sure the packet really comes from the server. The **mac_addr_equal(...)** function simply compares if the two six byte long arrays and returns whether they are equal. The **process_synack_packet(...)** function is two tasks to take care of. For one, it has to send an ACK packet back to the server and it has to write information about the new connection over the pipe to the Python application. Instead of freeing the SYN-ACK packet and allocating a new one from the memory pool, the received packet is overwritten. Still, the function returns the address of the *mbuf* that should be sent. To prepare the packet correctly, amongst other things, MAC addresses, IP addresses and ports have to be switched. Also, sequence and acknowledge numbers have to be set correctly. The new sequence number is simply the acknowledge number of the old packet. The new acknowledge number is the old sequence number incremented by one, which is not as trivial because of, again, byte order.

```
uint32_t ack = rte_cpu_to_be_32(rte_be_to_cpu_32(tcp_h->sent_seq) + 1);
tcp_h->recv_ack = ack;
```

The sequence number is taken from the old packet in big-endian order. After transforming it to CPU byte order, it can be incremented. To put it back into the packet correctly, it has to be transformed back to network byte order. Another interesting detail are the TCP and IP checksums. Luckily, DPDK offers functions to easily determine them.

```
ip_h->hdr_checksum = 0;
tcp_h->cksum = 0;

tcp_h->cksum = rte_ipv4_udptcp_cksum(ip_h, tcp_h);
ip_h->hdr_checksum = rte_ipv4_cksum(ip_h);
```

For the checksums to be calculated correctly, the checksum fields themselves have to be manually set to zero. **rte_ipv4_udptcp_cksum(...)** needs the IP and TCP header and returns the TCP checksum. The IP checksum does only depend on field of the IP header. The packet is now correctly modified and could be sent by returning the address of the *mbuf*, but before the new connection information has to be written to the named pipe.

```
write(connpipe, (const void *)&new_conn_info, sizeof(new_conn_info));
```

new_conn_info is a struct containing the IP address und port of the client.

```
struct {
        uint32_t ip_addr;
        uint16_t port;
} __attribute__((packed)) new_conn_info;
```

This is also the reason for the MAC address check. Without it, attackers could simply send SYN-ACK packets and in turn the VNF would flood the switch with rules for fake connections. The function returns the *mbufs* address, indicating it has to be transmitted.

When the packet processing loop detects a packet with only the SYN flag set, a call to the **process_syn_packet(...)** is issued. It rewrites the packet to a SYN-ACK packet,

with the special part being the generation of the sequence number. This variant of SYN cookies works by applying a one-way function to source/destination IP addresses and ports, together with a slowly increasing timestamp and a secret number. Currently, all these values are summed up and fed to DPDK's CRC hash function. It is not implied that this algorithm produces unpredictable results, but was used because of its simple implementation and the focus of this work was on getting everything around working, not finding a secure way of hashing. This simple approach also means that no TCP options are supported.

For ACK packets it is not sufficient to check for only the ACK flag set, since also every following data packet will have the ACK flag set. A possible scenario would be an incoming ACK packet, followed by multiple data packets, that reach the DPDK application because the network is not yet reconfigured. The client has not received any data from the server, thus the acknowledge number used in the data packets will be the same as the one in the ACK packet. The expected number is calculated from address/port information and of course, for every packet it is correct, so the application will send multiple SYN packets to the server. To prevent this, the payload length of every ACK packet is determined and only packets with length zero are processed. It is possible for a later packet to look exactly like the ACK packet, but only when the server sends data to the client first and the client acknowledges it without sending payload himself. However, if that is the case, those packets will never reach the VNF, since the network has to be reconfigured for that to even happen.

```
uint16_t packet_len = rte_be_to_cpu_16(ip_h->total_length);
uint8_t ip_h_len = (ip_h->version_ihl & 0x0f) * 4;
uint8_t tcp_h_len = (tcp_h->data_off & 0xf0) >> 2;
if (packet_len - (ip_h_len + tcp_h_len) == 0) {
        resp = process_ack_packet(bufs[i]);
        if (resp != NULL)
                bufs[resp_i++] = resp;
}
else
        rte_pktmbuf_free(bufs[i]);
```

The payload length is calculated with the formular $size\_of\_ip\_packet - size\_of\_ip\_header - size\_of\_tcp\_header$. If it is zero, the packet is processed. Different to the other cases, the processing function might not have to send a packet - if the acknowledge number is invalid. Then, the **process_ack_packet(...)** function returns a null pointer. When it returns another pointer it is handled like usual. Now, the first thing the processing function does is calculating the expected acknowledge number for the current and previous timestamp. If one matches, the ACK package is with a high probability not an attack packet. Then, a SYN packet is prepared to be sent to the server. The speciality here is that the sequence number the server should use has to be stored in the payload.

```
uint32_t *payload = (uint32_t *)(tcp_h+1);
*payload = rte_cpu_to_be_32(packet_isn);
```

The starting address of the packet payload is determined like the addresses of the headers. The pointer type used is 32 bit, because the sequence number, that has to be written using this pointer, is four bytes big.

When the **process_packets(...)** function processed its burst of packets, the response packets are put on the TX ring. Again, this can happen from multiple threads simultaneously, so the threadsafe enqueue variant has to be used.

```
rte_ring_mp_enqueue_burst(ring_tx, (void *const *)bufs, resp_i, NULL);
```

From there, the TX core can consume them and instruct the NIC to send them out. With that, the complete behavior of the DPDK application is explained.

5.3 Python Application

The Python3 application is responsible of two tasks. It has to read from the named pipe, which the DPDK application writes information about new connections and transform them into REST requests that are sent to the SDN controller to enable rerouting for the new connection. The other task consists of requesting flow statistics from the controller, again via REST requests, analyze them and instruct the controller to redirect suspicious IP addresses' flows through a throttled queue. A class diagram of the application can be seen in Figure 5.5. All the program logic resides in the **Connection_Manager**. The

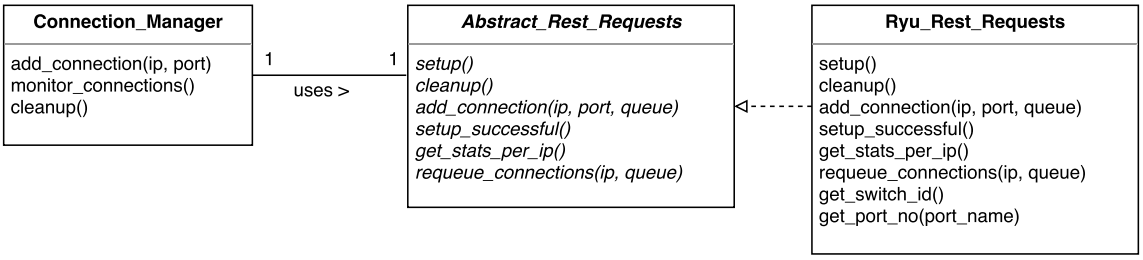

Figure 5.5: Python3 application's class diagram

Abstract_Rest_Requests is an abstract class that is the communication interface between **Connection_Manager** and the SDN controller. In development and testing the Ryu controller was used, thus the only implementation of the abstract class right now is the **Ryu_Rest_Requests** class.

When the application is started, an instance of the **Ryu_Rest_Requests** is created. The constructor needs quite a bit of information about the network, as well as some configuration parameters. The **Ryu_Rest_Requests** constructor just calls the constructor of **Abstract_Rest_Requests** with the same parameters.

```
def __init__(self, extern_port_name, vnf_port_name, server_port_name,
             server_ip_addr, rest_addr, conn_soft_timeout, conn_hard_timeout):
        self.extern_port_name = extern_port_name
        self.vnf_port_name = vnf_port_name
        self.server_port_name = server_port_name
        self.server_ip_addr = server_ip_addr
        self.rest_addr = rest_addr
        self.conn_soft_timeout = conn_soft_timeout
        self.conn_hard_timeout = conn_hard_timeout
        self.setup()
```

It has to be known which ports of the SDN controller are connected to the server, the VNF, and the router that external packets come from. Also necessary is the IP address of the server and, of course, the IP address and port with which the SDN controller can be reached. The timeout values are used in the connection rerouting flows to prevent them of existing forever. After all the arguments are saved, the **setup()** method is executed, which is the one implemented by **Ryu_Rest_Requests**. The Ryu Controller can control multiple SDN switches, so most REST requests have to specify which switch is targeted. The application expects that there is exactly one switch available and the first thing the **setup()** method does is getting the Datapath ID (DPID), by which the switch is identified, using the **get_switch_id()** method.

```
response = requests.get('http://' + self.rest_addr + '/stats/switches').text
parsed = json.loads(response)

if len(parsed) != 1:
        return -1
return str(parsed[0])
```

Issuing a REST request to `/stats/switches` returns a JSON encoded array of the existing DPIDs, which is expected to be of size 1. The DPID of the switch is returned on success. Also, the names of the ports of server, VNF and extern, are known, but in flows only port numbers can be used. The **get_port_no(port_name)** method can translate this by retrieving port information with a REST request to `/stats/portdesc/DPID`, finding the port with the given name and returning its port number. With all the necessary information acquired, the default flows, shown in the following table, can be created.

Priority	Match	Action
0	in_port: server	output: extern
0	in_port: extern	output: server
1	in_port: server, TCP	output: VNF
1	in_port: extern, TCP	output: VNF
1	in_port: VNF	output: extern
2	in_port: VNF, ip_dst: serverIP	output: server

The first two flows are used to still allow DHCP or ARP packets to be sent between the server and the gateway to extern. With completely static network configuration at the hosts, these flows are not necessary. The third and fourth flow redirect all TCP packets from server and extern to the VNF. The response traffic from the VNF is handled in the last two rules. Typically it is sent to the gateway, except when the destination IP address is the server's address. To demonstrate how flows are created via REST requests, the next listing shows the creation of the sixth flow.

```
data = {'dpid': self.switch_id,
        'priority': 2,
        'match': {
                'in_port': self.vnf_port_no,
                'eth_type': 0x0800,
                'ipv4_dst': '192.168.0.10'
        },
        'instructions': [
                {
                'type': 'APPLY_ACTIONS',
                'actions': [
                        {
                        'port': self.server_port_no,
                        'type': 'OUTPUT'
                        }
                ]}
        ]}
requests.post('http://' + self.rest_addr + '/stats/flowentry/add',
                data=json.dumps(data))
```

Now, the **Ryu_Rest_Requests** object is created and set up. Next, a object of **Connection_Manager** is instantiated. The constructor requires the **Abstract_Rest_Requests** object and two threshold values, used for the HTTP flood defense, which will be explained shortly.

After instantiating all the objects, an additional thread executing the **monitoring()** method is started. This will be looked at after the actions of the main thread are explained. At this point, the named pipe is opened with READ permission. From there, the main thread continuously tries to read from the pipe.

```
while True:
        c = os.read(pipe_new_conn, 6)

        if len(c) == 0:
```

```
                print("Pipe closed, exiting")
                conn_mgr.cleanup()
                sys.exit()

        ip_addr = str(c[0]) +'.'+ str(c[1]) +'.'+ str(c[2]) +'.'+ str(c[3])
        port = int.from_bytes(c[4:6], 'big')

        connection_manager.add_connection(ip_addr, port)
```

In the DPDK application section was explained that client IP address and port are written to the pipe, which sums up to six bytes per new connection. Therefore, it is always tries to read six bytes. If the returned byte array is of length zero, however, the pipe was closed, because the DPDK application terminated. The Python application also terminates after calling the **cleanup()** method which deletes all flows on the switch. Otherwise, address and port are parsed and the **add_connection(...)** method is called, which checks if the client IP is known to be suspicious. If it is, the new flows have to use the throttled queues (queue 1), else the flows are allowed to use the not throttled queues (queue 0). The actual code for creating the flows is found in the **add_connection(ip_addr, port, queue)** method.

Prio	Match	Action
10	in_port: server, ip_dst: clientIP, port_dst: clientPort	queue: 0|1, output: extern
10	in_port: extern, ip_src: clientIP, port_src: clientPort	queue: 0|1, output: server

These flows ensure that traffic from the server that is addressed to valid client is directly sent to the gateway and incoming traffic from that client can directly reach the server, in both directions either through queue 0 or queue 1. Additionally, the flows have soft and hard timeouts as specified when creating the **Ryu_Rest_Requests** object.

That is everything that happens on the main thread. The **monitoring()** method that is executed by the additional thread is very simple. In an endless loop it first sleeps for a fixed amount of time, then calls the **Connection_Manager**'s object's **monitor_connections()** method. That method begins with retrieving current flow statistics from the controller using the **get_stats_per_ip()**, which returns a dictionary where for every IP address that has connection rerouting flows the amount of flows as well as the total byte and packet count of these flows is stored. Using the same statistics saved from the last method executing, the traffic caused by each IP address in the monitoring interval is determined. If an IP address is in the throttling list, but the amount of traffic from the last interval is lower than the threshold specified when the constructor was called, the address is removed from the list. If an IP address that is not on the list exceeds the threshold, it is added to the list. Whenever an address is added or removed from the list, all the flows have to modified to now enqueue packets to the other queue. This is done with the **requeue_connections(ip_addr, queue)** method. The **Ryu_Rest_Requests**' implementation of this utilizes a flow modification request of the Ryu controller. Finally, the current statistics are stored, so the next execution of the method can compare the new statistics with them.

To prevent inconsistent states when **monitor_connections()** and **add_connection(...)** are executed simultaneously, at the beginning of each function a lock is acquired and at the function end the lock is released.

6. Evaluation

6.1 Testbed Architecture

To comfortably develop and evaluate the proposed defense system a virtual testbed consisting of QEMU/KVM virtual machines, Linux bridges and an Open vSwitch instance has been designed. The topology is displayed in Figure 6.1. When the approach was explained, Figure 4.1 already pointed out that the internal network has to contain the to be protected server and the VNF, which are - together with a gateway router to the Internet - all connected to a SDN switch, which in turn is controlled by a SDN controller. For the switch the Open vSwitch has been chosen, running natively on the host machine. The Open vSwitch instance is called *intn*, for internal network. To simulate accesses from the Internet, a second network, the *extn* (external network) exists, featuring one client, that is supposed to test service availability with legitimate requests. Trying to deny this availability is the task of the attackers, whose count can be configurable on testbed launch. The host machine, as participant of both networks, is responsibly for routing between them. However, *intn* and *extn* are supposed for data only, so there is yet another network for controlling and managing the VMs, the *mgmtn*. Alongside the host machine accessing VMs with this network, it is also used for the REST requests the VNF has to issue to the SDN controller. In the development phase, sometimes additional programs have to installed in VMs, so the host is able to allow them Internet access by setting up NAT using iptables. In contrast, when SYN packets with spoofed IP addresses are used to attack the server, SYN-ACK responses are not supposed to be routed out to the Internet, which is also realized with iptables rules. Bash scripts have been developed to automate VM creation and provisioning from just a Ubuntu ISO image, launching the VMs with a configured amount of attacker machines and setting up the networking between them, stopping the testbed and cleaning everything up, as well as executing attacks and measurements.

The script for creating the virtual machines requires the path to an Ubuntu installation image. If this is successfully provided, a virtual disk image is allocated and a VM is booted with disk image and installation image attached. The installation processes has to be done manually by the user, configuring hostname, username and password as required by the script, as well as make sure SSH is installed. After Ubuntu is installed on the disk image, the script takes over. After booting the machine, SSH is used to change the networking configuration to be able to handle two interface, which will be the case later. Also, the sudo configuration is altered so that it doesn't require password verification, otherwise executing remote root commands would be difficult. After the machine is shut back down,

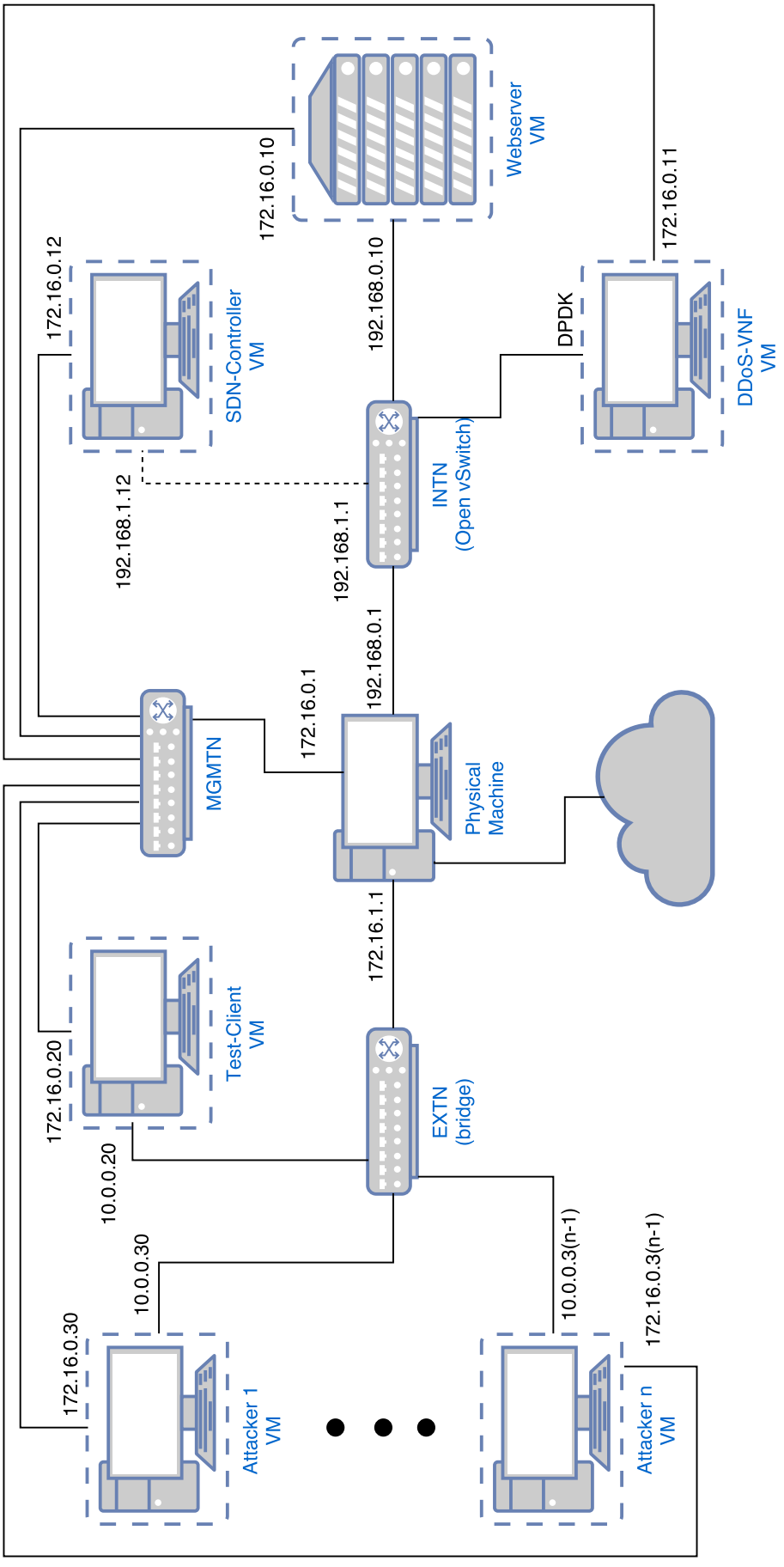

Figure 6.1: Testbed Topology

the actual testbed entities are created. To save memory, the base image is not copied once for every machine, but the VMs' hard drives are overlays with the just created image as base. In the following, it is explained which provisioning steps are necessary for the different VMs, apart from changing the hostname to something that makes each VM easily recognizable.

Server For providing a service the Apache2 web server is installed. To accept established connections from the VNF, the server also has to run a kernel with the modification from Section 5.1 built in. Therefore, the packages necessary for kernel compilation are installed, kernel sources are downloaded, the modification is applied and the kernel is compiled and installed.

VNF The VNF needs to compile and run the DPDK application. The source code is located in a git repository and so is the DPDK framework, thus git and compilation packages are installed first. After that, the DPDK repository is cloned and compiled, variables for compiling application are added to the `.bashrc` file and hugepages are allocated. The are tools for constantly binding NICs to DPDK compatible kernel drivers, however they did not work in the Ubuntu VM. Finally, the VNF application is cloned and python is installed to run the non-DPDK part of the VNF.

Controller On the Controller VM the SDN controller Ryu has to be installed:

```
$ sudo apt -y install python3-pip
$ pip3 install ryu
```

The simple installation and the well documented REST module were the reasons Ryu was chosen as the SDN controller.

Client The client needs to install tools required for evaluation, namely *hping3* and *iperf*.

Attacker To execute SYN flood attacks, the packet generation tool hping3 is installed on the attacker disk image.

Starting each VM by hand and configure networking between them is very tedious, so a launch script exists, which automates that process. For *extn*, *mgmtn* and for the connection to the SDN controller conventional Linux bridges are used.

```
ip link add name mgmtn type bridge
ip link set dev mgmtn up
ip addr add 172.16.0.1/24 dev mgmtn
```

When a bridge is created, it appears as a network interface. The host participates in the network by enabling the interface and giving it an IP address. Later, the interfaces that the VMs are connected with are added to the bridge and communication is possible. The same procedure is possible with the OVS bridge.

```
ovs-vsctl add-br intn
ip link set dev intn up
ip addr add 192.168.0.1/24 dev intn
```

However, the bridge interface itself is a special port in an OVS bridge. It does not have a port number, but rather is known as the LOCAL port. The problem with that is that it is not possible for flows to address different queues of the LOCAL port, which is essential for the HTTP flood defense mechanism. To give the host access to the bridge without using the bridge interface itself a *veth pair* is used. It consists of two network interfaces, which are connected with each other via virtual Ethernet, hence the name. One end is attached to the OVS bridge, the other end is enable and configured with an IP address by the host.

```
ovs-vsctl add-br intn
ovs-vsctl set-controller intn tcp:192.168.1.12:6633
ip link add intn_access type veth peer name intn_access_ovs
ovs-vsctl add-port intn intn_access_ovs
ip link set intn up
ip link set intn_access up
ip link set intn_access_ovs up
ip addr add 192.168.0.1/24 dev intn_access
```

The second line tells the bridge how to reach the SDN controller, the third creates a veth pair. One interface is named *intn_access*, which will be used by the host, the other is named *intn_access_ovs* and is used as a port of the OVS bridge.

The virtual machines do not have a static IP configuration, they acquired their network information with DHCP. Static configuration would work just as well for all the VMs except the attackers, because the number of attackers is variable and the same disk image is used for each one. To supply the correct IP addresses, *dnsmasq*, a DHCP and DNS server is started with a configuration that maps the MAC addresses of the devices to their respective IP addresses, as well as supply information like network mask and default gateway dependent over which bridge a request is received. With the network preparation finished, the VMs can be booted up.

```
qemu-system-x86_64 -enable-kvm -cpu host -smp $SERV_CPU -m $SERV_MEM
 -net nic,model=virtio,macaddr=AA:00:00:00:00:00
 -net tap,ifname=serv0,script=no,downscript=no
 -net nic,model=virtio,macaddr=AA:00:01:00:00:00,vlan=1
 -net tap,ifname=serv1,script=no,downscript=no,vlan=1
 ../disk_images/server.qcow2 &> /dev/null &
```

This command launches the server VM with two network interfaces. The principle is similar to veth pairs, one end is the NIC of the VM, the other end is an network interface visible for the host. *virtio* devices are paravirtualized NICs, offering lower overhead than emulating real NICs to the VM and they are also supported by DPDK. By specifying which MAC adddress should be used, the DHCP service can identify the device and deliver the matching IP, as explained. For the server example, the host will see the two interfaces *serv0* and *serv1*. Every attacker VM that has to be started needs its own disk image. These are obtained by create overlays over the provisioned attacker image. When the VMs are started with their respective overlay, the MAC addresses and interface names are incremented for every machine to make them distinguishable.

To be able to reach the DHCP service, the host ends of the VM's NICs have to be added to the bridges they belong to. This listing demonstrates added the server's second interface to the management network:

```
ip link set serv1 up
ip link set dev serv1 master mgmtn
```

Similarly, the data interfaces of VNF and server are added to the OVS bridge with the respective OVS command. Additionally, the queues for the OVS ports to host and server have to be created.

```
ovs-vsctl set port serv0 qos=@newqos -- \
  --id=@newqos create qos type=linux-htb \
      other-config:max-rate=100000000000 \
      queues:0=@queue0 \
      queues:1=@queue1 -- \
  --id=@queue0 create queue other-config:max-rate=10000000000 -- \
  --id=@queue1 create queue other-config:max-rate=200000000
```

Queue 0, for the non abusive connections, has such a high bandwidth limit that it is practically not throttled. Queue 1, however, is here limited to using 200 Mbit/s. Lastly, another script is invoked to enable Internet connectivity for the VMs and the attacker count is saved to a file, so that other scripts like the stopping script have a simple way of determining how many attackers exist.

The networking script is used to enable and disable VM Internet access by setting appropriate iptables rules. In both cases it is necessary to know which host network interface is connected to the Internet. With the ip route command the default gateway interface is determined, which is assumed to be the correct one. With no additional rules, packets from VMs are actually sent out over this default gateway, responses cannot come back, though, since the source address is a internal, local network address. To make it work, the host has to change the source address of packets routed to the default gateway, masquerading them as if they were sent be the host himself. When response packets come back, the destination address, which of course is the hosts external address, has to be changed back to the actual senders internal IP address before routing it to him. With iptables, this needs just one command:

```
iptables -t nat -A POSTROUTING -o $NIC -j MASQUERADE
```

For completely blocking Internet access, rules have to be installed that drop every packet routed to the default gateway. Setting this up is a single command as well:

```
sudo iptables -A FORWARD -o $NIC -j DROP
```

The script responsible for stopping the testbed connects to every VM via SSH and issues a shutdown command. The file created by the launch script, that contained the attacker, is used to determine how many attackers have to be powered off. Cleaning up after all machines stopped running consists of deleting the attackers' overlay disk images, removing all the bridges and queues, stopping the DHCP service and deleting the attacker count file.

The machine all the following measurement have been performed on featured a AMD Ryzen 7 1700 eight-core processor with SMT enabled and clock speeds up to 3.65 GHz and 32 GB of memory running at 2933 MHz.

6.2 Behavior Validation

To validate that the kernel modification functions as expected, SYN packets with four bytes of payload have to be sent to a machine running the patched kernel. The packet generator *hping3* is able to construct such packets. The payload can either be generated by *hping3* itself or read in from a file. To make the portion of interest more recognizable, 0x01234567 will be used.

```
echo -n -e '\x01\x23\x45\x67' > payload
hping3 --count 1 --syn --dest-port 80 --data 4 --file payload 192.168.0.10
```

The *-n* option of *echo* prevents the program of appending a newline character after the specified text. *-e* causes the interpretation of escaped characters. An escaped *x* indicates that the following are supposed to be treated as the hexadecimal representation of a byte, not as individual characters, which would be one byte each, values according to the ASCII table. The output is redirected into a file called `payload`, which will contain the four specified bytes after the command is executed. With that, *hping3* can send one TCP packet with the SYN flag set and four bytes of payload which are read in from the file to the server's port 80, where the Apache2 webserver is listening for SYN packets. The packet was generated on the first attacker (10.0.0.30), on whose NIC also packets were

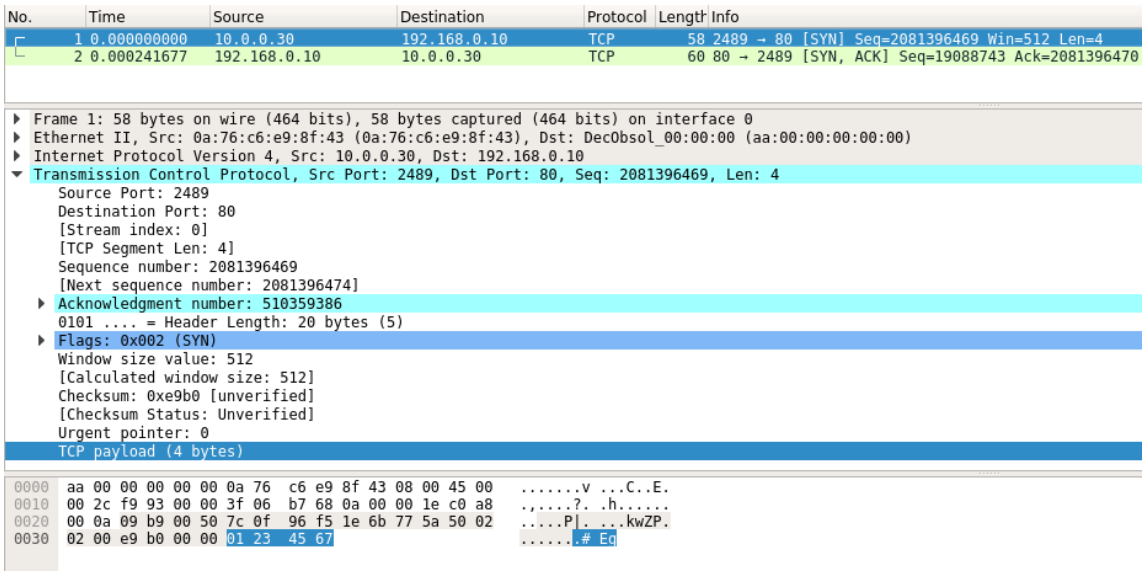

Figure 6.2: Wireshark capture: SYN packet

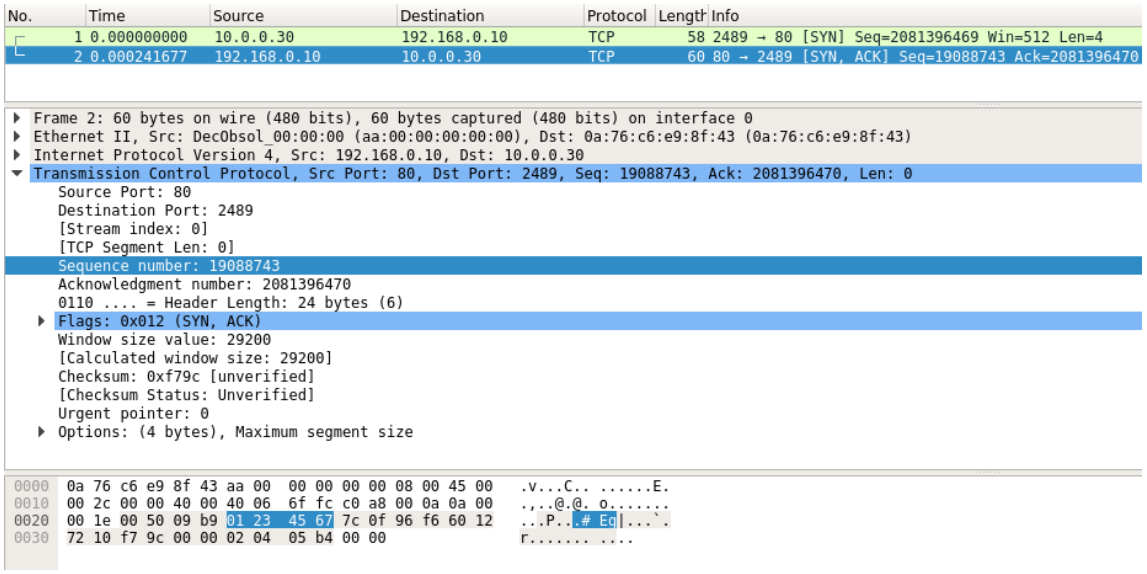

Figure 6.3: Wireshark capture: SYN-ACK packet

captured with *Wireshark*. Figure 6.2 displays the TCP details of the generated SYN packets. Important to see here is that the payload of the packet is four bytes long and indeed contains 0x01234567. Details of the SYN-ACK packet that the server replied with can be inspected in Figure 6.3. The sequence number displayed in the packet analysis is not of much use, because it is displayed as a decimal number. However, the in the hexdump of the packet the sequence number is also highlighted and it is 0x01234567, as expected.

Testing the SYN flood defense mechanism consists of establishing a TCP connection from the client with the server and checking transmitted packets, as well as the created flows in the switch. To open up a connection to the server, the client executes a *wget* command to retrieve a webpage from the web server running on the server's port 80. Packets were captured at the client's data NIC and the server's data NIC, but they were not captured by the VM's themselves, the physical host is also able to monitor the packets using his network interfaces connected to the VMs' NICs, *client0* and *serv0*. This approach has the advantage that the timestamps of captured packets use the same reference clock - the hosts clock. If capturing was done by each VM, their clocks might not be perfectly in sync. A screenshot showing the captures of both interfaces with every packet of the website access is contained in the Appendix, Figure .1. For clarity, the relevant information of important packets has been extracted into the following table.

iface	Δtime [μs]	source	dest	flags	datalen [byte]
client0	-	10.0.0.20 : 55510	192.168.0.10 : 80	SYN	0
client0	369	192.168.0.10 : 80	10.0.0.20 : 55510	SYN,ACK	0
client0	77	10.0.0.20 : 55510	192.168.0.10 : 80	ACK	0
client0	122	10.0.0.20 : 55510	192.168.0.10 : 80	ACK	193
serv0	11	10.0.0.20 : 55510	192.168.0.10 : 80	SYN	4
serv0	98	192.168.0.10 : 80	10.0.0.20 : 55510	SYN,ACK	0
serv0	1212	10.0.0.20 : 55510	192.168.0.10 : 80	ACK	0
client0	195971	10.0.0.20 : 55510	192.168.0.10 : 80	ACK	193
serv0	91	10.0.0.20 : 55510	192.168.0.10 : 80	ACK	193

The connection establishment begins with the client sending a SYN packet to the server. The packet never reaches the server's NIC, but still, 369μs later a SYN-ACK packet comes back, which thus has to be sent by the VNF. However, the packet looks like it comes from the server and the client completes the handshake with the ACK packet. For the client the connection looks established, so just 122μs later it sends the HTTP GET request, which is 193 bytes long and fits in a single TCP packet. The server has not received any packets yet and because the server is informed about new connections before the network is reconfigured, the packet will still be routed to the VNF, where it will be dropped. The VNF starts informing the server about the connection 11μs later, with a SYN packet which forces the server's kernel to use a specific ISN. Looking at the packets' hexdump, the payload of this SYN packet is the same as the sequence number of the SYN-ACK packet the client received, exactly how it should be. The SYN-ACK packet sent by the server does use the payload as sequence number, this has already been tested when the kernel modification was evaluated. Until the VNF fully establishes the connection with the ACK packet, 1212μs pass. The reason for this increased delay is probably that the VNF does not receive the packet, generate and immediately send the response, like it does with the other packets, but before sending the response, the information about the new connection is written to the pipe, to the Python application, which apparently takes it around 1ms to do. Around 196ms later, the client retransmits the GET requests and this

time it is directed to the server, where it is received after 91μs. All further data packets of the connection are handled the same.

Alongside the packet captures, the existing flows have been monitored, by saving the output of the *ovs-ofctl dump-flows* command every second. The first flow dump containing the flows for the connection is shown in the following listing. Flow cookie, byte count and idle age of flows are not included for better visibility.

```
NXST_FLOW reply (xid=0x4):
duration=0.921s, n_packets=3, idle_timeout=10, priority=10,tcp,in_port=1,
    nw_src=10.0.0.20,tp_src=55510 actions=set_queue:0,output:2
duration=0.918s, n_packets=1, idle_timeout=10, priority=10,tcp,in_port=2,
    nw_dst=10.0.0.20,tp_dst=55510 actions=set_queue:0,output:1
duration=126.038s n_packets=2, priority=2,ip,in_port=3,nw_dst=192.168.0.10
    actions=output:2
duration=126.051s n_packets=1, priority=1,tcp,in_port=1 actions=output:3
duration=126.047s n_packets=1, priority=1,tcp,in_port=2 actions=output:3
duration=126.060s n_packets=3, priority=0,in_port=1 actions=output:2
duration=126.055s n_packets=3, priority=0,in_port=2 actions=output:1
duration=126.042s n_packets=1, priority=1,in_port=3 actions=output:1
```

The third up to the eighth flow are the default flows, they are always active. The first two flows belong to the newly established connection. There are no existing flows for the client's IP address, thus it cannot be considered suspicious of using too much bandwidth yet and queue 0 is used, in both directions. The configuration used in this example does not specify a hard timeout, but an soft or idle timeout of ten seconds, which means that if no packet matched the flow for ten seconds, the flow is removed. A ten second hard timeout would destroy the flow ten seconds after its creation, regardless whether or not packets were recently matched.

The HTTP flood prevention's throttling mechanism is validated with *iperf*, a tool that measures the maximum throughput between two hosts. For the testing scenario, the VNF was configured to analyze flow statistics every second, requeuing connections to the throttled queue, when they exceeded 50 Mbit/s of traffic and requeuing connections back to queue 0, when causing less than 10 Mbit/s of traffic. *iperf* was running in server mode on the server and as client on an attacker VM. After the client part established a connection with the server part, it sends as much data as possible over that TCP connection. For a run with the duration of ten seconds all packets at the server's NIC have been captured with *Wireshark*, which is able to produce IO graphs, showing how many packets were captured in some time interval. The graph in Figure 6.4 shows the packet rate of each ten millisecond interval. The connection starts at around 0.61 seconds, maximum link capacity utilization is reached very quickly. The defense system reaction shows effect at second 1.74, from where the packet rate varies between 300 and 320 packets per 10 ms. It is hard to tell the bandwidth from the IO graph, since the packet sizes differ. The client, trying to send as much as possible, sends packets with maximum segment size of 536, which results in A Ethernet packet size of 590 bytes, while the server only acknowledges receiving the data, thus its packet size is 54 bytes. Also, there are less acknowledgement packets than data packets. *iperf* states to have transmitted 194 Mbyte in the ten seconds, which translates into an average bandwidth of 163 Mbit/s. The difference to the expected 100 Mbit/s is caused by the non throttled beginning. Therefore, measurements from 10s to 100s, with 10s steps, have been run. The results are visualized in Figure 6.5. The smaller the share of the unthrottled beginning becomes, the more the bandwidth approaches the targeted 100 MBit/s, with some fluctuation caused by different beginning phase lengths, which happen because of the monitoring checking statistics only every second.

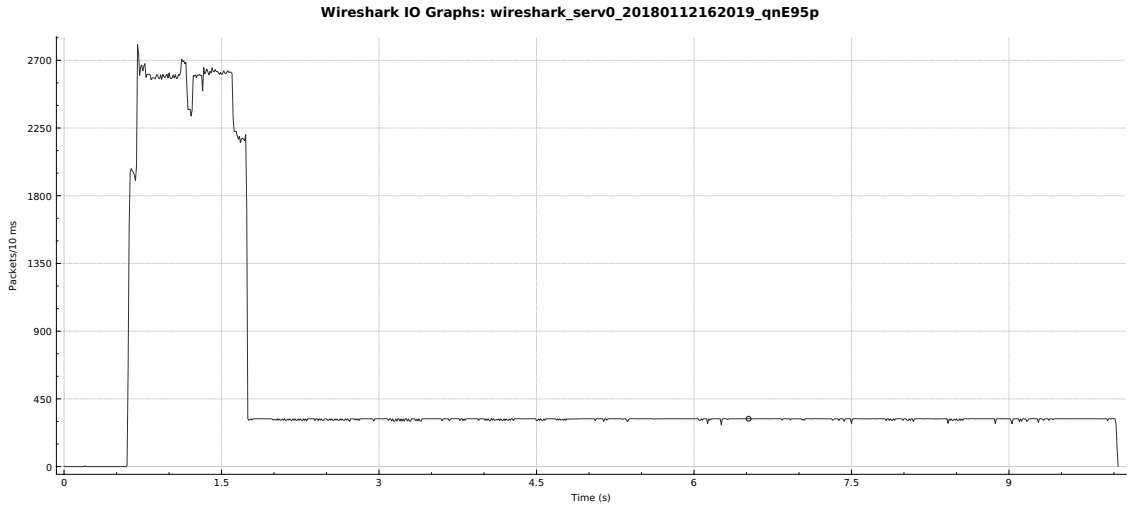

Figure 6.4: IO Graph of a throttled connection

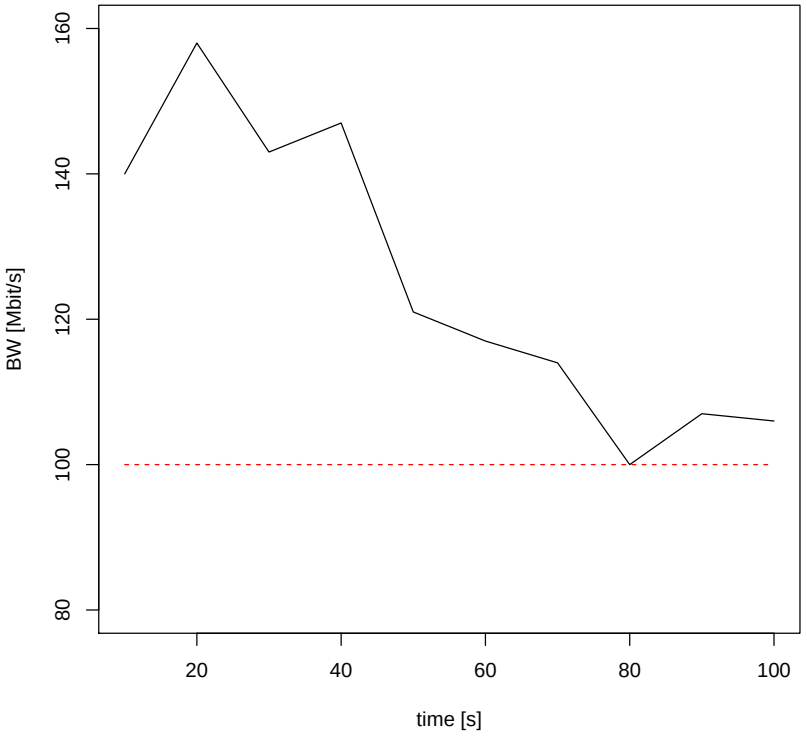

Figure 6.5: *iperf* results

6.3 SYN Flood Defense Performance

This section evaluates how effective the defense system can resist against SYN flood attacks and compares them to an existing mitigation mechanism, the kernel SYN cookies, for reference. At first though, the impact of SYN floods against an unprotected server is determined. The attack consists of running *hping3* on the host, generating SYN packets with a predetermined waiting interval.

```
hping3 -i u1000 -S -q -p 80 --rand-source 192.168.0.10
```

u1000 implies that $1000\mu s$ should be waited between two packets. Or the other way around, a rate of ideally 1000 packets per second. After the generation was started, ten seconds are waited before starting measuring service availability, which is done by the client VM. In 0.5s intervals, 50 SYN packets are send to the server and response packets are captured. For this test, the server got four logical CPUs assigned, together with 2 GB of RAM. The results for different attack rates are visualized in Figure 6.6. At 10 packets

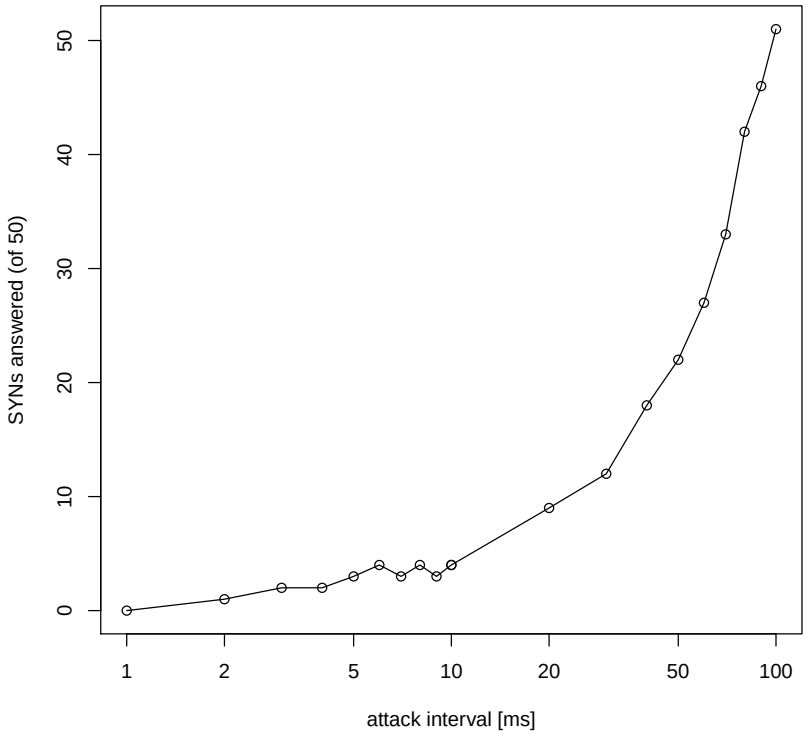

Figure 6.6: SYN flood impact without defense

per second, all of the client's SYNs are processed. The number drops with increasing attack rate, with 1ms attack delay, the service is practically unavailable for the client. This calculates to a theoretical rate of 1000 packet/s, however sending packets does not happen in zero time, so the more packets are sent, the more the actual packet rate varies from the specified waiting interval. The actual packet rate has been determined by running *hping3* with the *timeout* command, that is instructed to stop the packet generation after 10 seconds. Because *hping3* prints the amount of sent packets on termination, the packet rate of 1ms delay can be calculated to be around 930 packet/s.

The same methodology should have been used to determine how many packets the SYN cookies can withstand. However, even with the smallest usable delay of 1ms, every single

of the client's 50 SYN packets has been responded to. The attack rate with 1ms interval has been measured to be around 225,000 packet/s. *hping3* has the *–flood* option, which sends packets as fast as possible, which amounted to 375,000 packet/s on the test system. With that attack rate burdening the server, the client tried sending 50 SYN packets ten times. The average amount of answered packets was 35, the lowest 28 and the maximum 42. However, it is not possible to tell if the packets were dropped at the busy server or somewhere on the way between client and server.

The VNF DDoS defense system was tested in three different configurations. One time, the DPDK application had three cores assigned, thus one was polling the network card, one was processing packets and the other was sending packets. In the other configurations six and eight cores were available, so four, respectively six cores were processing packets. Again, ten runs consisting of 50 SYN packets each, have been run on the client, while the host was flooding. Results, together with the SYN cookie values are listed in the following table.

	SYN cookies	VNF 3 cores	VNF 6 cores	VNF 8 cores
Run 1	30	29	27	25
Run 2	37	34	22	23
Run 3	30	34	17	28
Run 4	41	25	37	27
Run 5	35	30	26	21
Run 6	38	29	27	25
Run 7	42	31	26	24
Run 8	27	29	23	30
Run 9	35	25	31	28
Run 10	35	34	24	26
AVG	35	30.2	26	25.7
MIN	28	25	17	21
MAX	34	34	37	30

Apparently the VNF did a bit worse than the SYN cookies. What is surprising however, is that more processing cores on average had inferior performance than a single core. This may indicate that the ringbuffer use for thread communication is preventing scalability. Or the networking in between might be the bottleneck and fluctuations were more favorable for the SYN cookies and lower core counts. Anyway, the results are not very meaningful and satisfactory, further testing with predictable packet generation and networking will be required.

6.4 Influence on Quality of Service

Because the time that is needed to inform server and network after a successful handshake, the first few packets of the client might be dropped, because they are still routed to the VNF and dropped there. This can even be seen in the validation example, Section 6.2. To measure its influence, the client accesses the Apache2 Ubuntu Default Page using *curl*, which can output the total loading time needed. Different packet transmission delays from the client to the server are considered as well and realized with *netem*, emulating the respective delay on every packet the client sends. The page is loaded 10 times for every delay and the average is used for plotting. Results for enabled DDoS protection and direct connection between client and server can be found in Figure 6.7. While the load times increase linearly with the delay for direction connections, with the VNF in between, delay increases have slightly more influence. In Section 6.2, the retransmission of the first data

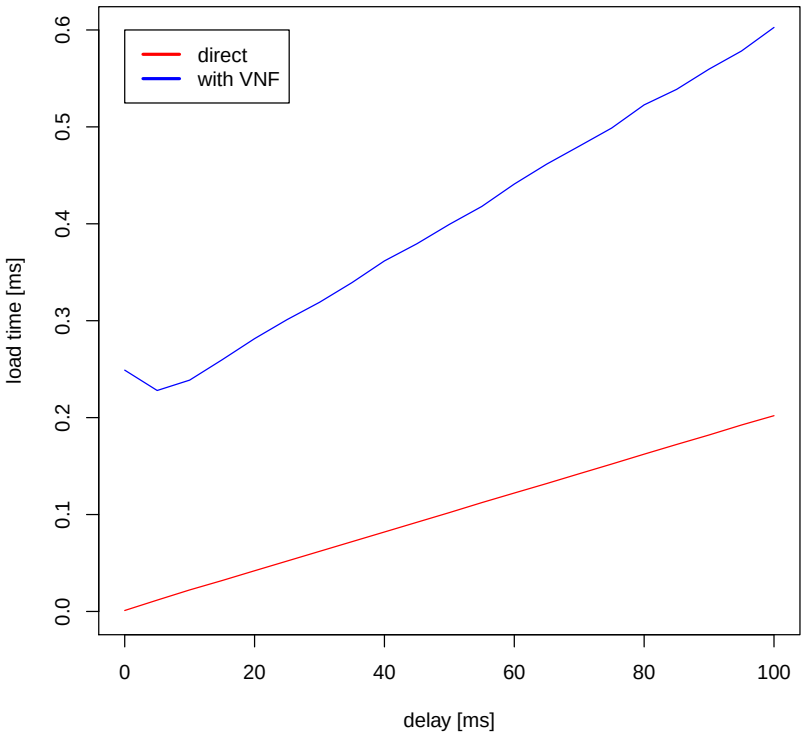

Figure 6.7: Page access timing results

packet happened after around 200 ms, which matches these results, showing 216 to 401 milliseconds longer page load times.

The increasing impact of delay with DDoS protection enabled probably was not because of the packet loss of the initial data, but rather because no TCP options are supported by the VNF. Many options, like MSS or window scaling, were introduced to support high bandwidth, high ping scenarios. With available bandwidth being very high, the direction connection with its enabled options is probably still faster, even with only moderate delays. To test bandwidth limitations more thoroughly, *iperf* was used again to determine maximum bandwidth for both cases. The throttling threshold of the HTTP flood protection has been set to 10 GBit/s to prevent it from interfering. Like in the previous measurement, the delay is applied to packets transmitted by the client. The graph in Figure 6.8 demonstrates the impact of missing TCP options. Without additional emulated delay, the bandwidth with enabled VNF is not too far off the direct connection, but with just 5 ms more the bandwidth drops from 814 Mbit/s to merely 76 MBit/s. From there it falls further to only 2 Mbit/s with 100 ms delay. The direction connections are also affected by the delay, the decline is less step, though. Up to 35 ms it is still able to transport more than 200 Mbit/s (with VNF: 13 Mbit/s). Even at 100 ms the throughput still amounts to 79 Mbit/s.

Overall, the DDoS protection in its current form noticeably degrades quality of service with its simple implementation of SYN cookies which does not support any TCP options. This effect could be lessened by using a more sophisticated cookie generation algorithm, like the one used in the Linux kernel. Another technique, also used with kernel SYN cookies, is on demand use. Thus, the VNF in a non-attack situation only monitors and forwards packets, checking how many SYN-ACK packets remain unanswered. When an

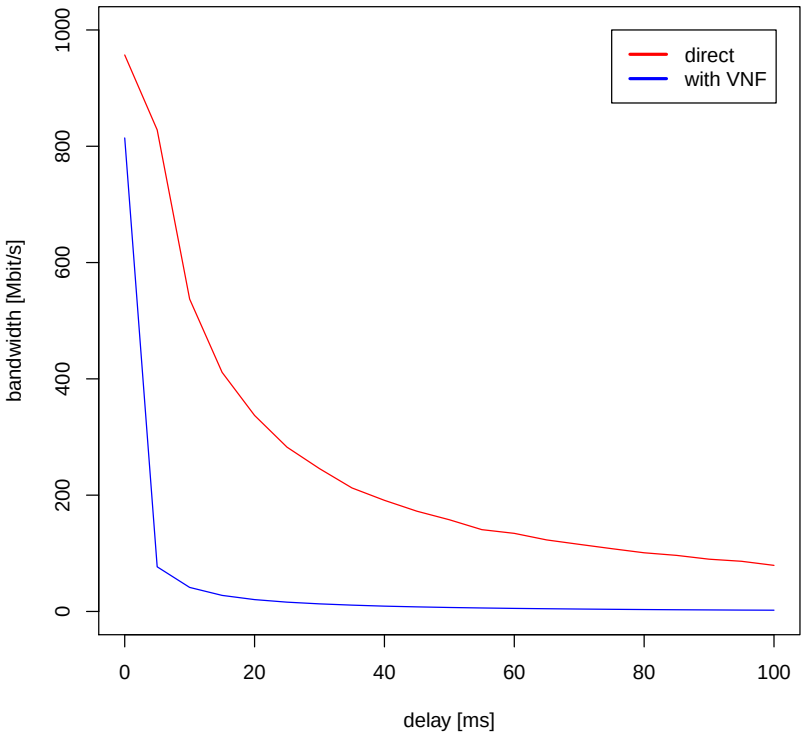

Figure 6.8: Bandwidth measurement results

attack is observed, the VNF changes to remote handshake behavior. In the end, in an attack scenario it is preferable to serve connections with inferior quality of service than not serve them at all.

7. Conclusion

This chapter concludes the thesis and presents the most important findings. Cyber attacks are a major threat to the infrastructure modern society is based on. One group of cyber attacks are the DDoS attacks with thousands and millions of attackers. In this work, we focused on mitigation of the effect of these attacks.

We introduced the required technical background, giving an overview of typical DDoS attacks. Furthermore the concept of Software Defined Networking was introduced with the example of the OpenFlow standard. Then, the possibilities offered by Network Function Virtualization were described.

Next, available related work was reviewed. An overview was given for the existing implementation in the Linux kernel, as well as other papers dealing with DDoS attacks.

Then, we introduced our approach of dealing with TCP SYN floods, as well as HTTP floods using a security VNF in a SDN enabled network. SYN requests are routed to the VNF until a connection has been established successfully, after which they are being rerouted to the actual target server using the SDN features. HTTP flood attacks are countered by splitting traffic into two different queues inside the network. Applications with a suspicious amount of traffic are sent via a bandwidth limited queue, while other connections with clients deemed benign are sent over an unrestricted queue, ensuring their Quality of Service.

The implementation was divided into three parts. First, we modified the Linux kernel to be able to force it to use specific sequence numbers for TCP connections. Second, we implemented an application inside the DPDK framework, realizing the handshake with the client and, in case of success, with the server, thus protecting the server from spoofed SYN requests. Third, a Python application realizes the communication of the security VNF with the SDN controller, alongside the analysis of flow statistics for the defense against HTTP floods.

Finally, we presented the evaluation of our approach. Therefore, we introduced a testbed consisting of a miniaturized realistic network environment. Machines to represent valid clients, attackers, the security VNF, an SDN controller and the application server are put in place. The network on the service side is SDN enabled. We performed a behavior validation, showing our approch works as designed. Next, we executed performance tests for the SYN flood defense, indicating promising performance, comparable with established

in-kernel protection features, proving our system as a valid alternative to competing solutions. Also, influence on connection's Quality of Service was examined, disclosing a noticeable impact, that however may be a worthy trade in heavy attack scenarios.

Future work may deal with the evaluation of the proposed approach in a larger environment, which should provide more comprehensible results. Furthermore, the introduction of further multi-threading to improve the performance for high load situations and its effect on the scalability is to be evaluated. Next, the approach could be integrated in a solution for intelligent function chains for security environments, reporting the number of attacks to a central instance to facilitate automatic reconfiguration based on models of the traffic, as well as the security appliances. For better reusability, the modification changes to the kernel could be converted into a separate module and introduced to the mainline kernel source.

Bibliography

[BBCC14] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 44–51, 2014.

[BNR+17] J. Boite, P.-A. Nardin, F. Rebecchi, M. Bouet, and V. Conan, "Statesec: Stateful monitoring for ddos protection in software defined networks," 2017.

[Bor12] D. Borman, "Tcp options and maximum segment size (mss)," 2012.

[D+04] L. Deri *et al.*, "Improving passive packet capture: Beyond device polling," in *Proceedings of SANE*, vol. 2004. Amsterdam, Netherlands, 2004, pp. 85–93.

[DYL+12] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, pp. 1471–1480, 2012.

[dyn] [Online]. Available: http://www.bbc.com/news/technology-37728015

[Edd07] W. M. Eddy, "Tcp syn flooding attacks and common mitigations," 2007.

[FM15] C. J. Fung and B. McCormick, "Vguard: A distributed denial of service attack mitigation method using network function virtualization," in *Network and Service Management (CNSM), 2015 11th International Conference on.* IEEE, 2015, pp. 64–70.

[GEW+15] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet io," in *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on.* IEEE, 2015, pp. 29–38.

[HR06] M. Handley and E. Rescorla, "Rfc 4732: Internet denial-of-service considerations," 2006.

[Int14] D. Intel, "Programmers guide," 2014.

[Int15] ——, "Data plane development kit," 2015.

[JP13] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: a survey," *IEEE Communications Magazine*, vol. 51, no. 11, pp. 24–31, 2013.

[JYR+16] A. Jakaria, W. Yang, B. Rashidi, C. Fung, and M. A. Rahman, "Vfence: A defense against distributed denial of service attacks using network function virtualization," in *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, vol. 2. IEEE, 2016, pp. 431–436.

[KSS14] C. Kachris, G. Sirakoulis, and D. Soudris, "Network function virtualization based on fpgas: A framework for all-programmable network devices," *arXiv preprint arXiv:1406.0309*, 2014.

[llo15] "Lloyd's CEO: Cyber attacks cost companies $400 billion every year," 01 2015. [Online]. Available: http://fortune.com/2015/01/23/cyber-attack-insurance-lloyds/

[MAB+08] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[MAR+14] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2014, pp. 459–473.

[P+81a] J. Postel *et al.*, "Rfc 791: Internet protocol," 1981.

[P+81b] ——, "Transmission control protocol rfc 793," 1981.

[PNFR15] M. Paolino, N. Nikolaev, J. Fanguede, and D. Raho, "Snabbswitch user space virtual switch benchmark and performance optimization for nfv," in *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE, 2015, pp. 86–92.

[Pos83] J. Postel, "Rfc 879: The tcp maximum segment size and related topics," *Internet Engineering Task Force (IETF)*, 1983.

[Riz12] L. Rizzo, "Netmap: a novel framework for fast packet i/o," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 101–112.

[Sal05] J. H. Salim, "When napi comes to town," in *Linux 2005 Conf*, 2005.

[syn] [Online]. Available: https://rhelblog.redhat.com/tag/synproxy/

[tcp] [Online]. Available: http://telescript.denayer.wenk.be/~hcr/cn/idoceo/tcp_header.html

[TMMR16] M. Trevisan, M. Mellia, M. Munafò, and D. Rossi, "Dpdk-stat: 40gbps statistical traffic analysis with off-the-shelf hardware," *Technical report*, 2016.

[WYK+14] J. P. Walters, A. J. Younge, D. I. Kang, K. T. Yao, M. Kang, S. P. Crago, and G. C. Fox, "Gpu passthrough performance: A comparison of kvm, xen, vmware esxi, and lxc for cuda and opencl applications," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. IEEE, 2014, pp. 636–643.

Appendix

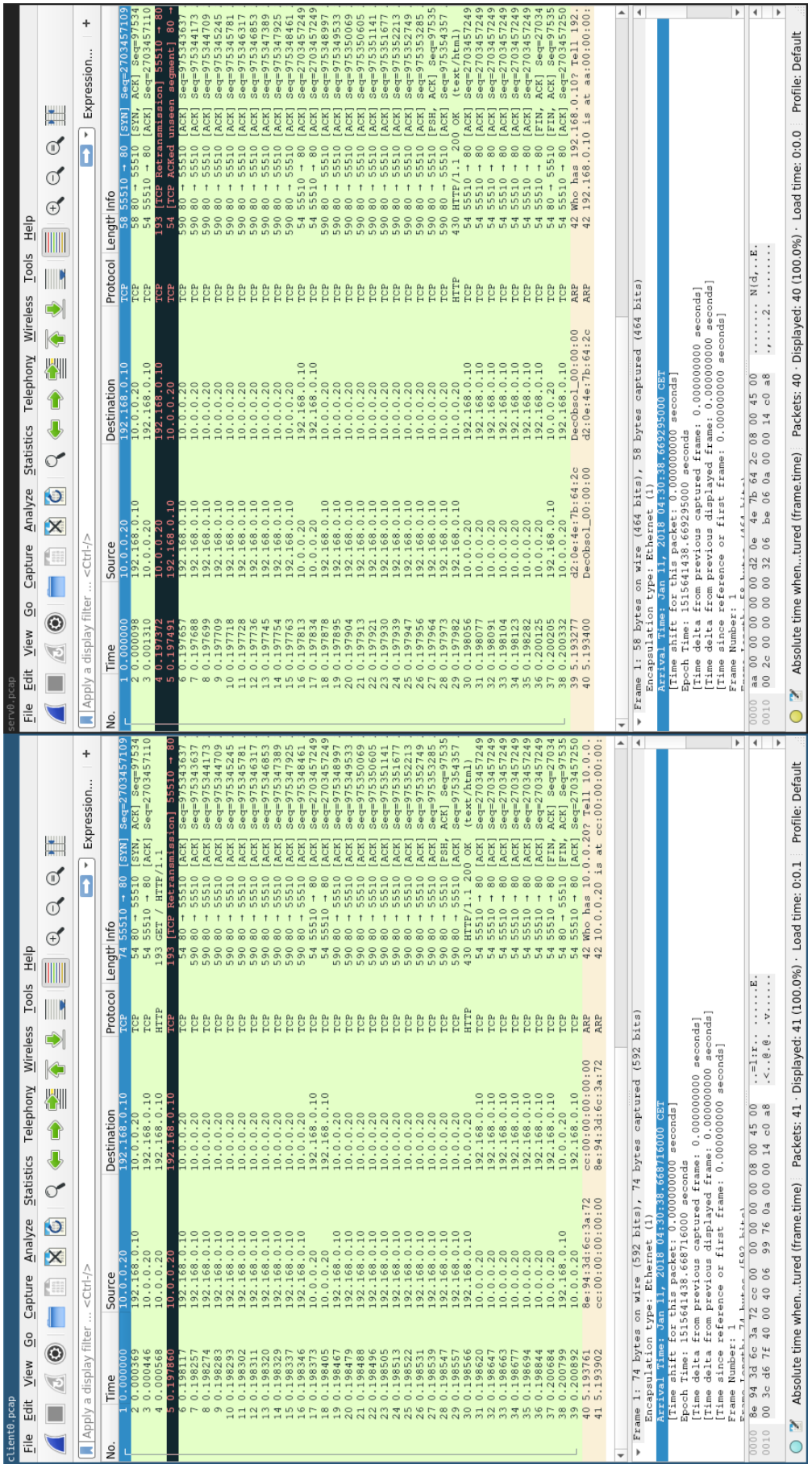

Figure .1: Complete packet captures at client and server