NTNU

Norwegian University of
Science and Technology

Software Defined Data Flow Isolation by Virtualization and Cryptographic Key Distribution

Steffen Fredrik Birkeland

Master of Science in Communication Technology

Submission date:	June 2016
Supervisor:	Stig Frode Mjølsnes, ITEM
Co-supervisor:	Christian Tellefsen, Thales

Norwegian University of Science and Technology
Department of Telematics

Title: Software Defined Data Flow Isolation by Virtualization and Cryptographic Key Distribution

Student: Steffen Birkeland

Problem description:

OpenFlow is a widely used protocol in Software Defined Networking (SDN). Transport layer security (TLS) is used for communication security between the SDN controller and each of the OpenFlow switches. OpenFlow allows for multiple virtual networks to run simultaneously over one physical network, by using unprotected VLAN ID tags in the data packets. The explorative question for this master thesis assignment is: Find out whether it is feasible and practical to use mechanisms of software runtime virtualization and cryptographic key distribution within an SDN architecture to achieve cryptographic separation/isolation between dynamic virtual networks. 'Dynamic' means that the data flow isolation through the network can be changed by network controller applications and policy. First, set up an experimental SDN testbed that enables datapath encryption and authentication using available open source software. Second, design a solution that may achieve software defined data flow isolation. Third, analyse and test the security (informational isolation) and the performance/efficiency of the proposed solution. Finally, if time allows, investigate the feasibility of this solution with openWRT-based wireless routers.

Professor: Stig Frode Mjølsnes, ITEM

Supervisor: Christian Tellefsen, THALES

Abstract

OpenFlow is a widely used protocol in Software Defined Networking (SDN). Transport layer security (TLS) is used for communication security between the SDN controller and each of the OpenFlow switches. However, OpenFlow does not provide any cryptographic security through OpenFlow.

This thesis explores the possibility of adding encryption to the datapath that can be controlled from a Software Defined Networking (SDN) controller. A virtual testbed is created using Pox, Open vSwitch (OVS), and Virtualbox. In the virtual testbed, different encryption concepts are tried out, and related performance testing is performed. Then, the solution is ported to a physical network consisting of a computer, two Raspberry Pi devices, and a router. A replay attack was tested on Generic Routing Encapsulation (GRE) and Internet Protocol Security (IPsec). The performance overhead from encryption and Pre Shared Key (PSK) renewal was evaluated. Some leaking traffic was discovered when changing PSK. Different ways of changing the PSK were tried out and evaluated. The best solution turned out to be adding new tunnel endpoints with a new PSK.

Sammendrag

OpenFlow er en protokoll som er mye brukt i SDN. Transport Layer Security (TLS) blir brukt i kommunikasjonen mellom SDN kontrolleren og OpenFlow switchene, men trafikken mellom switchene har ingen kryptografisk sikkerhet gjennom OpenFlow.

Denne oppgaven ser på muligheten for å legge til kryptering av datatrafikk i SDN som kan kontrolleres av kontrolleren. Et virtuelt testmiljø ble bygget ved bruk av Pox, OVS og Virtualbox. I det virtuelle miljøet ble krypteringskonsepter testet ut, og ytelsesmåling. Så ble oppsettet flyttet over på et fysisk nettverk bestående av en computer, to Raspberry Pi enheter og en ruter. Et 'replay attack' ble testet på både GRE og IPsec. Ytelses målinger relatert til kryptering og PSK utskiftninger ble utført. Ved utskiftning av PSK ble det funnet sårbarheter. Noe trafikk lekker ut under utskiftning av PSK. Forskjellige metoder for å skifte ut PSK på ble testet og evaluert. Den beste løsningen var å legge til en ny port med en ny PSK.

Preface

This work is my Master's thesis in Communication Technology at the Norwegian University of Science and Technology specializing in Information Security.

I would like to thank my supervisor Christian Tellefsen and Professor Stig Frode Mjølsnes for feedback and valuable guidance during this thesis.

I would like to thank Scott's Weblog @ blog.scottlowe.org for providing valuable insight into Open Vswitch and the underlying mechanisms that helped me greatly.

Contents

List of Figures

List of Tables

List of Acronyms

AH Authentication Header.

AP Access Point.

ARP Address Resolution Protocol.

CPU Central processing unit.

DHCP Dynamic Host Configuration Protocol.

DPID Datapath Identifier.

GRE Generic Routing Encapsulation.

HDMI High-Definition Multimedia Interface.

ICMP Internet Control Message Protocol.

IP Internet Protocol.

IPsec Internet Protocol Security.

IPsec_GRE Gre tunnel using IPsec.

MAC Media Access Control.

MTU Maximum transmission unit.

NTNU Norwegian University of Science and Technology.

OpenFlow OpenFlow.

OS Operating system.

OSPF Open Shortest Path First.

OVS	Open vSwitch.
OVSDB	Open vSwitch Database Management Protocol.
PSK	Pre Shared Key.
QOS	Quality of service.
RTT	Round-trip delay time.
SAD	Security Association Database.
SDN	Software Defined Networking.
SPD	Security Policy Database.
SSH	Secure Shell.
TCP	Transmission Control Protocol.
TLS	Transport Layer Security.
VM	Virtual Machine.
VPN	Virtual Private Network.

Chapter 1

Introduction

1.1 Motivation

Securing data is crucial when sending it over a network. Not only to provide privacy but also for security and availability purposes. Securing data traffic is mainly achieved through encryption and isolation. When data is sent over an unsecured or uncontrolled network the data can only be secured with encryption. Data sent over a controlled network can be secured by segmentation and isolation. An example is Virtual Local Area Network (VLAN)s and Demilitarized Zone (DMZ).

With the introduction of SDN, a centralized controller can control the data flow on switches in the network. Today it is used for controlling the traffic management. However, this concept could also be applied to the handling of security. Although, when it comes to the security opportunities this provides, it has not yet been employed. As in many other cases, the security aspects have been looked past. Additionally, in OpenFlow (OpenFlow), an SDN protocol, the never versions enforce a less strict security policy. Using a cryptographically secure connection between the controller and switches was mandatory in OpenFlow1. However, in OpenFlow1.4 this became optional[ofsc][Sam15]. Additionally, there is no functionality to ensure that the traffic between the switches is encrypted.

Ensuring that data is encrypted in a data center or internal network might not be necessary as it is secured by isolation. However, as the SDN is evolving and being used outside of small controlled networks encryption will become a necessity. OVS and Cisco switches both support encryption tunnels. However, to my knowledge, there is no open source platform that supports a centralized controller that dynamically controls the security parameters along with changes in the data flow. To my knowledge there is no research on what opportunities this enables, and how it works on physical devices like a network router. There are several cases where this type of system would be interesting:

- If one were to use SDN over a wireless network; It would no longer be possible to physically isolate the traffic, and therefore encryption will be paramount.
- When you have multiple networks that are physically separated. An example of this could be a research network that needs to share information across countries. In this case, the traffic has to pass through open networks, and it would be desirable to ensure that all traffic leaving your controlled network would be encrypted.

  There is an argument that these problems can be solved by using end-to-end encryption, with solutions like Virtual Private Network (VPN) and TLS. Yet there are cases where this would not be sufficient:

- The problem with this is that it puts requirements on the 'end-equipment' to support the latest secure cryptography. We put more and more devices and technology online. With smaller devices the encryption becomes more expensive.
- The case might also be that the encryption the hardware supports is deprecated but you still need a backward compatible system to talk with it without having to rebuild the whole system.
- End-to-end encryption can provide authenticity and confidentiality, but not availability. If using an unprotected GRE tunneling protocol that uses sequence numbers, it is possible to inject arbitrary sequence numbers and launch a Denial of Service (DOS) attack.
- You introduce a smaller attack surface. The probability of configuration errors will decrease as this can be handled in one place.
- As technology gets more complex, the danger of bugs also increases. An example is wireless routers. They are known for having security vulnerabilities [CR08]. Patching bugs require the users to update their software which is also known not to be done enough. Adding encryption to the network cloud be, if not an alternative, at least an addition to end-to-end encryption.
- Lastly, you have the human factor. No matter how secure you system is, it is always vulnerable to weak passwords, and social engineering. By adding encryption to the network the only human factor would be the system engineer.

1.1.1 Scope and Objectives

The overall goal of this thesis is to explore how existing open source SDN technology can be used to extend an SDN framework to handle a centralized control of encryption. The scope of this project is to focus on the datapath performance and security, and the communication between OVS and the controller.

1.1.2 Objectives

- The first part will consist of exploring SDN, how it works and how encryption can be dynamically deployed.
- As this is a relatively new subject, the next focus is on implementing the solution consisting of: exploring what configurations are possible, what technology to use, verifying that it works, and test the performance.
- The final part consists of implementing it on a physical network and evaluate the security and performance.

1.1.3 Methodology

The methodology in this work has mainly consisted of experimentation and reading. In the beginning, there was a steep learning curve as both SDN and networking in Linux were new subjects to me. There were no tutorials and little information about the underlying mechanisms. Most of the discoveries were done by looking through the source code. After successfully getting to a solution, much of the experiments conducted were based on the result in the previous experiments.

1.1.4 Contribution

To my knowledge, only one other paper has looked at on encryption of the datapath in SDNs [GK14]. The hope is that the work presented here can be built on to create a full-worthy cryptographically secured SDN. Much of this thesis has consisted of using existing software in new ways, resulting in the discovery of bugs that were reported. The wireless Raspberry Pi performance experiments with encapsulation protocols in conjunction with OpenFlow has as far as I know not been performed before.

1.1.5 Clarifications

Some word used in this paper need some clarification, as they have a specific meaning here that might be different from what they ordinarily would mean.

- When referring to a switch I mean a Linux device with network connectivity running OVS. OVS can make its hosts device become a remotely controllable software switch.
- A *port* is an OpenFlow port. This is either a physical interface, or a virtual interface created by OVS.

- When referring to a tunnel, I mean two ports that function as endpoints on an encapsulation tunnel.
- *Key* is only referring to a number that is used for separating tunnels. The key that is used for encryption and decryption is referred to as PSK.
- Gre tunnel using IPsec (IPsec_GRE) means a GRE tunnel that uses IPsec to encrypt the payload.
- gre-between-gre and ipsec-between-ipsec: means an IPsec/GRE tunnel connecting each switch.
- gre-in-gre and ipsec-in-ipsec: means packets in an IPsec/ GRE tunnel encapsulated in another IPsec/ GRE tunnel. As discussed in 3.2.4.

Chapter 2 Background

2.1 SDN

SDN is an architecture where the control plane is decoupled from the forwarding plane, allowing for an abstraction of the underlying infrastructure from the application point of view. The components in SDN consist of a controller and switches, see Figure 2.1. The controller can manage the forwarding plane on the switches in the network. This centralized control allows the controller to have a global view of the network and create customized flow paths, which adds functionality to the network[onf]. This increased control is useful for security, congestion control, load balancing and more effective utilization of the network resources. Another goal with SDN is to use open standards and be vendor neutral.

To get a clear understanding of SDN, we can compare it to how a traditional network works. A traditional network will consist of routers and switches. Packet routes are decided by using a routing table and routing protocol. A router will be connected to two interfaces or more, and the routing table will be used to select which interface to forward the packet to. The routing protocol is what is used to create the routing table. The routing table is what that decides how the packet is forwarded. If each router in Figure 2.1 implemented Open Shortest Path First (OSPF), the traffic from `H1` to `H2` would go from `s1` to `s3`. However, with SDN, we could tell the switches to send the traffic through `s1 - s2 - s4 - s3`, or `s1 - s2 - s3`. The case could be that the link speed through `s1 - s2 - s3` would be a lot faster, or that `s1 - s3` is heavily congested, or that the information was sensitive, and `s1 - s2 - s3` were the only secure lines. More on SDN can be found at [ope].

2.2 OpenFlow

In SDN a communication protocol is used to send messages between a switch and a controller. This communication protocol is used to configure the forwarding plane on the switches and pass information to the controller. In this thesis, I will use

In_port	Port that the packet arrives on
dl_src	802.3 MAC
dl_dst	802.3 MAC
dl_vlan	802.1Q ID
dl_type	802.3 EtherType
nw_proto	IPv4 Protocol
nw_src	IPv4 Source
nw_dst	IPv4 Destination
tp_src	TCP/UDP Source Port
tp_dst	TCP/UDP Destination Port

Table 2.1: Used OpenFlow match fields

OpenFlow. OpenFlow is increasingly being used in production systems. Cisco, HP, and Juniper are offering OpenFlow support, and Google uses it in their datacenter backbone network[ofab].

A switch that uses OpenFlow forwards packets based on flow tables. The flow tables consist of matching rules and corresponding actions. The matching rules are used to decide what fieldvalues that should be used to determine appropriate action. A complete list of fields used for matching can be found at [ofsd]. Table2.1 shows the matching rules I have utilized in this thesis.

Note that In_Port is the OpenFlow port. Matching rules can rely on multiple fields and use wildcarding. Together with the matching rules follow the action rules. These can change a packet header or output port. As with the matching rules these can be chained so that multiple operations can be applied to a packet. A complete list of actions can be found at [ofaa]. If the packet does not match any entries in the flow table, it is sent to the controller. Then the controller can decide what to do with the packet and send out an OpenFlow rule to the switch so the subsequent packets will follow the same pattern.

There are two typical ways the controller installs OpenFlow rules. Reactive and proactive. Reactive means that the switch does a table lookup to see if the packet matches any of the OpenFlow rules. If not, it sends that packet to the controller that then decides what OpenFlow rules to install on the switch. Proactive, on the other hand, is that the controller populates the flow tables in the switch before the relevant packet enters. The matching OpenFlow rules can be either static or they can have a hard or idle timeout. Hard timeout means that the OpenFlow rule will expire after a given amount of time. Idle timeout means that the OpenFlow rule will expire if it not used for a given amount of time.

The channel between the controller and switches can and should be secured. For OpenFlowV1 using TLS for the communication was mandatory. However, for the newer specifications, this is not required. Also according to [ofv] most vendors have skipped the TLS implementation, and OVS is the only one with full TLS support. The OpenFlow protocol does not provide any specification for encryption of the communication between the switches. I will attempt to use the SDN framework with OpenFlow to explore this feature.

2.2.1 Paramiko

'Paramiko is a Python (2.6+, 3.3+) implementation of the SSHv2 protocol [1], providing both client and server functionality. While it leverages a Python C extension for low-level cryptography (Cryptography), Paramiko itself is a pure Python interface around SSH networking concepts [parb].'

I use Paramiko to configure the OVS from the controller. Basically, I execute terminal commands over Secure Shell (SSH). There is, as I became aware of later, an Open vSwitch Database Management Protocol (OVSDB) that supposedly can be used to configure the OVS configuration remotely. In hindsight, it cloud have been a useful tool, and is suggested in my future work. However, as shown in 5.2.9, some additional commands are needed to ensure security. These could not have been executed with OVSDB. Additionally, SSH commands can be used on any switch that can install an sshserver and has an Command Line Interface (CLI).

The code I used is the following, and can be found in the appendix:

```
# Create ssh object
ssh = paramiko.SSHClient()
# To accept unknown Hostkeys
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
# connectinng to the host this would be equvalent to writing:
# ssh <Username>@<IP address>
# <User password>
ssh.connect(<IP address>, username=<Username>,
password=<User password>)

# Executes the command on the host.
# stin, the command that is sent
# out, the output form executing the command
# Err, error message
stin, out, err = ssh.exec_command(<command>)

# If an error occurred, this would be shown in the err object
# So I check if err contains a message
if(err.read()): print '[exec_ssh_command (error)] %s !!!!!!!!
```

```
Error '%(err.readlines())

# Close the ssh connection
ssh.close()
```

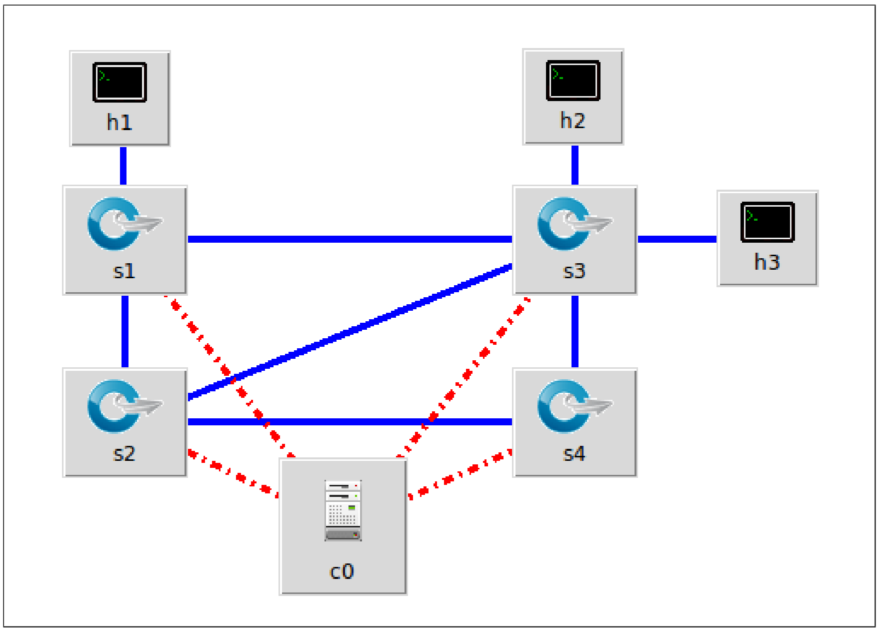


Figure 2.1: Example SDN architecture

2.3 Mininet

Mininet is a network emulator that allows building virtual networks[mina]. In this thesis, I have used a Mininet image running on a VM in Virtualbox. Mininet has built-in OpenFlow and OVS support, in addition to a Wireshark that can filter and display the OpenFlow packets in human readable language. For testing and exploring SDN Mininet was initially used. However, due to the way tunneling works in OVS, Mininet was not suitable to use in this thesis. The reason is that all the switches are on the host machines loopback interface, and OVS tunnels use the hosts network stack to send the packets. This caused some confusion as to why the setup did not work. As a result, I changed over to just use OVS and separate VMs in Virtualbox.

2.4 Generic Routing Encapsulation (GRE)

GRE is a tunneling protocol that encapsulates packets over IP networks[gre]. It works similarly to VPN in the sense that it creates a point-to-point connection. Packets that are being sent over the GRE tunnel are encapsulated at the tunnel entry-point with new headers. The GRE packet has the form: Delivery Header, GRE Header, and Payload packet. Every hop between the endpoints only use the added headers for routing and forwarding. At the endpoint, the encapsulating header is removed, and the original packet is routed from there on. GRE is often used for securing the transport of packets through a public network together with VPN, or to create a bridge between separated networks. It is important to know that GRE in itself does not provide any encryption of the encapsulated data. It can provide a checksum for the GRE header, and the payload. But this needs to be specified. [gre].

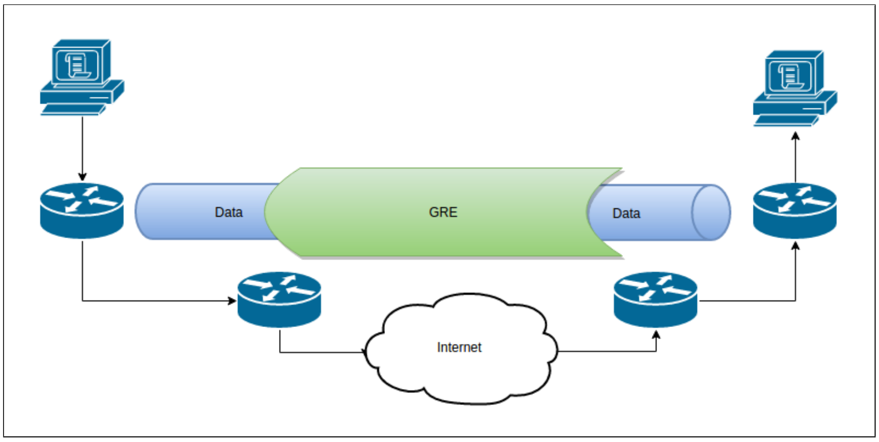


Figure 2.2: GRE model

2.5 Ipsec

To provide the encryption of data traffic I will focus on IPsec. IPsec is a protocol suite used for encryption and authentication of IP packets. The protocol works on the network layer (L3) and is end-to-end. Consequently, all traffic is encrypted, independent of protocol and traffic type, allowing transparent data encryption in the sense that the end client does not know about it. This makes it a good way to secure legacy systems that don't support encryption.

IPsec has two modes of operation. Tunnel mode and transport mode. In transport mode the only the payload that is encrypted, resulting in the packet being handled as it normally would. An Authentication Header (AH) can be used to secure the packet header. However, this puts a restriction on the network as any changes made to the header would invalidate the packet. You could disable the AH, but then you leave yourself vulnerable to replay attack [ipsa].

Consequentially tunnel mode is the better choice for IPsec in SDN. When using tunnel mode, the whole packet including headers is encrypted and then encapsulated into a new packet see Figure 2.3. The encapsulation will hopefully let the packet be handled like any other packet in the SDN.

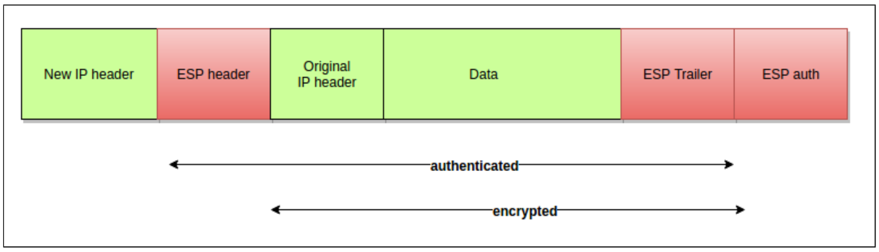


Figure 2.3: IPsec encapsulation

There are two important databases used in IPsec. Security Association Database (SAD) and Security Policy Database (SPD). In short the security policy is used to decide when to use IPsec encryption, and the security association is used to decide how to do the encryption. To establish a Security Association (SA) a keying daemon can be used. OVS uses racoon. Racoon uses Internet Key Exchange (IKE) to set up the SA. More about IPsec and IKE can be found here [ipsb]

2.6 Open vSwitch

OVS is a virtual switch that is OpenFlow enabled, meaning that it can be accessed by a remote OpenFlow controller that can configure the OpenFlow settings on the switch [ovsa]. OVS also has built in various tunneling protocols including `gre_ipsec` which I use in this thesis. The reasons for choosing OVS as OpenFlow switch are:

- Open source, and works well in virtualized environments
- Compatible with most linux-based environments
- Easily deployable.
- Widely used in SDN research
- Supports encrypted OpenFlow ports.
- Supports encrypted OpenFlow controller channel.

2.6.1 IPsec in Open Vswitch

A full list of OVS supported tunneling protocols can be found here [ovsb]. The relevant ones in this thesis are `ipsec_gre` and `gre`. These also have an equivalent

64-bit key version called `gre64` and `ipsec_gre64`. This key, however, is only used for administration purposes to differentiate tunnels, and must not be confused with the PSK used for encryption [rfc]. Each endpoint of the tunnel must be uniquely identifiable to work. The key field is for this purpose. It is useful if you want to logically separate traffic that goes over the same GRE tunnel. The same counts for the IPsec_GRE tunnel. The ipsec_gre tunnel supports both PSK and certificate for encryption. In this thesis I will focus on the use of PSK for encryption. A full list of the configuration options for IPsec in OVS can be found here [ovsb].

2.6.2 Traffic Routing with OVS

When using OVS on a host, it affects how the host handles traffic. An OVS consist of bridges and ports connected to those bridges. A port can be thought of as an interface except that a port can contain multiple interfaces. These ports are what is being referred to as OpenFlow ports. The interfaces that belong to a port can be virtual or physical. Adding a physical port to OVS will affect all traffic passing through that interface. It is important to realize how OVS handles the traffic to be able to do the correct configurations.

Physical interface

When adding a physical interface to an OVS port, all the incoming traffic on that interface is handled by OVS. Including Address Resolution Protocol (ARP) messages. What this means is that unless specified in the OpenFlow rules on that switch, the switch will not respond to ARP requests, but instead send it to the controller. This can lead to confusion when attaching interfaces to OVS, because hosts will experience connectivity loss. It can be resolved by adding the `action=NORMAL`, which will let the OVS act as a layer 2 switch. However, if you want layer 3 connectivity, you will need to set this up with the controller.

When forwarding a packet between two physical interfaces, none of the packet headers are changed unless specified. This can cause some problems when using the ARP protocol. As seen in Figure 2.4 Host2 will automatically forward the ARP request to Host3, who will answer the request. However, it will not be seen on any of the devices, because it is sent to MAC: AA which is on another network. Even setting the Host 2 to change the source MAC address before sending it to Host 3 will not work, because it only affects the Ethernet header and not the MAC address in the ARP request, which is the one Host3 will use for its reply. This can be seen in Figure 2.6. Therefore, all ARP messages should either be reactively handled by the controller or proactively when setting up the connection.

Assuming that each host knows each others correct MAC address, they also need to change the destination MAC address when forwarding packets. This has

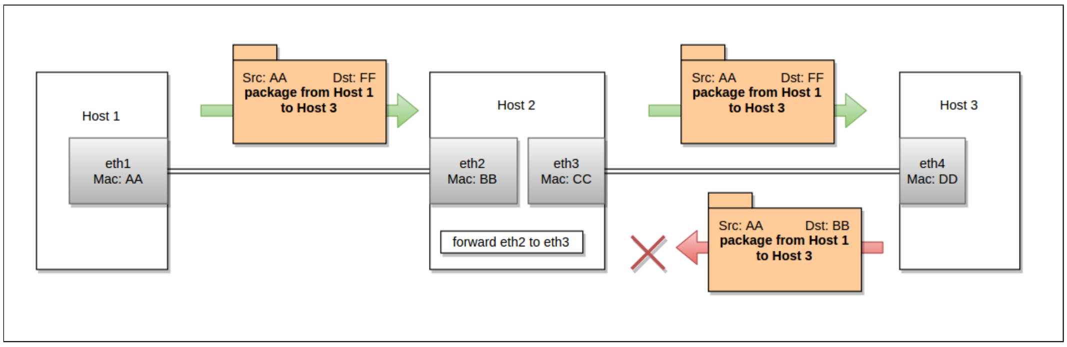


Figure 2.4: Arp discovery with interfaces connected to Ovs

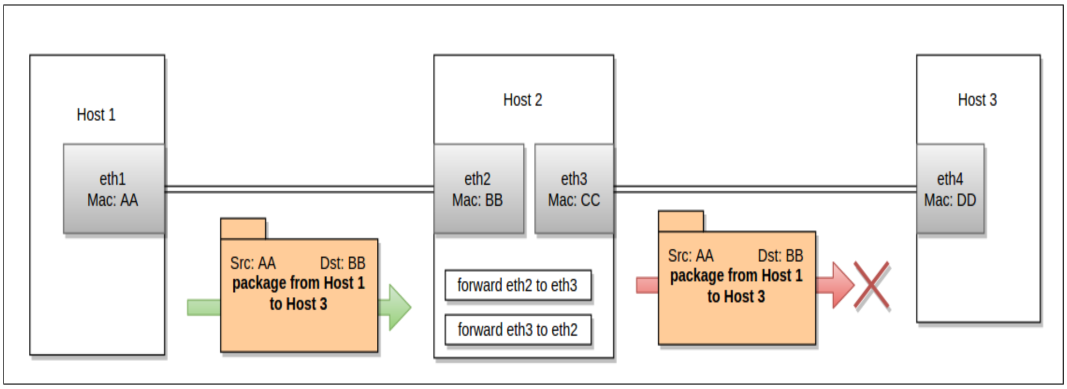
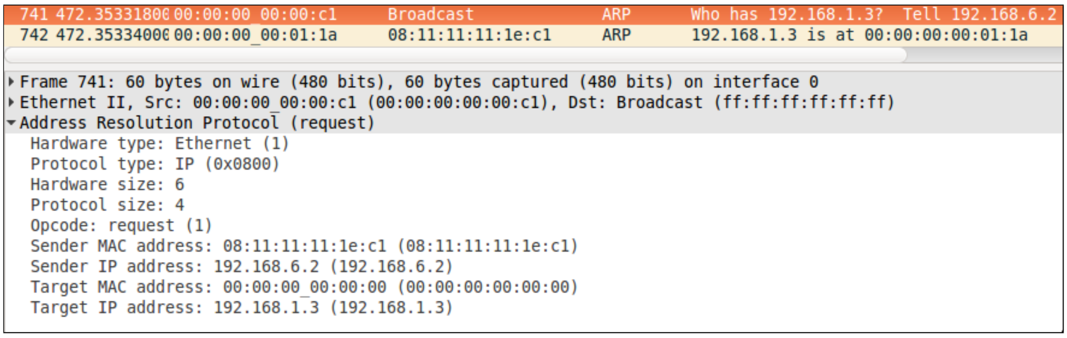


Figure 2.5: Ping problem when physical interfaces are connected to OVS

to be specified in the OpenFlow rules. As shown in Figure 2.5 Host 3 will not receive the packets sent by Host 2, if it still has Host 2s MAC address, because it only listens for packets with destination MAC *dd*. The target MAC can be changed in the OpenFlow `actions`. Note that the OpenFlow output port needs to be the last action set. Anything set after output will not be added. The command to change destination MAC address would be: `sudo ovs-ofctl add-flow in_port=<in_of_port_nr>,actions=mod_dl_dst:<mac-address>,output:<out_of_port_nr>` . In Figure 2.7 the OVS on Host 2 has installed the necessary OpenFlow rules for handling the MAC addresses, and can sucessfully forward the packets.

OVS GRE

The scenarios presented above, are all related to OVS using the physical ports on the host. The upside of using the physical interface in OVS is that we have full control over everything that is sent through. It also adds complexity to our configuration. When using the GRE and IPsec_GRE tunnels in OVS, we are using virtual interfaces.



```
741 472.3533180( 00:00:00_00:00:c1    Broadcast           ARP     Who has 192.168.1.3?  Tell 192.168.6.2
742 472.3533400( 00:00:00_00:01:1a    08:11:11:11:1e:c1   ARP     192.168.1.3 is at 00:00:00:00:01:1a

▸Frame 741: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
▸Ethernet II, Src: 00:00:00_00:00:c1 (00:00:00:00:00:c1), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▾Address Resolution Protocol (request)
  Hardware type: Ethernet (1)
  Protocol type: IP (0x0800)
  Hardware size: 6
  Protocol size: 4
  Opcode: request (1)
  Sender MAC address: 08:11:11:11:1e:c1 (08:11:11:11:1e:c1)
  Sender IP address: 192.168.6.2 (192.168.6.2)
  Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Target IP address: 192.168.1.3 (192.168.1.3)
```

Figure 2.6: Wireshark capture that shows that the responder will respond to the address in the ARP packet, and not the OVS port that forwarded it

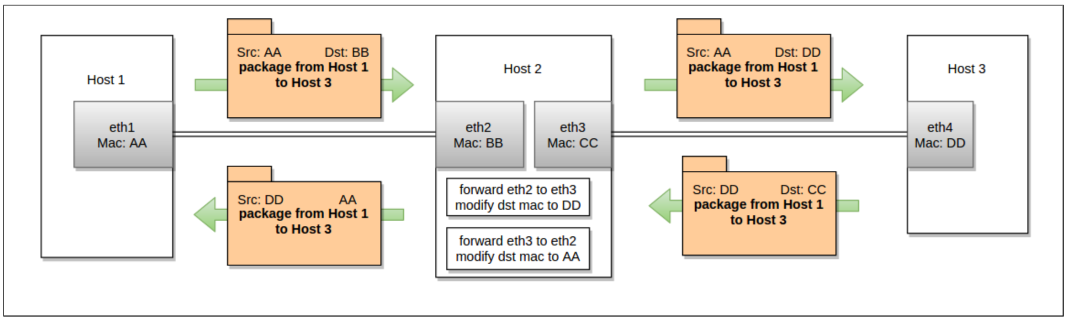


Figure 2.7: OVS is installed on Host 2, and controls eth2 and eth3. It uses the forwarding rules shown in Host 2, which modify the MAC addresses correctly.

These are added to our switch in the same way as the physical ones. However, they are virtual and only used by OVS. This means that after OVS sends the packet out on the virtual interface, the packet is forwarded using the hosts IP stack. Thus, other traffic not in the SDN, like control traffic, is unaffected when using virtual interfaces. Additionally, OVS does not need to handle MAC addresses in this case. This is handled by the hosts IP stack. However, the host is required to have the tunnel endpoint in its routing table. If it does not, it will silently drop the packets. When using virtual interfaces with OVS one does not need to change the headers, because this will be handled as normal by the hosts IP stack. The physical port that has the next step for an GRE tunnel can not be attached to the OVS. The reason is that the physical port is taken over by OVS and does therefore not belong to the hosts IP stack anymore. Figure 2.8 shows how OVS handles the GRE forwarding.

Ovs commands

In this thesis, administration of the OVS is done with `ovs-vsctl`, and `ovs-ofctl`.

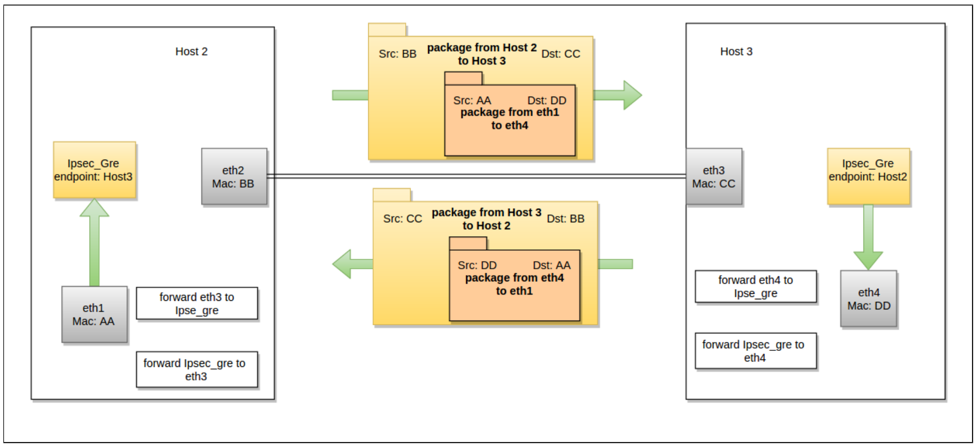

Host 2
Ipsec_Gre endpoint: Host3
eth2 Mac: BB
eth1 Mac: AA
forward eth3 to Ipse_gre
forward Ipsec_gre to eth3
Src: BB
package from Host 2 to Host 3
Dst: CC
Src: AA
Dst: DD
package from eth1 to eth4
Src: CC
package from Host 3 to Host 2
Dst: BB
Src: DD
Dst: AA
package from eth4 to eth1
Host 3
eth3 Mac: CC
Ipsec_Gre endpoint: Host2
forward eth4 to Ipse_gre
eth4 Mac: DD
forward Ipsec_gre to eth4

Figure 2.8: GRE IPsec tunnel example. The inner yellow box in Host2 and Host3 are the ports seen by OVS. The inner white boxes on Host2 and Host3 are the Openflow-rules used by OVS

`ovs-ofctl`, presumably short for Open vSwitch - OpenFlow Control, is a command line tool that is used for monitoring and managing the OpenFlow protocol part of OVS. It can do everything that the OpenFlow controller can, and is useful when debugging and troubleshooting. The full manual for ovs-ofclt can be found here [ovsc].

These are the most useful commands that were used:

`ovs-ofctl add-flow and ovs-ofctl mod-flows`

Used for adding and modifying flows directly on the switch. This is a great tool when an OpenFlow rule is not working correctly, and you need to modify it. The problem with using Pox is that it is made to react on events, meaning that in a test setting you need to reset the controller to apply a new configurations. This can be time-consuming, and sometimes you do not want to reset every switch in the network.

`ovs-ofctl dump-flows <ovs-name>`
gives you all the information about the flows installed on the switch. It is especially useful that it shows how many packets have been sent over each OpenFlow rule, making it easy to find out where the traffic is stopping when packet are not being forwarded. Additionally, it is useful to check for conflicting OpenFlow rules, and whether old OpenFlow rules were successfully removed.

`ovs-ofclt show <ovs-name>` displays the OpenFlow ports on the switch. The reason this was useful was that in OVS v 2.0.2 adding a port that was incorrectly configured, would not give any error message. Additionally, the `ovs-vsctl show` showed that the port was added, causing some confusion when setups were not working. The only way to detect that it was incorrectly configured, was to use `ovs-ofclt show` command and check if the port had a OpenFlow port number. If not, that indicated that the port was not actually added. OVS 2.0.2 is the version that is used in Mininet 2.2.1, which was used in the beginning of this thesis. In OVS v 2.3.1 this issue is fixed, so adding an incorrectly configured port will result in the message `ovs-vsctl: Error detected while setting up <portname>. See ovs-vswichd log for details`

`ovs-vsctl`, presumably short for Open vSwitch - virtual switch Control, is used for configuring the switch itself. It queries and applies changes to the Open vSwitch configuration database. All the configurations that were not OpenFlow configurations were done with `ovs-vsctl`. After installing OVS you can make the device work as an SDN switch with these three commands:

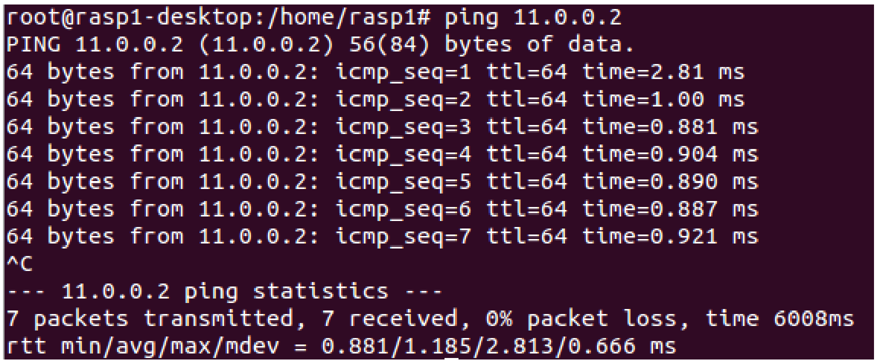
– `ovs-vsctl add-br <bridge name >` is used to add a bridge to OVS. All ports that you want to connect need to be on the same bridge.

– `ovs-vsctl set-controller <bridge name > <tcp | ssl > <ip address > : <port>` is used to connect the created bridge to a controller. When it is connected, the controller can communicate with the switch through the OpenFlow protocol.

– `ovs-vsctl add-port <bridge name> <port name>` is used to add ports to the switch. If you want to add a physical interface, you just use the name of that interface as port-name. If you want to add a virtual interface you can give it any name. However, you need to create the virtual interface and set it to the desired port. For instance: `ovs-vsctl set Interface <port name > type=< gre > options:remote_ip=<ip address>`

2.7 Iperf

To measure performance, I used Iperf. Iperf is a tool used to measure the maximum bandwidth in an IP network [ipeb]. The version used was 2.0.5. It is installed by running `sudo apt-get install iperf`. It works by setting up a server on one device and a client on another other. It supports both User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). In this thesis, I use TCP. Setting up the client and server is done in one line in a bash. The server is listening on a specified port. The client just has to specify the port and IP number for the server. More information about Iperf can be found here [ipeb].



```
root@rasp1-desktop:/home/rasp1# ping 11.0.0.2
PING 11.0.0.2 (11.0.0.2) 56(84) bytes of data.
64 bytes from 11.0.0.2: icmp_seq=1 ttl=64 time=2.81 ms
64 bytes from 11.0.0.2: icmp_seq=2 ttl=64 time=1.00 ms
64 bytes from 11.0.0.2: icmp_seq=3 ttl=64 time=0.881 ms
64 bytes from 11.0.0.2: icmp_seq=4 ttl=64 time=0.904 ms
64 bytes from 11.0.0.2: icmp_seq=5 ttl=64 time=0.890 ms
64 bytes from 11.0.0.2: icmp_seq=6 ttl=64 time=0.887 ms
64 bytes from 11.0.0.2: icmp_seq=7 ttl=64 time=0.921 ms
^C
--- 11.0.0.2 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6008ms
rtt min/avg/max/mdev = 0.881/1.185/2.813/0.666 ms
```

Figure 2.9: Output when running ping

2.8 Ping

To measure the latency I used ping. Ping is used for testing reachability and round-trip-time. It works by sending out ICMP packets. The sender sends out an ECHO_REQUEST and the target responds with an ECHO_RESPONSE. The packet contains an IP header, ICMP header, a time value, and then some pad bytes. The output from running the ping command can be seen in Figure 2.9. As seen **min** is the minimum, and **max** is the maximum. **avg** is the average time, which can be verified by summing up the times shown in Figure 2.9 and dividing by seven. I was unsuccessful in finding any documentation on what **mdev** was. Thus, I did the calculations using the numbers in Figure 2.9 on https://www.easycalculation.com/statistics/standard-deviation.php. The conclusion is that **mdev** is the population standard deviation. Indicating the spread of scores in a sample. More information about ping can be found here [pin].

2.9 MTU

Maximum transmission unit (MTU) is the largest data unit that can be passed on a network layer. MTU is usually used when referring to packet size in TCP. The size is specified in bytes and is typically 1500 in IP networks [cis]. The upside with increasing the MTU is less overhead per data. However, the MTU might differ between systems. Too large packets will lead to a retransmission or fragmentation.

2.10 Debugging

When setting up an SDN it will most likely be necessary with some debugging. The tools that I used were Wireshark, the ovs-ofctl command line tool, and the Pox controller. Wireshark was convenient because I most of the time used VMs.

However, when using clients that only have a command line interface, or are distributed remotely, it is possible to use SSH with X11 forwarding. This allows you to run the graphical interface of a program running on a remote host, on your local computer. X11 forwarding is achieved by simply adding an -X, when establishing the ssh connection:

```
ssh <user>@<ip address> -X
```

2.11 Wireshark

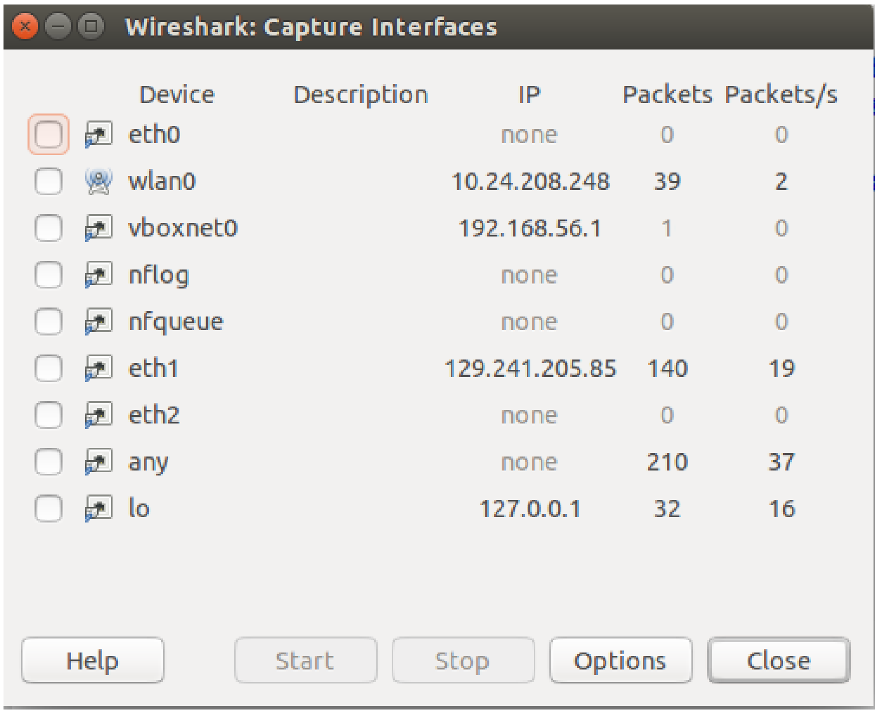
Wireshark is an open-source network protocol analyzer. It is widely used as a tool for troubleshooting, education, and network analysis. It displays what packets are being transmitted over each interface in an easily readable way. It was useful to see which interfaces that were sending and receiving traffic. Note that you need to run Wireshark as superuser for this to work. When opening Wireshark and pressing `Ctrl + I` you can see the packet activity on all interfaces as seen in Figure 2.10. This is very helpful when checking that forwarding rules are working, and the correct MAC addresses are used.

2.12 SDN controller

The SDN controller is written in software to control the connected switches, using a communication protocol like OpenFlow. This way the controller is not hardware dependent. The only restrictions are made by the programming language. Today you have controllers written in Java, Python, and Ruby. As I will illustrate in this thesis, you can easily extend the controllers functionality by using existing modules for the chosen language. I use the Pox controller. The reasons for choosing Pox was that it is considered easy to learn, is aimed towards research and education, python based, and there are good tutorials and examples. There are three functions that are especially important when it comes to setup and troubleshooting. All the controller realted code can be found at https://github.com/Steffb/MstThesis/tree/master/Pox.

launch

Launch is the first function that is called when you start the controller. I use this to run the static setup that creates the virtual port and adds the relevant ports to the OVS.


Wireshark: Capture Interfaces

Device	Description	IP	Packets	Packets/s
eth0		none	0	0
wlan0		10.24.208.248	39	2
vboxnet0		192.168.56.1	1	0
nflog		none	0	0
nfqueue		none	0	0
eth1		129.241.205.85	140	19
eth2		none	0	0
any		none	210	37
lo		127.0.0.1	32	16

Figure 2.10: Wireshark showing activity on all interfaces

__handle__ConnectionUp

Handle connectionUp is called every time a switch is connecting to the controller. It takes in an event object that contains a connection object. The connection object is an association for the controller to communicate with the switch. I use this function to handle the individual OpenFlow configurations on each switch. All proactive OpenFlowrules should be handled within __handle__ConnectionUp.

__handle__PacketIn

__handle__packetIn is triggered when the switch receives an OpenFlow `packet_in` message. This happens when the packet does not match any OpenFlow rules on the switch. This way the controller can work in a reactive way by creating new rules to network changes. It is also a useful way to check whether the OpenFlow rules were successfully applied on the switch. If an OpenFlow rule was not applied in a switch. That switch will send the packet to the controller. In my setup, the controller prints out the package and which switch it was sent from, for debugging purposes. This gives a good picture of where the network is failing to forward the packet. If you want to create a controller that installs the OpenFlow rules reactively this should be handled _handle_packetIn

2.13 Top

Top is a Linux and Unix command that displays the running processes, and the processing power being managed by the kernel in real-time. Top can be run solarix mode and irix mode. Solarix mode gives the CPU usage divided by the amount of CPU cores. In irix mode it will not divide the CPU usage on the total amount of CPUs. Thus, when saying that it uses 100% CPU on the Raspberry Pi that has a quad-core processor, in irix mode, this means 25% of the total available CPU [top].

2.14 xfrm

Xfrm is an IP framework for the Linux kernel that handles the encryption of packets. It is used to implement IPsec. It handles the SAD and SPD. In this thesis, xfrm is used to set the security policy to stop unencrypted traffic from leaving a host.

ksoftirqd/0

ksoftirq (kernel software interrupt request) is a process that was observed using a lot of CPU power when IPsec was used with high traffic throughput. Handling of interrupt requests is outside the scope of this thesis. What is important to know is that ksoftirqd indicates that the machine is under a heavy workload. 'ksoftirqd is a per-cpu kernel thread that runs when the machine is under heavy soft-interrupt load. Soft interrupts are normally serviced on return from a hard interrupt, but it's possible for soft interrupts to be triggered more quickly than they can be serviced. If a soft interrupt is triggered for a second time while soft interrupts are being handled, the ksoftirq daemon is triggered to handle the soft interrupts in process context. If ksoftirqd is taking more than a tiny percentage of CPU time, this indicates the machine is under heavy soft interrupt load'[ksob].

'An interrupt is simply a signal that the hardware can send when it wants the processor's attention' [ksoa].

Softirq handles processing that is almost as important as hardware interrupts. The ksoftirqd is triggered when the load is too high and it is necessary to take time from user processes.

2.15 Network namespace

To be able to run traffic on a host through OVS network namespacing was used. From the Linux manual: ' A network namespace is logically another copy of the network stack, with its own routes, firewall rules, and network devices. By default, a process inherits its network namespace from its parent. Initially, all the processes share the same default network namespace from the init process' [net]. What this means is that you can run processes in an isolated virtual network within one host. If you create a namespace and type `ifconfig` you will not see any interfaces. You can then create a tap, give it an IP address and add it to the namespace. If you then type `ifconfig` you would see `tap` appear. All processes executed in that namespace will then only be able to see and interact with `tap`.

2.16 Related Work

The security research in SDN is mostly concerned with the management of the OpenFlow rules. Regarding the OpenFlow rules, there are two approaches. The first, focuses on how the added network control can enhance security. One example is using unpredictable IP mutations to thwart scanning of host IPs [JASD12]. Another, is creating a framework that provides security monitoring for cloud networks based on SDN [SG12]. The second approach is that the added control opens up for new vulnerabilities [KKS13], and look at ways to prevent these. An example is Fortnox [PSY$^+$12].

One of the reasons for choosing my problem description was that there is very little research that look at datapath encryption in SDN. The security aspects regarding encryption in SDN are mostly focusing on the communication between the switch and the controller [Sam15]. This is not the focus of this thesis. However, in [GK14] by Prasad Gorja and Rakesh Kurapati, a methodology is proposed to extend the OVS functionality to L4-L7 services to OpenFlow. Looking at ways to modify the OVS source code to extend the functionality to support load balancing, firewall, and IPsec. This is a step towards enabling OpenFlow datapath encryption.

Chapter 3 Lab

3.1 Setup

To experiment with SDN I needed a testbed. As discussed in the introduction I started out with Mininet but discovered that it did not provide the functionality that I needed. As mentioned earlier, the problem with Mininet is that all the virtual switches are on the same loopback interface. Therefore, they can not communicate through GRE tunnels, additionally, when adding a GRE tunnel, it does not show up in Mininet. Thus, after some testing, I realized, that Mininet in itself was not sufficient.

I used Virtualbox to set up a virtual network. See Figure 3.1. Each of the squares with bold lining represents a VM. The connecting tubes between the machines represent the network connection between them. The network connection was created using Virtualbox' 'Internal Network' option. The double arrowed red line represents the connection between the VMs and the host computer running Virtualbox, using the Virtualbox 'Host-only Adapter' option. Additional information about the VMs used can be seen in 3.1, and more about Virtualbox networking can be found at [vbo]. The SDN controller was running on the host machine on port 3365.

Creating an SDN-enabled switch

Installing OVS with IPsec support on a supported Linux device can be achieved in three lines. All you need to do is to run these commands:

```
# This installs the open vSwtich
sudo apt-get install openvswitch-switch

# Enables the vSwitch to use IPsec
sudo apt-get install openvswitch-ipsec
```

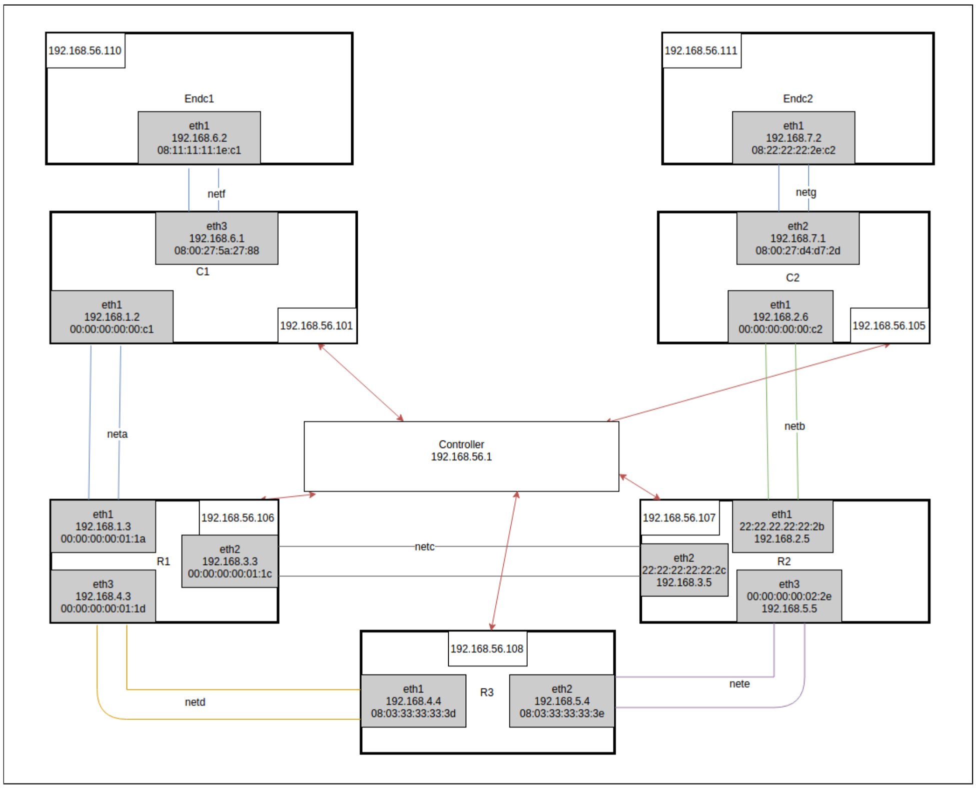


Figure 3.1: Lab setup. The inner gray boxes display the interfaces connected to the network; the name, IP address, and mac address. A fullsize figure can also be found at https://github.com/Steffb/MstThesis/blob/master/Lab.png

Name	OS	Software	RAM	CPU	OVS-version
Host	Ubuntu 14.04.1 - 64 bit	Pox	11951 MB	8	–
R1	Ubuntu 15.10 - 64 bit	Open vSwitch	1024 MB	1	2.3.1
R2	Ubuntu 15.10 - 64 bit	Open vSwitch	1024 MB	1	2.3.1
Endc1	Ubuntu 15.10 - 64 bit	Open vSwitch	1024 MB	1	2.3.1
Endc2	Ubuntu 15.10 - 64 bit	Open vSwitch	1024 MB	1	2.3.1
C1	Ubuntu 14.04 - 64 bit	Mininet, Open vSwitch	1024 MB	1	2.0.2
C2	Ubuntu 14.04 - 64 bit	Mininet, Open vSwitch	1024 MB	1	2.0.2

Table 3.1: Specifications of the VMs that were used. C1 and C2 are Mininet VMs downloaded from [minb]. The reason for setting up this many VMs was to test if it was possible to use IPsec encapsulation on packets that were already encapsulated using IPsec.

```
# IPsec Internet Key exchange daemon
sudo apt-get install racoon
```

Other dependencies that are already installed in Ubuntu can be found here [ipsc].

3.2 Discoveries

The first part of this thesis consisted of experimenting with OVS in the virtual environment. Mainly, figuring out how it worked, and discovering what possibilities there were for adding dynamic encryption. Additionally, some issues with OVS were found.

3.2.1 DPID

In OpenFlow the Datapath Identifier (DPID) is supposed to be a unique number that is used to identify each switch in the controller. However, to quote the Stanford Pox Wiki :

"A DPID is 64 bits. The spec claims the lower 48 bits are intended to be the switch's Ethernet address. ... But in implementations, this is not always true because the OpenFlow control channel often may originate from one of several interfaces and it'll use whatever Ethernet address goes with it. In practice, its pretty arbitrary, and often user-configurable independent of any Ethernet address. It's probably a decent idea to always just treat it as an opaque value which should be unique. If a vendor happens to base it on some particular Ethernet address, treat that as an implementation detail for how they achieve uniqueness and not as there being any sort of real relationship between the two. To give a more concrete answer: with Open vSwitch, it defaults to the Ethernet address of the switch's "local" port with the top 16 bits zeroed (this Ethernet address being generated at random when last I checked) [pox]".

I experienced that some of the OVS' alternated between two DPIDs, depending on what interfaces they had connected to them. I suspect this is because of the switching between physical and virtual interfaces. As mentioned earlier, the OVS process takes over the control of the physical interface when the interface is added to OVS. What I think is happening is that it then uses the MAC the new interface, to generate the DPID. Therefore, the IP and MAC, which is also sent in the first packet from the switch to the controller when it connects, should be used to identify the switch.

3.2.2 Dynamic tunnels

When setting up, and taking down virtual tunnels, the interfaces are given OpenFlow port numbers. These port numbers are only used as reference numbers within OpenFlow when adding/modifying OpenFlow rules and has nothing to do with the ports belonging to the IP address. Adding/removing ports will result in a new port number each time, to ensure that a new port will not interfere with old OpenFlow rules. However, this can be confusing when adding and removing IPsec tunnels. That is why I made:

```
def get_portnr_from_name(connection.features.ports, 'eth3'):
 for port in port_list:
   if(in_port == port.name ):
     return port.port_no
```

`Connection` is the object that is the controllers reference to the switch. This way I could give logical names to the port, and reference them by name. The function can be found in the Appendix.

Additionally, I discovered that the OpenFlow rules were not removed when the corresponding out-port was removed. This is something one should be aware of when adding and removing ports. For instance, if a port is set to forward all traffic from MAC address xx, then later that port is deleted, and another port is added that is set to forward all traffic. The result would be that all packets from xx would be dropped because OpenFlow selects the most precise match to decide the next action. Therefore, if removing a port from the switch, the OpenFlow rules that have that port as the output should also be removed. `ovs-vsctl del-flows <portname>`, does this.

3.2.3 Secure router

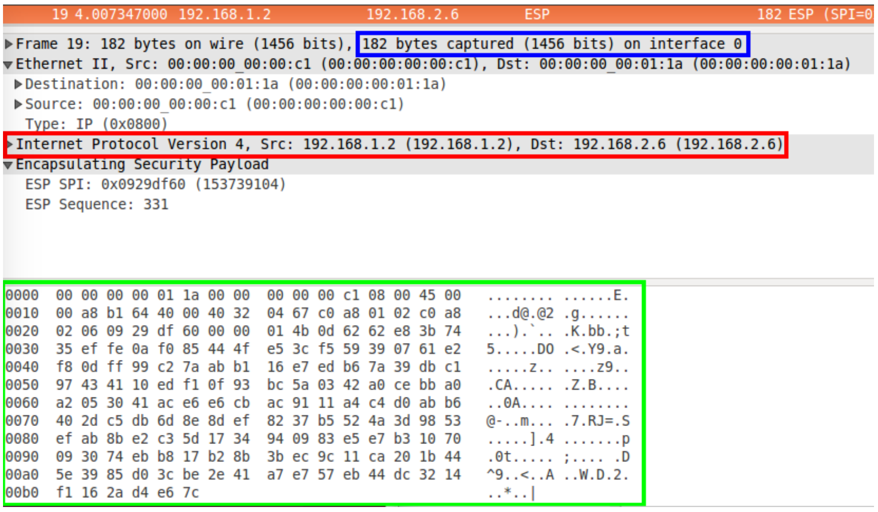
One of the interesting things to test out was whether or not it is possible for an SDN switch to forward traffic already encrypted by another SDN switch. This is interesting because it would allow you to use a switch in your SDN without worrying about it leaking any data in the case that it would be compromised.

As seen in Figure 3.2 the whole packet is encrypted, including VLAN tag and bridge key. What is not encrypted is the source and destination MAC and IP. I wanted to test if it still would be possible to control the traffic flow based on the MAC and IP address. Then test if it would be possible to modify those fields and still transport the packet successfully.

What I did was creating an IPsec tunnel from R1 to endc1 through C1 in Figure 3.1. Secondly, I added eth1 and eth3 on C1 to the OVS. Note that OVS on

C1 does not decrypt the traffic in the tunnel, it only manages traffic between eth1 and eth3. Thus C1 only sees an encrypted packet from R1 to endc1.

Initially, 'C1' had no OpenFlow rules applied, so the packets were stopping at 'C1'. Then, I applied a rule to forward traffic with an unused mac address. Finally, I added the OpenFlow rule to forward traffic with source MAC of 'endc1', resulting in the packets being successfully forwarded. The same procedure was done with the IP address of 'endc1'.



```
   19 4.007347000 192.168.1.2          192.168.2.6       ESP                        182 ESP (SPI=0
▶Frame 19: 182 bytes on wire (1456 bits), 182 bytes captured (1456 bits) on interface 0
▼Ethernet II, Src: 00:00:00_00:00:c1 (00:00:00:00:00:c1), Dst: 00:00:00_00:01:1a (00:00:00:00:01:1a)
 ▶Destination: 00:00:00_00:01:1a (00:00:00:00:01:1a)
 ▶Source: 00:00:00_00:00:c1 (00:00:00:00:00:c1)
  Type: IP (0x0800)
▶Internet Protocol Version 4, Src: 192.168.1.2 (192.168.1.2), Dst: 192.168.2.6 (192.168.2.6)
▼Encapsulating Security Payload
  ESP SPI: 0x0929df60 (153739104)
  ESP Sequence: 331

0000  00 00 00 00 01 1a 00 00  00 00 00 c1 08 00 45 00   ........ ......E.
0010  00 a8 b1 64 40 00 40 32  04 67 c0 a8 01 02 c0 a8   ...d@.@2 .g......
0020  02 06 09 29 df 60 00 00  01 4b 0d 62 62 e8 3b 74   ...).`.. .K.bb.;t
0030  35 ef fe 0a f0 85 44 4f  e5 3c f5 59 39 07 61 e2   5.....DO .<.Y9.a.
0040  f8 0d ff 99 c2 7a ab b1  16 e7 ed b6 7a 39 db c1   .....z.. ....z9..
0050  97 43 41 10 ed f1 0f 93  bc 5a 03 42 a0 ce bb a0   .CA..... .Z.B....
0060  a2 05 30 41 ac e6 e6 cb  ac 91 11 a4 c4 d0 ab b6   ..0A.... ........
0070  40 2d c5 db 6d 8e 8d ef  82 37 b5 52 4a 3d 98 53   @-..m... .7.RJ=.S
0080  ef ab 8b e2 c3 5d 17 34  94 09 83 e5 e7 b3 10 70   .....].4 .......p
0090  09 30 74 eb b8 17 b2 8b  3b ec 9c 11 ca 20 1b 44   .0t..... ;.... .D
00a0  5e 39 85 d0 3c be 2e 41  a7 e7 57 eb 44 dc 32 14   ^9..<..A ..W.D.2.
00b0  f1 16 2a d4 e6 7c                                  ..*..|
```

Figure 3.2: IPsec packet captured between C1 and C2

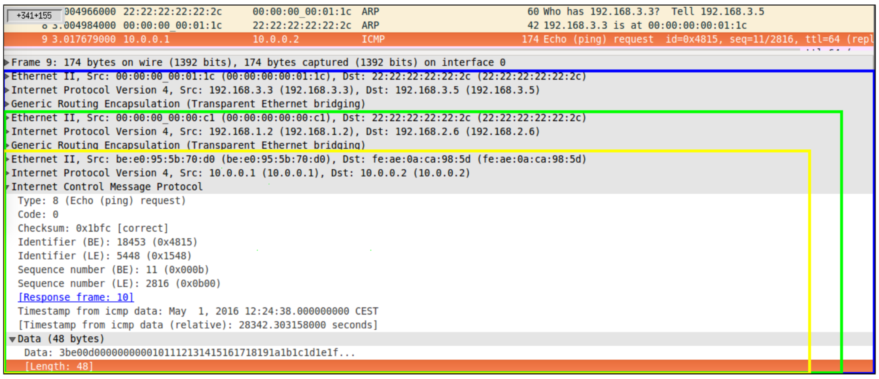
3.2.4 GRE in GRE

An interesting concept when it comes to securing the data traffic is IP-in-IP, which means encapsulating an IP packet in another IP packet. This is a concept used in GRE tunneling. What separates GRE from IP-in-IP is that GRE adds a GRE header. What I wanted to test was the possibility of encapsulating an already GRE encapsulated packet. Successfully doing this would allow for multiple layers of encryption when using the ipsec_gre option. Allowing network separation that is not just separated by VLAN tags, but also by encryption. In Figure 3.1 C1 does not need to worry about whether R1 is encrypting the traffic, and R1 does not need to worry about whether C1 encrypts its traffic when forwarding it.

GRE

In this experiment, I used the same setup as shown in Figure 3.1. The tunnels that were set up were set up from C1 to C2 and from R1 to R2. Then I created a host h1 on C1 with IP 10.0.0.1, and a host h2 on C2 with IP 10.0.0.2, using Mininet.

Then I tried to ping h2 from h1. This experiment was successful. The flow dumps showed that the traffic was going through R1 and R2. Figure 3.3 shows a packet captured from between R1 and R2: The ICMP packet packed(Yellow), in a GRE packet between C1 and C2 (green), and again encapsulated in another GRE packet between R1 and R2 (blue).



```
+341+155 004966000 22:22:22:22:22:2c      00:00:00_00:01:1c  ARP                     60 Who has 192.168.3.3?  Tell 192.168.3.5
       8 3.004984000 00:00:00_00:01:1c      22:22:22:22:22:2c  ARP                     42 192.168.3.3 is at 00:00:00:00:01:1c
       9 3.017679000 10.0.0.1               10.0.0.2           ICMP                   174 Echo (ping) request  id=0x4815, seq=11/2816, ttl=64 (repl

▶Frame 9: 174 bytes on wire (1392 bits), 174 bytes captured (1392 bits) on interface 0
▶Ethernet II, Src: 00:00:00_00:01:1c (00:00:00:00:01:1c), Dst: 22:22:22:22:22:2c (22:22:22:22:22:2c)
▶Internet Protocol Version 4, Src: 192.168.3.3 (192.168.3.3), Dst: 192.168.3.5 (192.168.3.5)
▶Generic Routing Encapsulation (Transparent Ethernet bridging)
▶Ethernet II, Src: 00:00:00_00:00:c1 (00:00:00:00:00:c1), Dst: 22:22:22:22:22:2c (22:22:22:22:22:2c)
▶Internet Protocol Version 4, Src: 192.168.1.2 (192.168.1.2), Dst: 192.168.2.6 (192.168.2.6)
▶Generic Routing Encapsulation (Transparent Ethernet bridging)
▶Ethernet II, Src: be:e0:95:5b:70:d0 (be:e0:95:5b:70:d0), Dst: fe:ae:0a:ca:98:5d (fe:ae:0a:ca:98:5d)
▶Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.2 (10.0.0.2)
▼Internet Control Message Protocol
  Type: 8 (Echo (ping) request)
  Code: 0
  Checksum: 0x1bfc [correct]
  Identifier (BE): 18453 (0x4815)
  Identifier (LE): 5448 (0x1548)
  Sequence number (BE): 11 (0x000b)
  Sequence number (LE): 2816 (0x0b00)
  [Response frame: 10]
  Timestamp from icmp data: May  1, 2016 12:24:38.000000000 CEST
  [Timestamp from icmp data (relative): 28342.303158000 seconds]
 ▼Data (48 bytes)
   Data: 3be00d000000000001011121314151617181191a1b1c1d1e1f...
   [Length: 48]
```

Figure 3.3: Captured GRE encapsulated packet within a GRE encapsulated packet

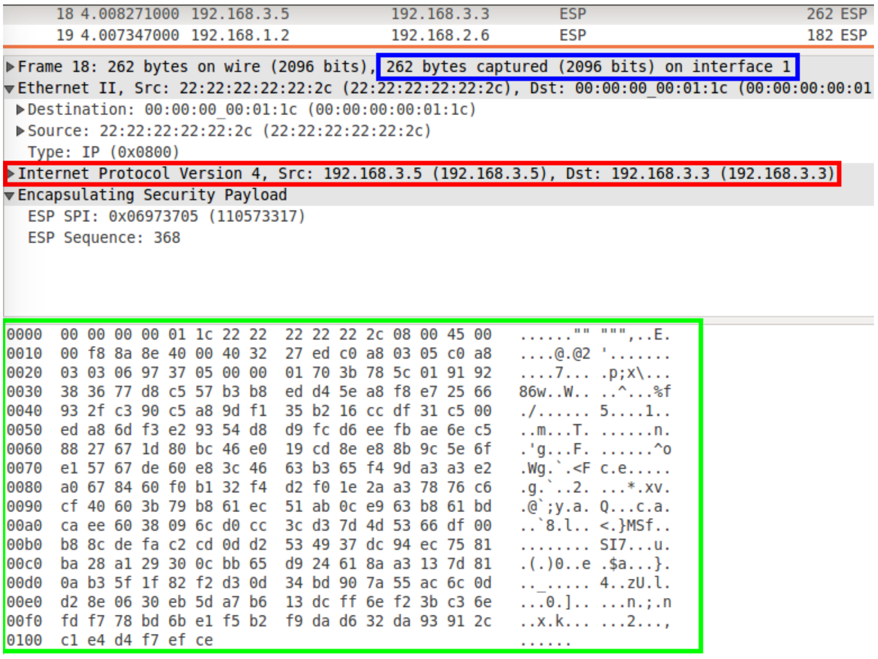
IPsec

Knowing that the principle worked, I went on to try this with IPsec tunnels. This also worked. What was a little confusing was the time it took to establish a connection. The Internet Security Association and Key Management Protocol (ISAKMP) between two endpoints does not start until there is traffic passing through, resulting in a delay before the data will be passed on. When using IPsec in IPsec this has to be done twice before the communication starts. This resulted in up to 15 seconds delay before traffic started flowing. To verify that the traffic was encapsulated twice I used Wireshark and `ovs-ofctl`. With `ovs-ofctl` I could verify that the IPsec_GRE port was used. With Wireshark, I could verify that the packets were encrypted. Figure 3.2 shows the packet that was captured between C1 and R1. Note that in the blue square you can see that the IPsec packet is larger than the double GRE encapsulated packet in Figure 3.3. All packets shown are ping request and replies.

Figure 3.4 shows the packet captured between R1 and R2. The only way to see that this is encapsulated twice it by looking at the size in the blue square. You can also see that it is the Encapsulating Security Protocol (ESP) method of IPsec that is used.

ARP Connection

The challenge with getting the double encapsulation to work. Was that in C1 the output port can not be controlled by OpenFlow because the output port needs to



```
     18 4.008271000 192.168.3.5          192.168.3.3        ESP                        262 ESP
     19 4.007347000 192.168.1.2          192.168.2.6        ESP                        182 ESP

▶Frame 18: 262 bytes on wire (2096 bits), 262 bytes captured (2096 bits) on interface 1
▼Ethernet II, Src: 22:22:22:22:22:2c (22:22:22:22:22:2c), Dst: 00:00:00_00:01:1c (00:00:00:00:01
  ▶Destination: 00:00:00_00:01:1c (00:00:00:00:01:1c)
  ▶Source: 22:22:22:22:22:2c (22:22:22:22:22:2c)
   Type: IP (0x0800)
▶Internet Protocol Version 4, Src: 192.168.3.5 (192.168.3.5), Dst: 192.168.3.3 (192.168.3.3)
▼Encapsulating Security Payload
   ESP SPI: 0x06973705 (110573317)
   ESP Sequence: 368

0000  00 00 00 00 01 1c 22 22  22 22 22 2c 08 00 45 00   ......"" """,..E.
0010  00 f8 8a 8e 40 00 40 32  27 ed c0 a8 03 05 c0 a8   ....@.@2 '.......
0020  03 03 06 97 37 05 00 00  01 70 3b 78 5c 01 91 92   ....7... .p;x\...
0030  38 36 77 d8 c5 57 b3 b8  ed d4 5e a8 f8 e7 25 66   86w..W.. ..^...%f
0040  93 2f c3 90 c5 a8 9d f1  35 b2 16 cc df 31 c5 00   ./...... 5....1..
0050  ed a8 6d f3 e2 93 54 d8  d9 fc d6 ee fb ae 6e c5   ..m...T. ......n.
0060  88 27 67 1d 80 bc 46 e0  19 cd 8e e8 8b 9c 5e 6f   .'g...F. ......^o
0070  e1 57 67 de 60 e8 3c 46  63 b3 65 f4 9d a3 a3 e2   .Wg.`.<F c.e.....
0080  a0 67 84 60 f0 b1 32 f4  d2 f0 1e 2a a3 78 76 c6   .g.`..2. ...*.xv.
0090  cf 40 60 3b 79 b8 61 ec  51 ab 0c e9 63 b8 61 bd   .@`;y.a. Q...c.a.
00a0  ca ee 60 38 09 6c d0 cc  3c d3 7d 4d 53 66 df 00   ..`8.l.. <.}MSf..
00b0  b8 8c de fa c2 cd 0d d2  53 49 37 dc 94 ec 75 81   ........ SI7...u.
00c0  ba 28 a1 29 30 0c bb 65  d9 24 61 8a a3 13 7d 81   .(.)0..e .$a...}.
00d0  0a b3 5f 1f 82 f2 d3 0d  34 bd 90 7a 55 ac 6c 0d   .._..... 4..zU.l.
00e0  d2 8e 06 30 eb 5d a7 b6  13 dc ff 6e f2 3b c3 6e   ...0.].. ...n.;.n
00f0  fd f7 78 bd 6b e1 f5 b2  f9 da d6 32 da 93 91 2c   ..x.k... ...2...,
0100  c1 e4 d4 f7 ef ce                                  ......
```

Figure 3.4: IPsec packet captured between R1 and R2

be the virtual IPsec/GRE port, and that has to be sent over the hosts IP stack, resulting in c1 sending out an ARP broadcast for `192.168.2.6`, the endpoint of the tunnel. This would be received on the eth1 port of R1. However, if it is setup as a simple forwarding port, the ARP broadcast will be sent to the second GRE tunnel out-port. It turns out that the GRE tunnels will not encapsulate nor decapsulate packets unless it knows the next step MAC address, and so the packet will be lost. This is why the routing and ARP entries need to be added proactively. In this case, C1 would need C2 in its routing table, and the MAC address of as the next hop. Therefore, the ARP entries have to be added proactively for double encapsulation to work.

MTU

MTU became a problem when running a test with ping, and is something that should be considered when using encapsulation protocols. As can be seen in the blue squares in figure Figure 3.2 and Figure 3.4, encapsulating a packet with IPsec_GRE adds 640 bits(80 bytes), and encapsulating a packet in GRE adds 304 bits (38 bytes) as can be seen in Figure 3.3. It became a problem when running ping testing with packets size set to 1400 bytes, and when running Iperf. It worked when having GRE tunnels between each switch. However, it did not work when encapsulating a GRE tunnel in another GRE tunnel. The maximum packet size that worked with a double GRE tunnel was 1396 bytes. When using one GRE tunnel at a time, the max packet size was 1434 bytes. Added packet size needs to be considered when using encapsulation. Especially, if using encapsulation multiple times. The end-nodes need to consider this when setting the Maximum Segment Size (MSS) size.

'The MSS defines the maximum amount of data that a host is willing to accept in a single TCP/IP datagram. The MSS value is sent as a TCP header option only in TCP SYN segments. Each side of a TCP connection reports its MSS value to the other side. The sending host is required to limit the size of data in a single TCP segment to a value less than or equal to the MSS reported by the receiving host' [cis]

3.2.5 Verifying changes

During these tests I have used Wireshark and manual packet inspection to verify that the packets being sent are encrypted. However, in a bigger network manual packet inspection would be inefficient. The 'paramiko.SSHClient().exec_command('command') raises SSHException: if the server fails to execute the command' [para], which can be handeld at the controller. Additionally, with OpenFlow the controller can query the switch to see what OpenFlow configurations that are in use [ofsa].

3.2.6 Using keys

When setting up multiple tunnels, it is important to use keys. Not the PSK but the key used for separating tunnels. The reason this is important is that if OVS has two tunnel endpoints that match the incoming packet, OVS will just drop the packet. This is also true even though one port is using IPsec and another is using GRE. If they are not separated by a key, or IP address, the packets will be dropped. The way to set a key to a port can be seen below. In this case, the tunnel key is 4. This will require that the key on both ends of the tunnel is set to 4. It is also possible to distinct keys by using a remote and local key. `ovs-vsctl add-port bridge2 ipsec_port - set interface ipsec_port type=ipsec_gre options:remote_ip=129.241.205.110 options:psk=password123 options:key=4`

3.2.7 Port Name

Apparently the port name can only be 15 characters long, which should be considered when choosing a naming convention. A problem that occurred was a miss-match when looking up port names. Using the port name `ipsec_gre-rasp1-r1` shows up as `ipsec_gre-r2-ra` in OpenFlow. Furthermore, if you want to delete or modify the port you have to reference it by its full name. Therefore, port names should not be longer than 15 characters.

3.2.8 Racoon

You need to run `sudo racoon` to make IPsec work with OVS. As far as I have seen, this is not documented anywhere. The command must be run after every start up. If not, the IPsec_GRE tunnels will not work, without giving any reason. The way I discovered it was by looking at the system log and seeing that the ISAKMP did not complete. I ran `sudo racoon` to check what options I had and discovered that IPsec connection started working.

3.2.9 Securing traffic to the Host

OVS is available for Ubuntu, Debian, and Fedora [ovsa]. If the host also supports namespacing, it is possible to remotely control that traffic is secured all the way to the host machine. By only allowing processes that require network connectivity to run withing the namespace, it is possible to force all the traffic to go through OVS while keeping it transparent to the user. This is illustrated in the setup I used here Figure 4.1. The processes only see the tap1 interface, so changes done in the `ovs-bridge` will not be visible. This could be useful in a office wireless network. One could use this to secure the wireless connection to the Access Point (AP) without having to give out a PSK to persons. It could also be certificate based, and the PSK could be changed remotely without user interaction.

Chapter 4 Performance

After doing configuration testing and discovering what was possible, I went on to test the performance. Latency and throughput is interesting to test because there is no point in having a secure network if it degrades the performance to an unacceptable level. Latency and throughput was tested. To quickly freshen up the difference between them: latency is the time it takes for a bit to travel from a to b, and throughput is how much data that can be sent from a to b per time. While running these experiments I also used Top, to monitor the CPU usage. I used the same setup as in Figure 3.1. In all performance results shown here the ISAKMP was already negotiated before running the tests.

4.1 Latency

The latency testing was done with the command: `sudo ping 192.168.7.2 -s 1300 -c 10000 -f`. `-s 1300` sets the packet size to 1300 byte. As discussed earlier, this is close to the maxium packetsize that can be encapsulated in IPsec twice without fragmentation. `-c 1000` means sending 1000 packets, and `-f` means flood. About flood : *'... If interval is not given, it sets interval to zero and outputs packets as fast as they come back or one hundred times per second, whichever is more. Only the super-user may use this option with zero interval'*[pin]. The results can be seen in Table 4.1. All the values are given in Milliseconds.

As we can see from Table 4.1, IPsec does introduce some latency and has a greater deviation. Also interesting is that whether the GRE tunnel is encapsulating once or twice has little effect.

4.2 Throughput

To test the throughput I used iperf. The commands used were: `iperf -c 192.168.7.2 -t 60 -M 1300` and `iperf -s`. `-c` stands for client, `192.168.7.2` is the IP address

Setup	**min**	**avg**	**max**	**mdev**	**time**
gre-between-gre	*0.103*	*1.587*	*16.407*	*1.150*	*18879*
ipsec-between-ipsec	*0.255*	*1.743*	*21.738*	*1.412*	*20630*
greingre	*0.192*	*1.537*	*13.004*	*1.103*	*18582*
ipsecinipsec	*0.302*	*1.986*	*15.011*	*1.424*	*23754*

Table 4.1: Latency when ping flooding

Setup	**Transfer**	**Time**	**Bandwidth**
gre-between-gre	*1.51 GByte*	*60 s*	*215 Mbits/sec*
ipsec-between-ipsec	*649 MByte*	*60 s*	*90.7 Mbits/sec*
greingre	*1.58 GByte*	*60 s*	*227 Mbits/sec*
ipsecinipsec	*802 MByte*	*60 s*	*112 Mbits/sec*

Table 4.2: Throughput

of the server side running `iperf`, `-t 60` tells the client to send traffic for 60 seconds, `-M 1300` tells it to used MSS of 1300 byte. `iperf -s` means acting as the server side. 'The MSS is usually the MTU - 40 bytes for the TCP/IP header. For Ethernet, the MSS is 1460 bytes (1500 byte MTU).' [ipea]

Table 4.2 shows the results of the tests. We see that the added GRE encapsulation actually has a better throughput than just one layer of GRE encapsulation. We also see that IPsec decreases the throughput with over 50%.

CPU usage

Performing the ping test the CPU usage did not go above 2.3 %. When running the throughput test on the IPsec tunnels between C1 and C2, R1 and R2 constantly used between 20 - 30 % and was not affected by the direction of the traffic. The CPU usage was the same on `ipsec-in-ipsec` and `ipsec-between-ipsec`. Thus, whether is was doing encrypt then decrypt or just encrypt did not make any difference, indicating that the R1 and R2 machines were not the performance bottleneck.

What was interesting was that when running the iperf loadtest the CPU usage was different depending on which way the packets were sent. It changed depending on traffic direction. This happened both when using IPsec and normal GRE. It seems to require greater CPU power to do decapsulation.

4.3 Raspberry Pi

The original idea was to transform a wireless router to an SDN switch. However, there were some compatibility issues when adding the OpenFlow support on the openWRT firmware. Secondly, after discovering that OVS is compatible with Ubuntu mate (an operating system that runs on Raspberry Pi) using a Raspberry Pi became a better solution. A Raspberry Pi is cheaper than a router with equal processing power, and OpenFlow enabling it is easier than on openWRT.

4.3.1 Setup

In this part, I used the Raspberry Pi 3, Model B, 1 GB RAM. Full specification can be found here [ras]. It was running Ubuntu mate 15.10.3 32-bit version with OVS version 2.4.0. The steps to install Open vSwitch on a Raspberry Pi running Ubuntu mate are the same as shown in 3.1. The Raspberry Pi has a High-Definition Multimedia Interface (HDMI) port so it can be connected to a screen. However, when using them as a network switch it is more convenient to use SSH. To use SSH you need to install openssh-server and openssh-client by running: `sudo apt-get install openssh-server openssh-client`.

Access Point

As an AP i used a Linksys EA2700, more information can be found here [rou]. The AP used Wi-Fi Protected Access (WPA)2 with PSK `Itemize2016` to avoid other clients from connecting to the network and add noise to the results. Linksys EA2700 supports 802.11 b/g/n. I used the 2.4 MegaHertz (mHz) band, with channel selection set to automatic. The Raspberry Pi uses 802.11n. So the theoretical maximum throughput is 54 Mbits/s.

4.4 Performance

The next step was to connect my virtual environment on my host computer to the physical network with the Raspberry Pi. Using C1 and C2 as hosts, and connecting R1 and R2 as switches to my physical network. Therefore, R1 and R2 needed a bridge to the physical network. I did this by using the `bridged-adapter` in Virtualbox to bridge the wireless network interface to a virtual interface at R1 and R2. However, since the internal network is using internal IP addresses it is necessary to add an ARP and route entry, to tell it to forward the traffic through the newly added bridge. If connecting a VM in Virtualbox to a Dynamic Host Configuration Protocol (DHCP) router you also need to do some manual changes in the network configuration. In Ubuntu Linux, this is done in `/etc/network/interfaces`. The interface that is bridged to the wireless interface needs to be set to use DHCP. This is done by

Table 4.3: Latency between C1 and C2 through a Raspberry Pi

Setup	**min**	**avg**	**max**	**mdev**	**time**
gre-between-gre	*2.827*	*15.745*	*735.427*	*51.470*	*90213*
gre-between-gre	*3.587*	*19.527*	*415.197*	*36.830*	*90515*
gre-between-gre	*2.240*	*9.655*	*161.186*	*13.119*	*12.272*
ipsec-between-ipsec	*2.545*	*13.598*	*634.938*	*38.371*	*84852*
ipsec-between-ipsec	*3.130*	*13.097*	*658.111*	*13.119*	*85874*

Table 4.4: Throughput between C1 and C2 through a Raspberry Pi

Setup	**Transfer**	**Time**	**Bandwidth**
gre-between-gre	*11.9 MBytes*	*60 sec*	*1.53 Mbits/sec*
gre-between-gre	*21.1 MBytes*	*60 sec*	*2.73 Mbits/sec*
ipsec-between-ipsec	*18.4 MBytes*	*60 sec*	*2.55 Mbits/sec*
ipsec-between-ipsec	*21.5 MBytes*	*60 sec*	*2.26 Mbits/sec*

changing:
```
auto <interface name>
iface <inteface name> inet static.
```
To:
```
auto <interface name>
iface <inteface name> inet dhcp.
```
Also, the promiscuous mode in the network preferences in Virtualbox needs to be set to 'Allow all'. If not, the VM will be able to ping the Raspberry Pi but not the other way around. The results of the performance tests can be seen in Table 4.3 and Table 4.4.

MAC addresses

What threw me off a bit was that when connecting the VMs to the real network, was that the traffic had to go through the physical network card on the host machine, therefore all traffic that is going through that network card needs to set the mac of the physical wireless card as its MAC source / destination, and not the internal MAC. This would be handled automatically by the ARP protocol, but it is something you need to be aware of when adding ARP entries proactively.

Results

The reason I have added multiple entries in Table 4.4 and Table 4.3 is to illustrate that the results were very inconsistent. Looking at latency, we can see that `gre-between-gre` has a inconsistent average latency. Similarly, when we look at the throughput, the results vary a lot. There could be multiple reasons:

- A factor could be the wireless card on the host machine. Even though the traffic is generated and received on different VMs, they are on the same host machine, and travel through the same wireless card. I have not control over how Virtualbox handles the network connection to a physical port
- It could be the wireless cards on the Raspberry Pi that has unstable performance.
- The network is in infrastructure mode which means that all the traffic is going through the AP
- Finally, it could be noise from other wireless networks in the area. Even microwaves can create noise on the 2.4 gHz band and affect wireless signals.

New Setup

To eliminate factors that could limit performance besides the Raspberry Pi and wireless signal, I switched to a different setup that just sent traffic between the Raspberry Pi devices. This was done by using network namespacing and creating a namespace on each of them. Then creating an internal interface connected to that namespace. Finally, I used OVS to connect that internal interface to a GRE port that used the hosts IP stack to connect to the other Raspberry Pi. This way I could connect a 'physical' port in the namespace to the OVS and connect it to a tunnel without giving OVS the control of the IP stack. The setup is shown in Figure 4.1. The commands used to achieve this are shown in Figure 4.2

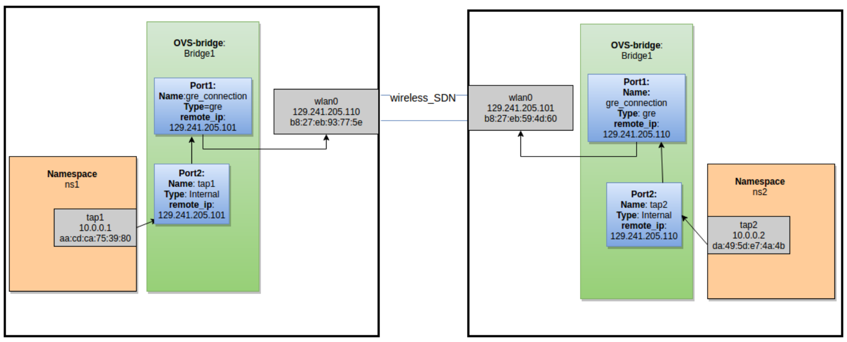
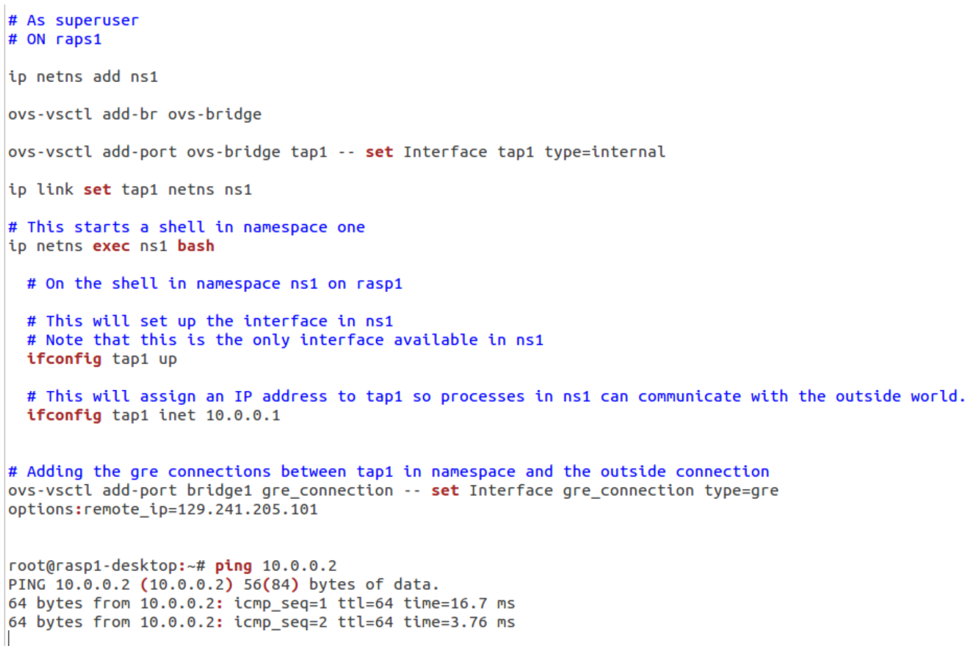

OVS-bridge: Bridge1
Port1: Name:gre_connection Type=gre remote_ip: 129.241.205.101
Port2: Name: tap1 Type: Internal remote_ip: 129.241.205.101
Namespace ns1
tap1 10.0.0.1 aa:cd:ca:75:39:80
wlan0 129.241.205.110 b8:27:eb:93:77:5e
wireless_SDN
wlan0 129.241.205.101 b8:27:eb:59:4d:60
OVS-bridge: Bridge1
Port1: Name: gre_connection Type: gre remote_ip: 129.241.205.110
Port2: Name: tap2 Type: Internal remote_ip: 129.241.205.110
Namespace ns2
tap2 10.0.0.2 da:49:5d:e7:4a:4b

Figure 4.1: Two Raspberry Pi devices communicating through an internal interface in a network namespace connected to OVS



```
# As superuser
# ON raps1

ip netns add ns1

ovs-vsctl add-br ovs-bridge

ovs-vsctl add-port ovs-bridge tap1 -- set Interface tap1 type=internal

ip link set tap1 netns ns1

# This starts a shell in namespace one
ip netns exec ns1 bash

  # On the shell in namespace ns1 on rasp1

  # This will set up the interface in ns1
  # Note that this is the only interface available in ns1
  ifconfig tap1 up

  # This will assign an IP address to tap1 so processes in ns1 can communicate with the outside world.
  ifconfig tap1 inet 10.0.0.1


# Adding the gre connections between tap1 in namespace and the outside connection
ovs-vsctl add-port bridge1 gre_connection -- set Interface gre_connection type=gre
options:remote_ip=129.241.205.101


root@rasp1-desktop:~# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=16.7 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=3.76 ms
```

Figure 4.2: Script to set up a network namespace connected to the outside through OVS

I went on to create an isolated network to remove noise from the Internet by using a router that supported Network Address Translation (NAT). The only traffic over the wireless interface on the Raspberry Pi devices was the traffic that I generated. Then the throughput and latency experiments were performed. Note that in this case the Raspberry Pi was either encrypting or decrypting. Not both as in Table 4.4.

Network noise

The Android app 'Wi-Fi Analyzer' showed that the other networks were running on most of the available channels as can be seen in Figure 4.3. I tried switching to channel three which supposedly was least used. This still resulted in a 18.9 ms standard deviation over 10000 packets when pinging between the Raspberry Pi devices, indicating that there was signal noise. Therefore, as a final attempt to get some valid results I tried to re-run the experiments during nighttime. This way there would be minimal disturbance from other wireless units running. The networks were still active, the difference was that fewer people were using them. The results are shown in Table 4.5 and Table 4.6.

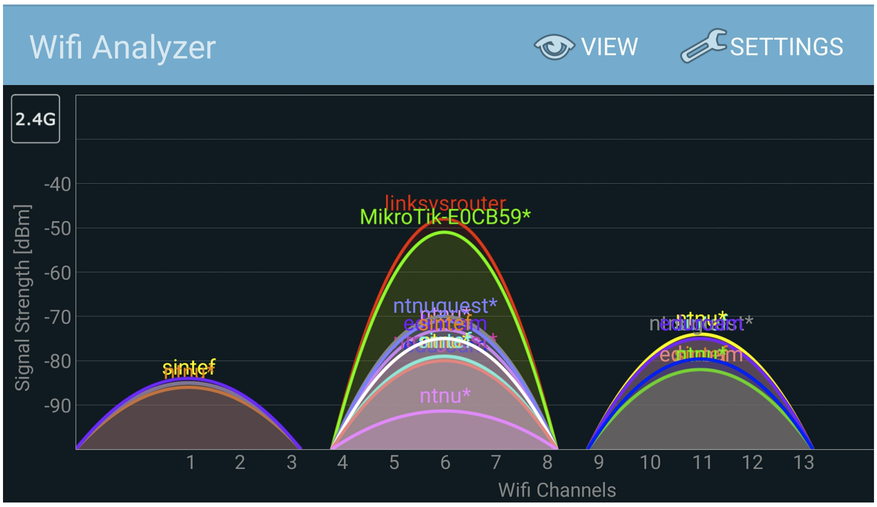


Figure 4.3: Showing the signal strength on the different channels

Setup	min	avg	max	mdev	total time	% Increase from normal
`normal`	*2.108*	*4.064*	*101.746*	*5.246*	*37889ms*	*0*
`gre-tunnel`	*3.438*	*4.810*	*42.043*	*1.927*	*34450ms*	*18.36*
`ipsec-tunnel`	*3.763*	*5.231*	*37.821*	*1.646*	*54550ms*	*28.72*

Table 4.5: The latency performmance between two Raspberry Pi devices over a wireless network. Performed late at night to reduce noise

Setup	Transfer	total time	Bandwidth	% Decrease from normal
`normal`	*146 MBytes*	*60.1 sec*	*20.5 Mbits/sec*	*0*
`gre`	*137 MBytes*	*60.0 sec*	*19.2 Mbits/sec*	*6.34*
`ipsec`	*86.8 MBytes*	*60.2 sec*	*12.1 Mbits/sec*	*40.98*

Table 4.6: The throughput performance between two Raspberry Pi devices over a wireless network. Performed late at night to reduce noise.

4.4.1 On wired network

Sending traffic over Wi-Fi is usually done on the connection end-point on the client side. When forwarding traffic over long distances or with high throughput, a wired connection is preferred, as it can provide higher speeds. Therefore, the same latency and throughput tests were also performed on a wired network. This was done by attaching the two Raspberry Pi devices directly to the AP with a Category (CAT) 5 twisted pair cable. The results can be seen in Table 4.7 and Table 4.8.

Setup	Transfer	total time	Bandwidth	% decrease from normal
`normal`	*669 MBytes*	*60.0 sec*	*93.5 Mbits/sec*	*0*
`gre`	*651 MBytes*	*60.0 sec*	*91.0 Mbits/sec*	*2.67*
`ipsec`	*601 MBytes*	*60.0 sec*	*83.9 Mbits/sec*	*10.27*
`ipsec 2 threads`	*605 MBytes*	*60.1 sec*	*84.6 Mbits/sec*	*9.52*
`ipsec 4 threads`	*606 MBytes*	*60.2 sec*	*84.2 Mbits/sec*	*9.95*

Table 4.7: Throughput wired network

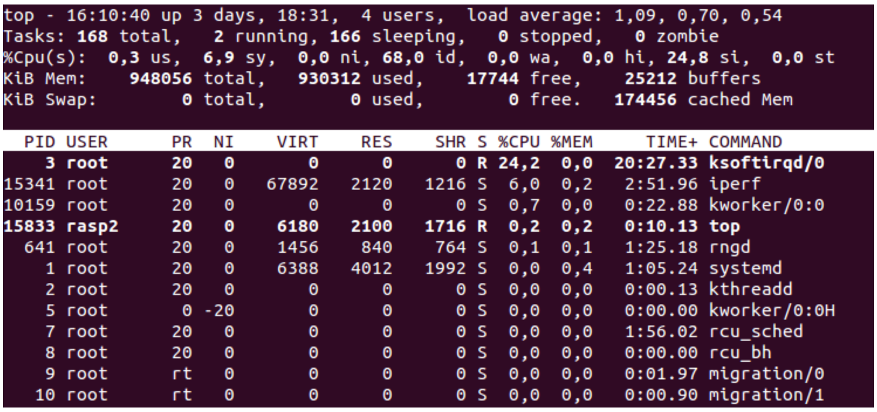
Setup	min	avg	max	mdev	total time	% increase from normal
`normal`	*0.842*	*0.860*	*1.417*	*0.044*	*9441ms*	*0*
`gre-tunnel`	*0.871*	*0.911*	*2.724*	*0.031*	*10064ms*	*5.93*
`ipsec-tunnel`	*1.252*	*1.404*	*5.220*	*0.124 ms*	*18639ms*	*63.26*

Table 4.8: Latency On wired network. Performed late with less noise.

CPU usage

What was interesting to observe was that running iperf on a wired connection resulted in iperf using 21-21 % CPU, and the other processes using less than 5%. When running iperf over the IPsec tunnel, ksoftirqd/0 on the sender side used 57% CPU and maxed out on the receiver side. This was similar to what was observed in the virtual environment. The total CPU is shown in Figure 4.5

Raspberry Pi has an Arm quad-core processor, which means that it has four CPUs. Looking at Figure 4.4 we see that switching to the Solaris mode, that only one thread is used when iperf is running with four threads, meaning that the Raspberry Pi would have a potential for more throughput with IPsec if all four cores were utilized because we saw the CPU becoming a limiting factor in IPsec.



```
top - 16:10:40 up 3 days, 18:31,  4 users,  load average: 1,09, 0,70, 0,54
Tasks: 168 total,   2 running, 166 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0,3 us,  6,9 sy,  0,0 ni, 68,0 id,  0,0 wa,  0,0 hi, 24,8 si,  0,0 st
KiB Mem:    948056 total,   930312 used,    17744 free,    25212 buffers
KiB Swap:        0 total,        0 used,        0 free.   174456 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
    3 root      20   0       0      0      0 R 24,2  0,0  20:27.33 ksoftirqd/0
15341 root      20   0   67892   2120   1216 S  6,0  0,2   2:51.96 iperf
10159 root      20   0       0      0      0 S  0,7  0,0   0:22.88 kworker/0:0
15833 rasp2     20   0    6180   2100   1716 R  0,2  0,2   0:10.13 top
  641 root      20   0    1456    840    764 S  0,1  0,1   1:25.18 rngd
    1 root      20   0    6388   4012   1992 S  0,0  0,4   1:05.24 systemd
    2 root      20   0       0      0      0 S  0,0  0,0   0:00.13 kthreadd
    5 root       0 -20       0      0      0 S  0,0  0,0   0:00.00 kworker/0:0H
    7 root      20   0       0      0      0 S  0,0  0,0   1:56.02 rcu_sched
    8 root      20   0       0      0      0 S  0,0  0,0   0:00.00 rcu_bh
    9 root      rt   0       0      0      0 S  0,0  0,0   0:01.97 migration/0
   10 root      rt   0       0      0      0 S  0,0  0,0   0:00.90 migration/1
```

Figure 4.4: Screenshot of `top` in Solaris mode. Show that Raspberry Pi is not using multithreading on the server side. The client is running iperf with four threads.

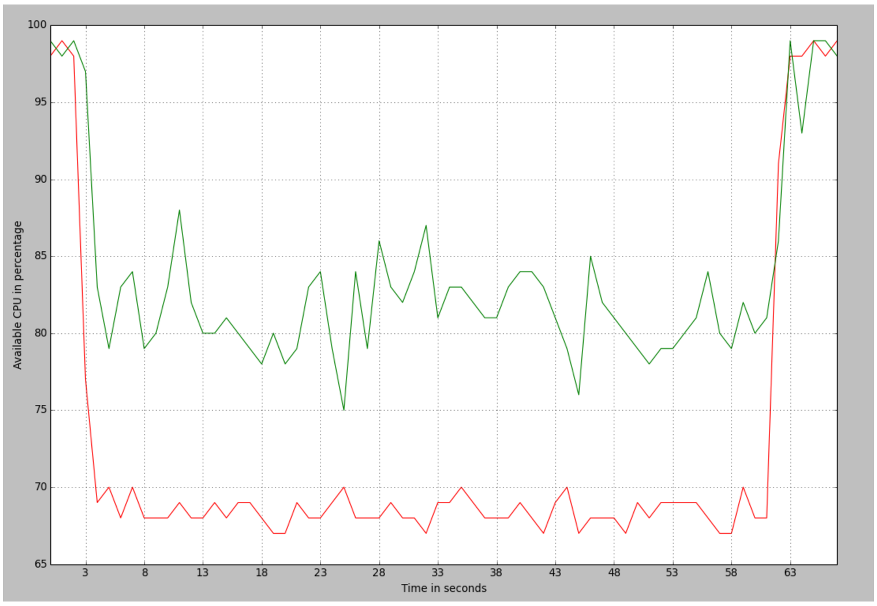


Figure 4.5: Shows the available CPU on the sender and receiver side when using IPsec. The green line is the Raspberry Pi device that sends and encrypts the traffic. The red line is the Raspberry Pi device that receives and decrypts the traffic. It illustrates that the receiver has less available CPU power.

Chapter 5
Security on Raspberry Pi

This chapter contains the security related experiments with OVS. These experiments were run on the Raspberry Pi setup shown in Figure 4.1, with the controller running on the host machine described in Table 3.1. After discovering that a Raspberry Pi devices could be a wireless SDN switch, running the experiment on them was more interesting than using Virtualbox, because it would be comparable to a physical SDN network. It is important to separate these experiments from the ones in chapter 3 because they run on a different operative system, different hardware and a different OVS version.

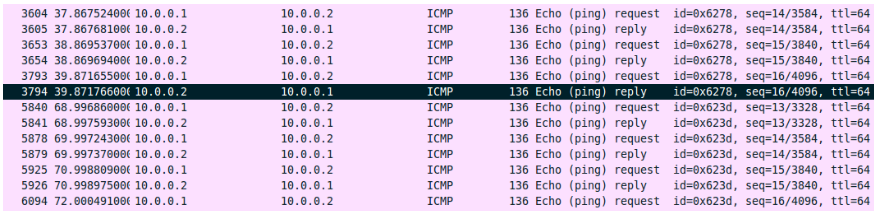
5.1 Replay attack

I wanted to verify that the IPsec encryption in OVS actually provided protection against a replay attack. A replay attack is where data is maliciously delayed or repeated. This is an attack that is possible just by eavesdropping on the traffic being sent. An example would be if somebody sent a system command over an encrypted channel. If there is no replay protection an attacker could just eavesdrop and resend the same packet and the same configuration would be applied. Typical ways to protect against this are by the use of session tokens and timestamps.

To simulate a replay attack I used `tcpdump`, `tcpreplay` and Wireshark. First, I tried it on the normal GRE tunnel. Starting with running ping between the the Raspberry Pi devices over a GRE tunnel while running `tcpdump` with: `sudo tcpdump dst 129.241.205.101 -w capture_file`. This command allowed me to capture all packets with destination IP of 129.241.205.101 (the receiver of the ping requests) and write it to a file called capture file. Then I stopped the pinging and ran:

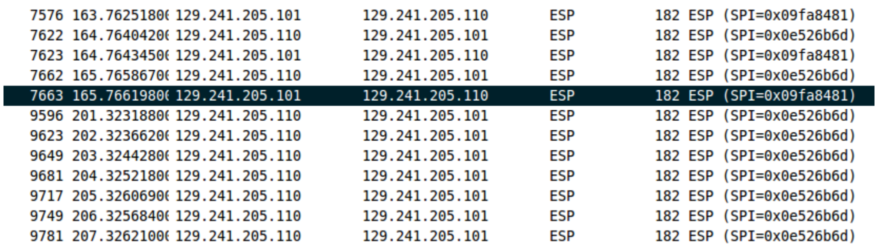
`tcpreplay -intf1=wlan0 capture_file` replays the packets in `capture_file` on interface `wlan0`. `wlan0` is the wireless interface on the Raspberry Pi.

Running Wireshark on the remote host that was pinged (129.241.205.101), I could



```
3604 37.86752400( 10.0.0.1     10.0.0.2     ICMP     136 Echo (ping) request  id=0x6278, seq=14/3584, ttl=64
3605 37.86768100( 10.0.0.2     10.0.0.1     ICMP     136 Echo (ping) reply    id=0x6278, seq=14/3584, ttl=64
3653 38.86953700( 10.0.0.1     10.0.0.2     ICMP     136 Echo (ping) request  id=0x6278, seq=15/3840, ttl=64
3654 38.86969400( 10.0.0.2     10.0.0.1     ICMP     136 Echo (ping) reply    id=0x6278, seq=15/3840, ttl=64
3793 39.87165500( 10.0.0.1     10.0.0.2     ICMP     136 Echo (ping) request  id=0x6278, seq=16/4096, ttl=64
3794 39.87176600( 10.0.0.2     10.0.0.1     ICMP     136 Echo (ping) reply    id=0x6278, seq=16/4096, ttl=64
5840 68.99686000( 10.0.0.1     10.0.0.2     ICMP     136 Echo (ping) request  id=0x623d, seq=13/3328, ttl=64
5841 68.99759300( 10.0.0.2     10.0.0.1     ICMP     136 Echo (ping) reply    id=0x623d, seq=13/3328, ttl=64
5878 69.99724300( 10.0.0.1     10.0.0.2     ICMP     136 Echo (ping) request  id=0x623d, seq=14/3584, ttl=64
5879 69.99737000( 10.0.0.2     10.0.0.1     ICMP     136 Echo (ping) reply    id=0x623d, seq=14/3584, ttl=64
5925 70.99880900( 10.0.0.1     10.0.0.2     ICMP     136 Echo (ping) request  id=0x623d, seq=15/3840, ttl=64
5926 70.99897500( 10.0.0.2     10.0.0.1     ICMP     136 Echo (ping) reply    id=0x623d, seq=15/3840, ttl=64
6094 72.00049100( 10.0.0.1     10.0.0.2     ICMP     136 Echo (ping) request  id=0x623d, seq=16/4096, ttl=64
```

Figure 5.1: Showing a replay attack on the GRE tunnel. The packets up until 3794 are ordinary packets, and the subsequent are replayed with `tcpreplay`. This can be seen by looking at the sequence number in the second column from the right



```
7576 163.7625180( 129.241.205.101   129.241.205.110   ESP   182 ESP (SPI=0x09fa8481)
7622 164.7640420( 129.241.205.110   129.241.205.101   ESP   182 ESP (SPI=0x0e526b6d)
7623 164.7643450( 129.241.205.101   129.241.205.110   ESP   182 ESP (SPI=0x09fa8481)
7662 165.7658670( 129.241.205.110   129.241.205.101   ESP   182 ESP (SPI=0x0e526b6d)
7663 165.7661980( 129.241.205.101   129.241.205.110   ESP   182 ESP (SPI=0x09fa8481)
9596 201.3231880( 129.241.205.110   129.241.205.101   ESP   182 ESP (SPI=0x0e526b6d)
9623 202.3236620( 129.241.205.110   129.241.205.101   ESP   182 ESP (SPI=0x0e526b6d)
9649 203.3244280( 129.241.205.110   129.241.205.101   ESP   182 ESP (SPI=0x0e526b6d)
9681 204.3252180( 129.241.205.110   129.241.205.101   ESP   182 ESP (SPI=0x0e526b6d)
9717 205.3260690( 129.241.205.110   129.241.205.101   ESP   182 ESP (SPI=0x0e526b6d)
9749 206.3256840( 129.241.205.110   129.241.205.101   ESP   182 ESP (SPI=0x0e526b6d)
9781 207.3262100( 129.241.205.110   129.241.205.101   ESP   182 ESP (SPI=0x0e526b6d)
```

Figure 5.2: Showing an unsuccessful replay attack on the IPsec tunnel. The packets up until 7663 are ordinary packets, and the subsequent are replayed with `tcpreplay`. 129.241.205.101 stops responding to the packets at 7663.

see that the ping requests were coming in and the host was responding to them, as can be seen in Figure 5.1. The next step was to test the IPsec tunnel. Repeating the same procedure as with the GRE tunnel. However, it did not work. The receiver did not respond to the packets being sent, as can be seen in Figure 5.2.

5.2 Change PSK

Part of this thesis was to look at how dynamic PSK distribution can be achieved. Different approaches were tried, and the results are given below.

5.2.1 Renew PSK on interface

The first approach consisted of changing the PSK option value on each of the endpoint interfaces of the tunnel. This was done by running the command: `ovs-vsctl set Interface <Port name> options:psk=<new PSK>`, where <Port name> is an existing IPsec_GRE port. This lead to some interesting results. For one, the traffic stopped, and the time it took for the communication to resume varied greatly as can be seen in Figure 5.3.

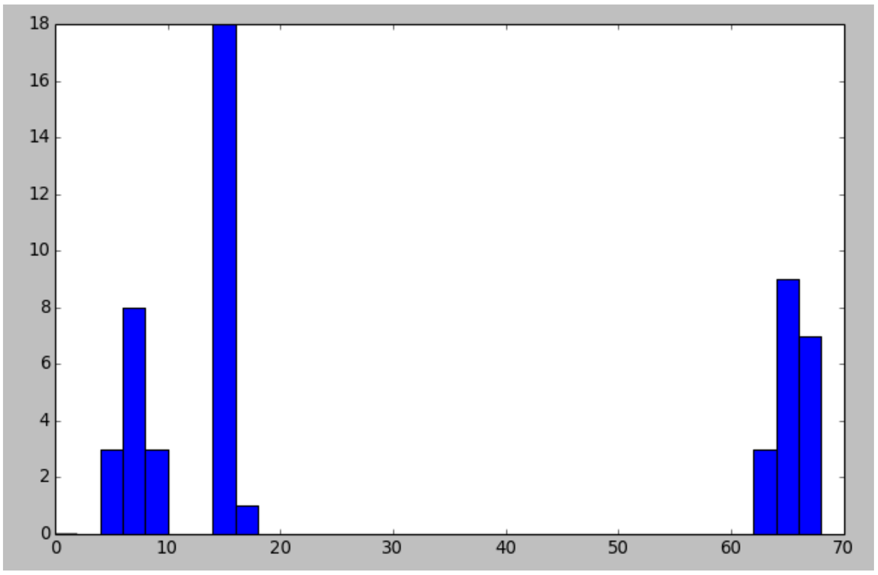
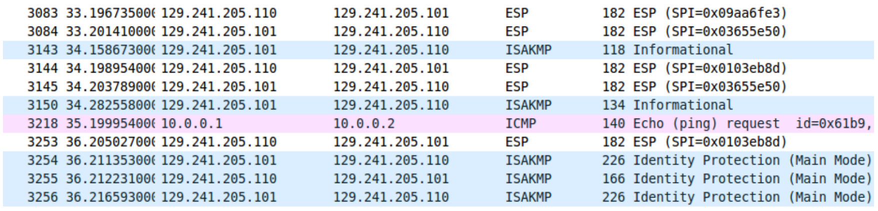


Figure 5.3: Shows time it took from changing the PSK until the traffic could resume. The x-axis is time in seconds and the y-axis is the amount of times it happened.

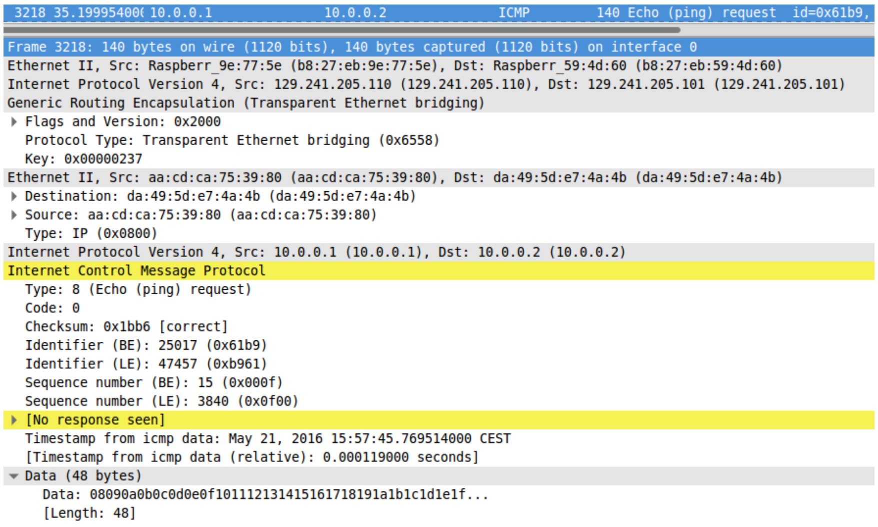


```
3083 33.19673500( 129.241.205.110      129.241.205.101      ESP        182 ESP (SPI=0x09aa6fe3)
3084 33.20141000( 129.241.205.101      129.241.205.110      ESP        182 ESP (SPI=0x03655e50)
3143 34.15867300( 129.241.205.101      129.241.205.110      ISAKMP     118 Informational
3144 34.19895400( 129.241.205.110      129.241.205.101      ESP        182 ESP (SPI=0x0103eb8d)
3145 34.20378900( 129.241.205.101      129.241.205.110      ESP        182 ESP (SPI=0x03655e50)
3150 34.28255800( 129.241.205.101      129.241.205.110      ISAKMP     134 Informational
3218 35.19995400( 10.0.0.1             10.0.0.2             ICMP       140 Echo (ping) request  id=0x61b9,
3253 36.20502700( 129.241.205.110      129.241.205.101      ESP        182 ESP (SPI=0x0103eb8d)
3254 36.21135300( 129.241.205.101      129.241.205.110      ISAKMP     226 Identity Protection (Main Mode)
3255 36.21223100( 129.241.205.110      129.241.205.101      ISAKMP     166 Identity Protection (Main Mode)
3256 36.21659300( 129.241.205.101      129.241.205.110      ISAKMP     226 Identity Protection (Main Mode)
```

Figure 5.4: A leaked ICMP packet when changing PSK

The measurement of the time it took to resume traffic flow was performed by running: `ping 10.0.0.2 -D` which gives the UNIX time of each ICMP packet. Then using the script 'change_key' to change the PSK on both sides of the tunnel. Finally, going through the ping log and registering all the time gaps greater than one second.

Even more interesting was that when changing the PSK some packets were leaked. This can be seen in Figure 5.4. During this experiment, I had Wireshark running and noticed that every time the PSKs were changed, one ICMP packet was leaking. When inspecting the packet it looks like it is just using the normal GRE option. See Figure 5.5. Even the tunnel key can be seen. This is a very serious bug and should be fixed before considering using IPsec. I am currently in a dialog with the primary author of the IPsec code in Open vSwitch. More about this in Future work.



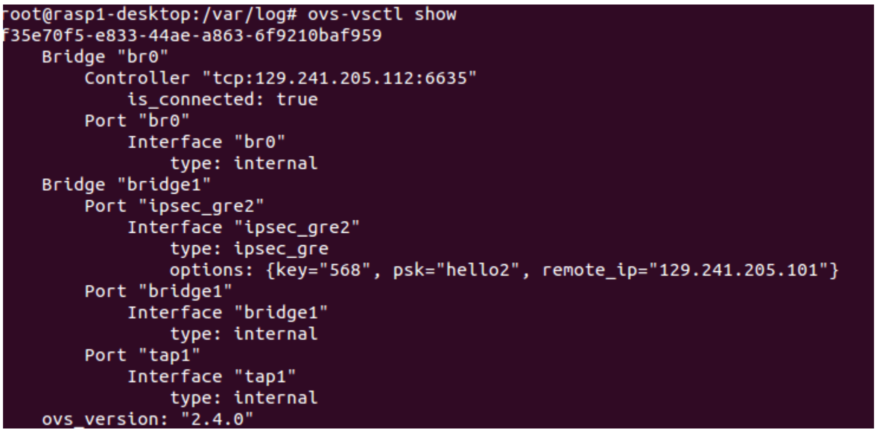
```
 3218 35.19995400( 10.0.0.1              10.0.0.2              ICMP        140 Echo (ping) request  id=0x61b9,

Frame 3218: 140 bytes on wire (1120 bits), 140 bytes captured (1120 bits) on interface 0
Ethernet II, Src: Raspberr_9e:77:5e (b8:27:eb:9e:77:5e), Dst: Raspberr_59:4d:60 (b8:27:eb:59:4d:60)
Internet Protocol Version 4, Src: 129.241.205.110 (129.241.205.110), Dst: 129.241.205.101 (129.241.205.101)
Generic Routing Encapsulation (Transparent Ethernet bridging)
 ▸ Flags and Version: 0x2000
   Protocol Type: Transparent Ethernet bridging (0x6558)
   Key: 0x00000237
Ethernet II, Src: aa:cd:ca:75:39:80 (aa:cd:ca:75:39:80), Dst: da:49:5d:e7:4a:4b (da:49:5d:e7:4a:4b)
 ▸ Destination: da:49:5d:e7:4a:4b (da:49:5d:e7:4a:4b)
 ▸ Source: aa:cd:ca:75:39:80 (aa:cd:ca:75:39:80)
   Type: IP (0x0800)
Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.2 (10.0.0.2)
Internet Control Message Protocol
   Type: 8 (Echo (ping) request)
   Code: 0
   Checksum: 0x1bb6 [correct]
   Identifier (BE): 25017 (0x61b9)
   Identifier (LE): 47457 (0xb961)
   Sequence number (BE): 15 (0x000f)
   Sequence number (LE): 3840 (0x0f00)
 ▸ [No response seen]
   Timestamp from icmp data: May 21, 2016 15:57:45.769514000 CEST
   [Timestamp from icmp data (relative): 0.000119000 seconds]
 ▾ Data (48 bytes)
     Data: 08090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f...
     [Length: 48]
```

Figure 5.5: A leaked ICMP packet. Under *Generic Routing Encapsulation* in line four from the top, the key field shows the hex value of the key used in the IPsec tunnel. This proves that the leaked packet originates from the IPsec tunnel.

5.2.2 Deleting ports

The next method to renew the PSK is by using multiple IPsec ports and adding new ports with a different PSK. This was tested by adding two IPsec ports with different PSK and using the forwarding rule NORMAL. Then manually deleting the used port with the command: `ovs-vsctl del-port <port-name>`. NORMAL makes the OVS act as a normal bridge [ofsb]. The default is to use the first port that matches the OpenFlow rule. The result was that the traffic stopped. The switch was still forwarding the traffic. However, the packets were not encrypted. It switched to use GRE without IPsec. By inspecting the packets I could see that it had the tunnel key of the second port, meaning that it had successfully changed to use the second port, yet it did not use IPsec even though that was specified. Figure 5.6 shows the information given in the console while the packets are sent out unencrypted. Some further testing showed that it also happened when using specific OpenFlow rules. What was a bigger problem was that deleting a port not in use, had the same effect on the other IPsec ports on the switch. All ports on the OVS stopped using IPsec. This is reported to the OVS security team and is being looked into. It only happens on the tunnels having the same remote IP as the port being deleted. The reason the communication stops is that the IPsec port on the receiver side only accepts traffic encrypted with the correct PSK.



```
root@rasp1-desktop:/var/log# ovs-vsctl show
f35e70f5-e833-44ae-a863-6f9210baf959
    Bridge "br0"
        Controller "tcp:129.241.205.112:6635"
            is_connected: true
        Port "br0"
            Interface "br0"
                type: internal
    Bridge "bridge1"
        Port "ipsec_gre2"
            Interface "ipsec_gre2"
                type: ipsec_gre
                options: {key="568", psk="hello2", remote_ip="129.241.205.101"}
        Port "bridge1"
            Interface "bridge1"
                type: internal
        Port "tap1"
            Interface "tap1"
                type: internal
    ovs_version: "2.4.0"
```

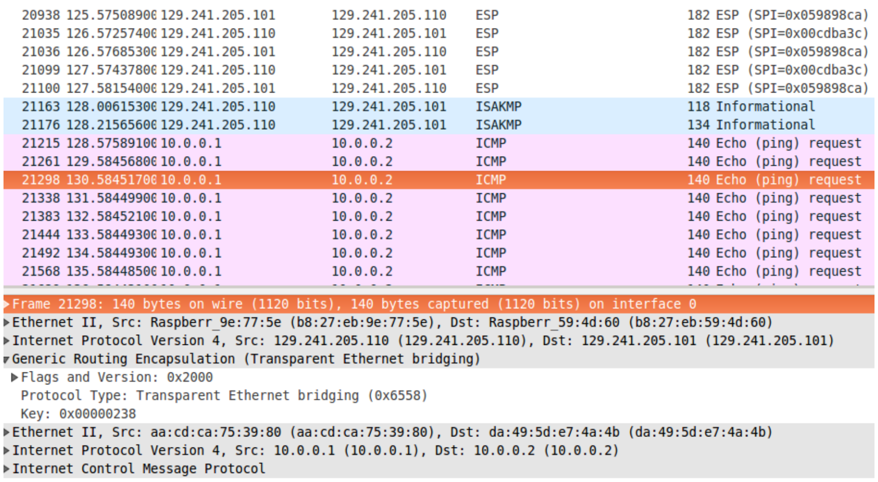
Figure 5.6: This is the information that `ovs-vsctl show` provides when OVS is sending out unencrypted packets.

Virtual environment

Observing the problems with the PSK changing in Figure 5.4 and Figure 5.7, I wanted to check if this might be a hardware problem. Thus, I repeated the experiment between endc1 and c1 in the virtual network setup seen in Figure 3.1. The result was the same as on the Raspberry Pi devices. Deleting an IPsec port resulted in the subsequent packet on the other IPsec ports not being encrypted. The whole Wireshark capture can be seen in the `Experiments/changing_key` folder on Github [git]. This is a serious problem and should definitely be further investigated. It only happens when the port is an IPsec port with the same destination IP as the port being used. I suspected that the problem could be with racoon, the keying(PSK) daemon used. However, restarting racoon by running `racoon-tool restart` did not solve the problem. This leads me to think that it has something to do with the OVS implementation of racoon.

5.2.3 PSK not changing

One time, during testing of setup time for different PSKs. The PSK renewal started failing. The traffic stopped for approximately 60 seconds to do the PSK negotiation as normal. Then subsequent PSK changes did not cause any stop in the traffic. This seemed strange compared to the earlier experiments that showed the PSK negotiation using typically either 5 or 60 seconds. Suspecting something was wrong I turned on Wireshark to see what packets were being sent. Wireshark did not show any ISAKMP packets being sent whenever the PSK changed. Suspecting that the new PSKs were not applied, I changed the PSK on one side of the tunnel to something random. The traffic was still going through the tunnel. Thus, it turns out that the



```
  20938 125.5750890 129.241.205.101      129.241.205.110    ESP                      182 ESP (SPI=0x059898ca)
  21035 126.5725740 129.241.205.110      129.241.205.101    ESP                      182 ESP (SPI=0x00cdba3c)
  21036 126.5768530 129.241.205.101      129.241.205.110    ESP                      182 ESP (SPI=0x059898ca)
  21099 127.5743780 129.241.205.110      129.241.205.101    ESP                      182 ESP (SPI=0x00cdba3c)
  21100 127.5815400 129.241.205.101      129.241.205.110    ESP                      182 ESP (SPI=0x059898ca)
  21163 128.0061530 129.241.205.110      129.241.205.101    ISAKMP                   118 Informational
  21176 128.2156560 129.241.205.110      129.241.205.101    ISAKMP                   134 Informational
  21215 128.5758910 10.0.0.1             10.0.0.2           ICMP                     140 Echo (ping) request
  21261 129.5845680 10.0.0.1             10.0.0.2           ICMP                     140 Echo (ping) request
  21298 130.5845170 10.0.0.1             10.0.0.2           ICMP                     140 Echo (ping) request
  21338 131.5844990 10.0.0.1             10.0.0.2           ICMP                     140 Echo (ping) request
  21383 132.5845210 10.0.0.1             10.0.0.2           ICMP                     140 Echo (ping) request
  21444 133.5844930 10.0.0.1             10.0.0.2           ICMP                     140 Echo (ping) request
  21492 134.5844930 10.0.0.1             10.0.0.2           ICMP                     140 Echo (ping) request
  21568 135.5844850 10.0.0.1             10.0.0.2           ICMP                     140 Echo (ping) request
▶Frame 21298: 140 bytes on wire (1120 bits), 140 bytes captured (1120 bits) on interface 0
▶Ethernet II, Src: Raspberr_9e:77:5e (b8:27:eb:9e:77:5e), Dst: Raspberr_59:4d:60 (b8:27:eb:59:4d:60)
▶Internet Protocol Version 4, Src: 129.241.205.110 (129.241.205.110), Dst: 129.241.205.101 (129.241.205.101)
▼Generic Routing Encapsulation (Transparent Ethernet bridging)
 ▶Flags and Version: 0x2000
  Protocol Type: Transparent Ethernet bridging (0x6558)
  Key: 0x00000238
▶Ethernet II, Src: aa:cd:ca:75:39:80 (aa:cd:ca:75:39:80), Dst: da:49:5d:e7:4a:4b (da:49:5d:e7:4a:4b)
▶Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.2 (10.0.0.2)
▶Internet Control Message Protocol
```

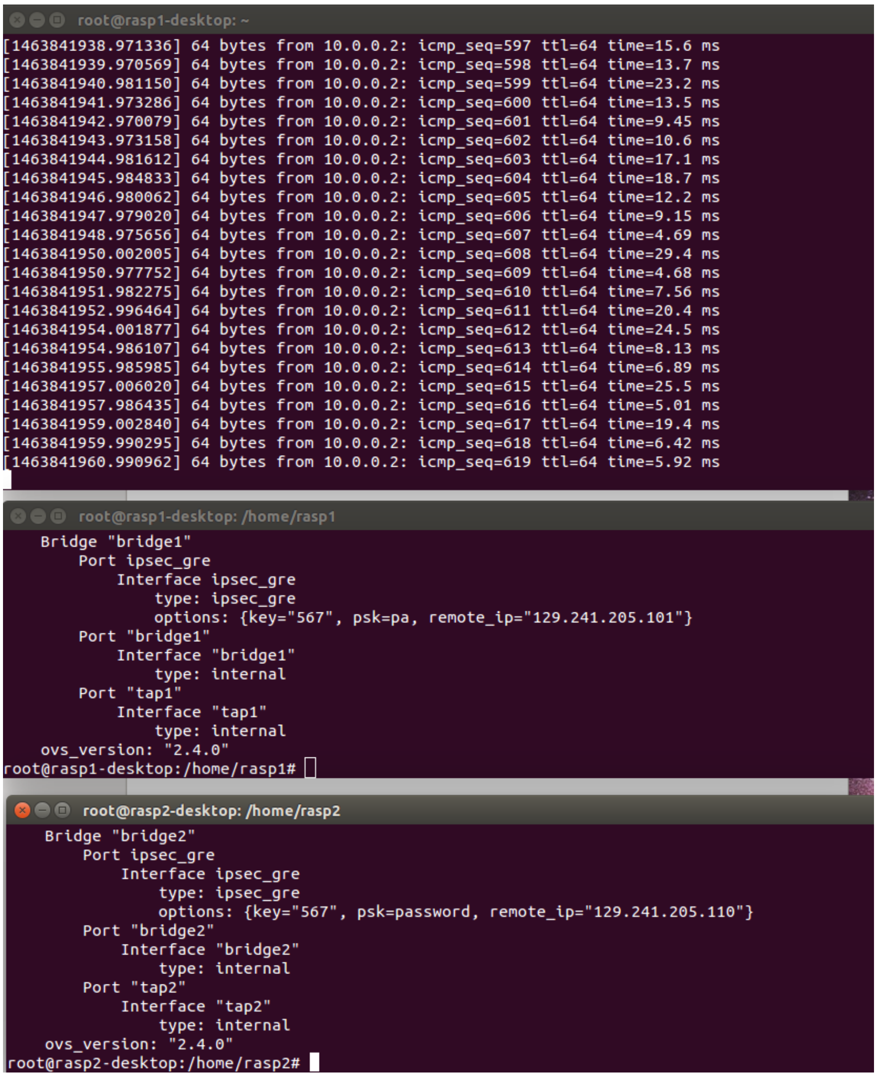
Figure 5.7: Showing the sender constantly leaking packets after the port was deleted

new IPsec PSK was not being applied. Screenshot is shown in Figure 5.8. However, every time I changed the PSK an ICMP packet still leaked out. The rest of the packets were still encrypted. I have added the log-files and packet captures capture in the `/Experiments/experiment/Raspberry/ipsectunnels/ipsec_psk_bug` folder on Github [git]. The command used to renew the PSK was `ovs-vsctl set Interface ipsec_gre options:psk=<New PSK>`

5.2.4 Adding ports

Adding ports does not seem to affect the encryption. This was tested by establishing an IPsec tunnel, and running traffic through it. Then adding another IPsec port and set OVS to use the newly added port. It worked. Flow-dumps showed traffic using the new port, and it did not have to renegotiate the ISAKMP, which is a huge advantage from a performance perspective. As we have seen, removing or editing the ports does not work well with IPsec in OVS. Thus, the best way to renew the PSK is to dynamically add new IPsec tunnels between the endpoints and use flow modification to change which port is being used for transporting traffic. This leads to two questions. How many IPsec tunnels is it possible to have, and how do multiple tunnels affect the performance.

Finding the maximum amount of available ports was done using `add_tunnel.py`, which creates a new port every 10 seconds and applies OpenFlow rules to use the newly added port. Each time it increases a counter that is used in the port-name, tunnel key, and the tunnel PSK, to ensure that all tunnels are different. A new endpoint can therefore not communicate with an old. Ping was ran while the script



```
root@rasp1-desktop: ~
[1463841938.971336] 64 bytes from 10.0.0.2: icmp_seq=597 ttl=64 time=15.6 ms
[1463841939.970569] 64 bytes from 10.0.0.2: icmp_seq=598 ttl=64 time=13.7 ms
[1463841940.981150] 64 bytes from 10.0.0.2: icmp_seq=599 ttl=64 time=23.2 ms
[1463841941.973286] 64 bytes from 10.0.0.2: icmp_seq=600 ttl=64 time=13.5 ms
[1463841942.970079] 64 bytes from 10.0.0.2: icmp_seq=601 ttl=64 time=9.45 ms
[1463841943.973158] 64 bytes from 10.0.0.2: icmp_seq=602 ttl=64 time=10.6 ms
[1463841944.981612] 64 bytes from 10.0.0.2: icmp_seq=603 ttl=64 time=17.1 ms
[1463841945.984833] 64 bytes from 10.0.0.2: icmp_seq=604 ttl=64 time=18.7 ms
[1463841946.980062] 64 bytes from 10.0.0.2: icmp_seq=605 ttl=64 time=12.2 ms
[1463841947.979020] 64 bytes from 10.0.0.2: icmp_seq=606 ttl=64 time=9.15 ms
[1463841948.975656] 64 bytes from 10.0.0.2: icmp_seq=607 ttl=64 time=4.69 ms
[1463841950.002005] 64 bytes from 10.0.0.2: icmp_seq=608 ttl=64 time=29.4 ms
[1463841950.977752] 64 bytes from 10.0.0.2: icmp_seq=609 ttl=64 time=4.68 ms
[1463841951.982275] 64 bytes from 10.0.0.2: icmp_seq=610 ttl=64 time=7.56 ms
[1463841952.996464] 64 bytes from 10.0.0.2: icmp_seq=611 ttl=64 time=20.4 ms
[1463841954.001877] 64 bytes from 10.0.0.2: icmp_seq=612 ttl=64 time=24.5 ms
[1463841954.986107] 64 bytes from 10.0.0.2: icmp_seq=613 ttl=64 time=8.13 ms
[1463841955.985985] 64 bytes from 10.0.0.2: icmp_seq=614 ttl=64 time=6.89 ms
[1463841957.006020] 64 bytes from 10.0.0.2: icmp_seq=615 ttl=64 time=25.5 ms
[1463841957.986435] 64 bytes from 10.0.0.2: icmp_seq=616 ttl=64 time=5.01 ms
[1463841959.002840] 64 bytes from 10.0.0.2: icmp_seq=617 ttl=64 time=19.4 ms
[1463841959.990295] 64 bytes from 10.0.0.2: icmp_seq=618 ttl=64 time=6.42 ms
[1463841960.990962] 64 bytes from 10.0.0.2: icmp_seq=619 ttl=64 time=5.92 ms
```

```
root@rasp1-desktop: /home/rasp1
    Bridge "bridge1"
        Port ipsec_gre
            Interface ipsec_gre
                type: ipsec_gre
                options: {key="567", psk=pa, remote_ip="129.241.205.101"}
        Port "bridge1"
            Interface "bridge1"
                type: internal
        Port "tap1"
            Interface "tap1"
                type: internal
    ovs_version: "2.4.0"
root@rasp1-desktop:/home/rasp1#
```

```
root@rasp2-desktop: /home/rasp2
    Bridge "bridge2"
        Port ipsec_gre
            Interface ipsec_gre
                type: ipsec_gre
                options: {key="567", psk=password, remote_ip="129.241.205.110"}
        Port "bridge2"
            Interface "bridge2"
                type: internal
        Port "tap2"
            Interface "tap2"
                type: internal
    ovs_version: "2.4.0"
root@rasp2-desktop:/home/rasp2#
```

Figure 5.8: A screen capture, showing the PSK not being applied. The top terminal shows ping traffic going between the two switches, and the two other terminals are from the switches and displays them having different PSKs

added the ports to verify that the tunnel was usable. I was able to add 420 ports and still successfully ping between the hosts. The reason for stopping at 420 was based on time. Thus, 420 can be considered a lower bound. However, the ports are not working if the host is restarted. When using three ports they are unaffected by a restart.

5.2.5 Adding ports performance

The next thing tested was whether having multiple IPsec tunnels affected performance. This experiment was also conducted by running `add_tunnel.py`, which adds a new IPsec tunnel every 10 seconds and adds OpenFlow rules that force the switches to use the new tunnel. While this script was running, I ran Iperf between the hosts to measure throughput. The results can be seen in Table 5.1 and Table 5.2. The script was stopped at 300 ports because there was no tendency of the traffic being affected.

Setup	**Transfer**	**total time**	**Bandwidth**
`300 open IPsec ports`	*104 MBytes*	*60.1 sec*	*14.5 Mbits/sec*
`Normal`	*137 MBytes*	*60.1 sec*	*19.1 Mbits/sec*

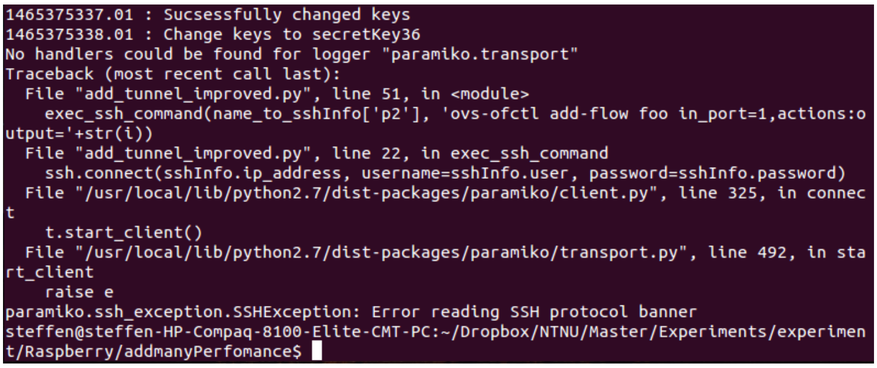
Table 5.1: Throughput over a wireless connection with 300 open IPsec ports.

Setup	**min**	**avg**	**max**	**mdev**	**total time**
`300 open IPsec ports`	*3.805*	*5.890*	*78.253*	*2.868*	*63279ms*
`Normal`	*3.206*	*5.050*	*67.820*	*2.516*	*51421ms*

Table 5.2: Latency over a wireless connection with 300 open IPsec ports.

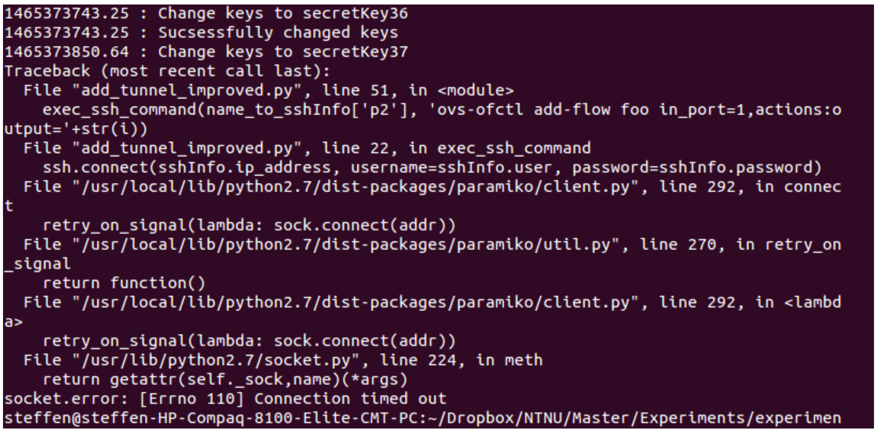
As can be seen in Table 5.2, having 300 open IPsec ports did not have a negative effect on the throughput or latency when compared with to Table 4.5 and Table 4.6. The normal entry in Table 5.2 is testing just between the Raspberry Pi devices without going through OVS.

Since wireless traffic did not seem to haven an effect, the same test was performed over a wired connection. `add_tunnel_improved.py` was ran while running an Iperf throughput test over a wired connection. The connection between the controller and the Raspberry was wireless but on the same AP. Here something interesting happened. While adding ports the wireless connection from the controller to the Raspberry Pi devices broke. The experiment was performed three times and it happened when adding port number 37,41, and 36. Two times the error in Figure 5.10 occurred, and one time the error shown in Figure 5.9.



```
1465375337.01 : Sucsessfully changed keys
1465375338.01 : Change keys to secretKey36
No handlers could be found for logger "paramiko.transport"
Traceback (most recent call last):
  File "add_tunnel_improved.py", line 51, in <module>
    exec_ssh_command(name_to_sshInfo['p2'], 'ovs-ofctl add-flow foo in_port=1,actions:o
utput='+str(i))
  File "add_tunnel_improved.py", line 22, in exec_ssh_command
    ssh.connect(sshInfo.ip_address, username=sshInfo.user, password=sshInfo.password)
  File "/usr/local/lib/python2.7/dist-packages/paramiko/client.py", line 325, in connec
t
    t.start_client()
  File "/usr/local/lib/python2.7/dist-packages/paramiko/transport.py", line 492, in sta
rt_client
    raise e
paramiko.ssh_exception.SSHException: Error reading SSH protocol banner
steffen@steffen-HP-Compaq-8100-Elite-CMT-PC:~/Dropbox/NTNU/Master/Experiments/experimen
t/Raspberry/addmanyPerfomance$ 
```

Figure 5.9: Error message that stopped `add_tunnel_improved.py` while adding IPsec port number 36



```
1465373743.25 : Change keys to secretKey36
1465373743.25 : Sucsessfully changed keys
1465373850.64 : Change keys to secretKey37
Traceback (most recent call last):
  File "add_tunnel_improved.py", line 51, in <module>
    exec_ssh_command(name_to_sshInfo['p2'], 'ovs-ofctl add-flow foo in_port=1,actions:o
utput='+str(i))
  File "add_tunnel_improved.py", line 22, in exec_ssh_command
    ssh.connect(sshInfo.ip_address, username=sshInfo.user, password=sshInfo.password)
  File "/usr/local/lib/python2.7/dist-packages/paramiko/client.py", line 292, in connec
t
    retry_on_signal(lambda: sock.connect(addr))
  File "/usr/local/lib/python2.7/dist-packages/paramiko/util.py", line 270, in retry_on
_signal
    return function()
  File "/usr/local/lib/python2.7/dist-packages/paramiko/client.py", line 292, in <lambd
a>
    retry_on_signal(lambda: sock.connect(addr))
  File "/usr/lib/python2.7/socket.py", line 224, in meth
    return getattr(self._sock,name)(*args)
socket.error: [Errno 110] Connection timed out
steffen@steffen-HP-Compaq-8100-Elite-CMT-PC:~/Dropbox/NTNU/Master/Experiments/experimen
```

Figure 5.10: Error message that stopped `add_tunnel_improved.py` while adding IPsec port number 37 and 41

5.2.6 Adding ports overhead

After establishing that adding ports with IPsec is a good way to renew the PSK it is interesting to see how adding ports affect the performance. It would be desirable to be able to renew the PSK without having to do it ahead of time. Therefore, I ran `add_tunnel.py` while running Iperf on each side of the connection. The results over Ethernet can be seen in Figure 5.11 and over Wi-Fi in Figure 5.12.

As seen in Figure 5.11 and Figure 5.12, adding a port did affect the throughput significantly. You can clearly see drops when the new port is being added. `add_tunnel.py` is supposed to add a tunnel every five seconds but as can be seen in Figure 5.13, adding a tunnel takes more time when the switch is also forwarding

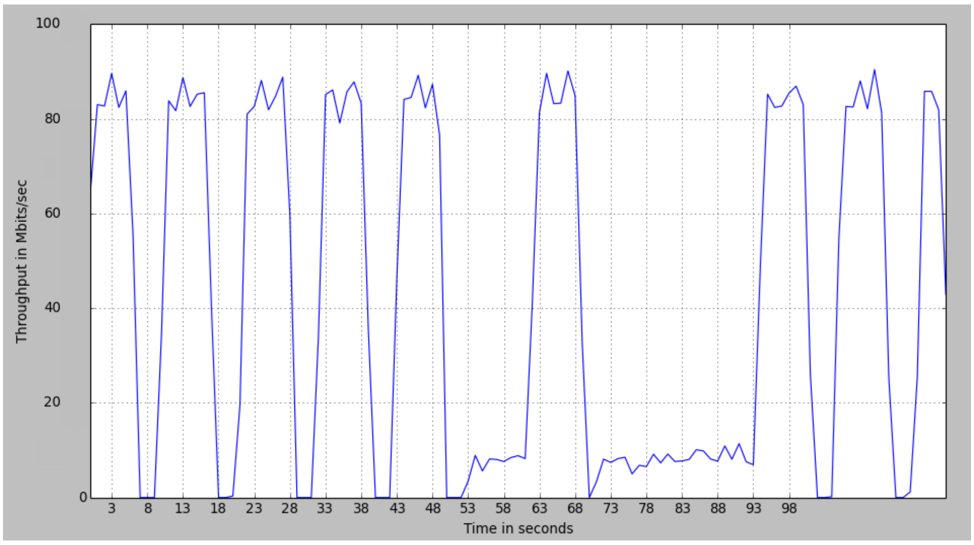


Figure 5.11: The graph shows the throughput on an OVS over Ethernet, when ports are being changed.

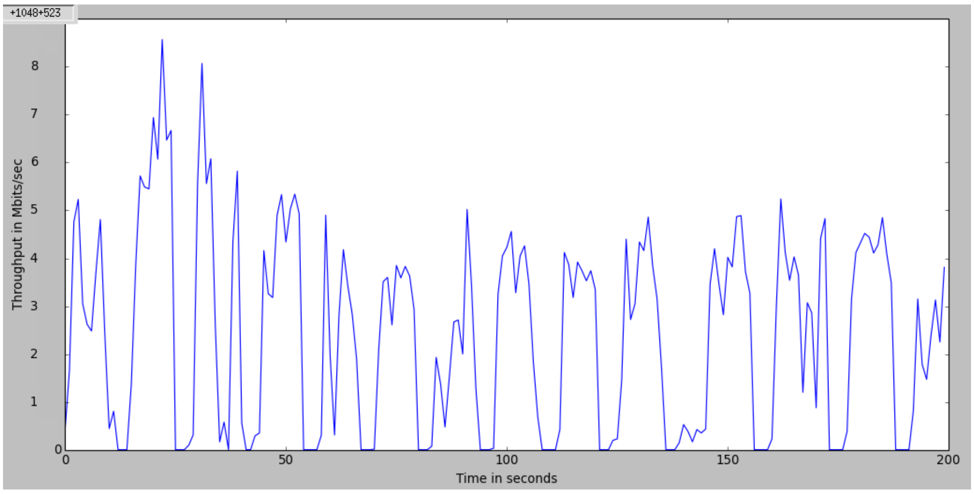


Figure 5.12: This shows how adding a port and changing the OpenFlow rules to used that port effects the throughput over Wi-Fi.

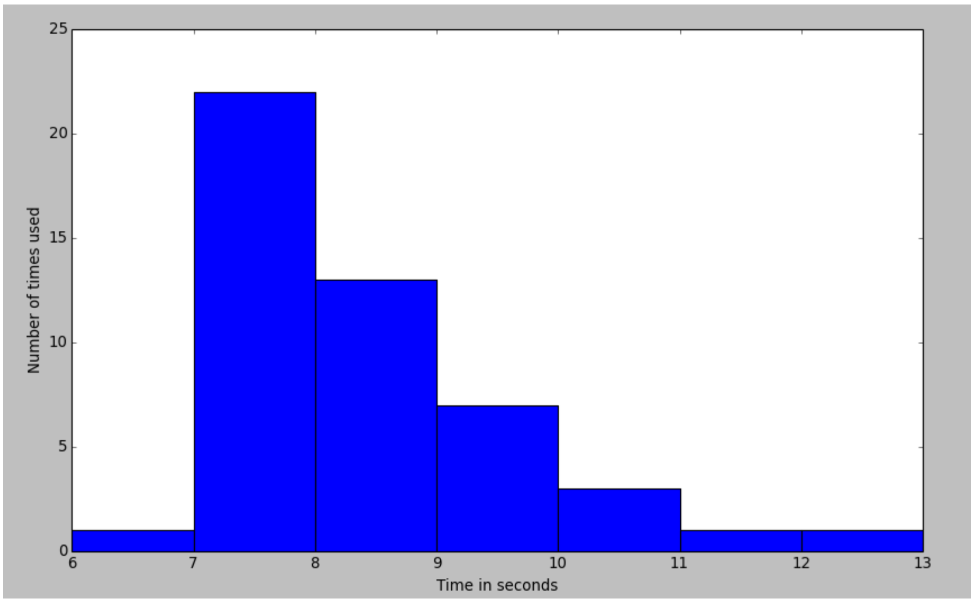


Figure 5.13: The graph shows the time it takes to add a port and change the flow on a switch that is forwarding the traffic.

traffic.

5.2.7 PSK change performance

The next thing to test was throughput when switching ports. To do this the script `port_switcher.py` was used. `port_switcher.py` adds a new OpenFlow rule to switch to an already added IPsec port every 5 seconds. The results can be seen in Figure 5.14. Here we see that changing the port does not affect the throughput nearly as much.

5.2.8 Improved method

From Figure 5.12 and Figure 5.14 we observe that switching between tunnels does not nearly have the same impact on the throughput as adding a tunnel and changing tunnel. From Figure 5.13 we can observe that performing changes on the switch typically takes between 7 and 11 seconds. The `add_tunnel.py` adds a port on switch one and tells it to use the new port. Then it does the same on switch two. Observing the graphs above, we see that this is an inefficient way to do it. During the time the controller adds a port and new flow entry on the second switch all traffic will be lost. Therefore, I changed the code to `add_tunnel_improved.py` which adds the new port on both switches before changing the OpenFlow rules. The result are shown in Figure 5.15.

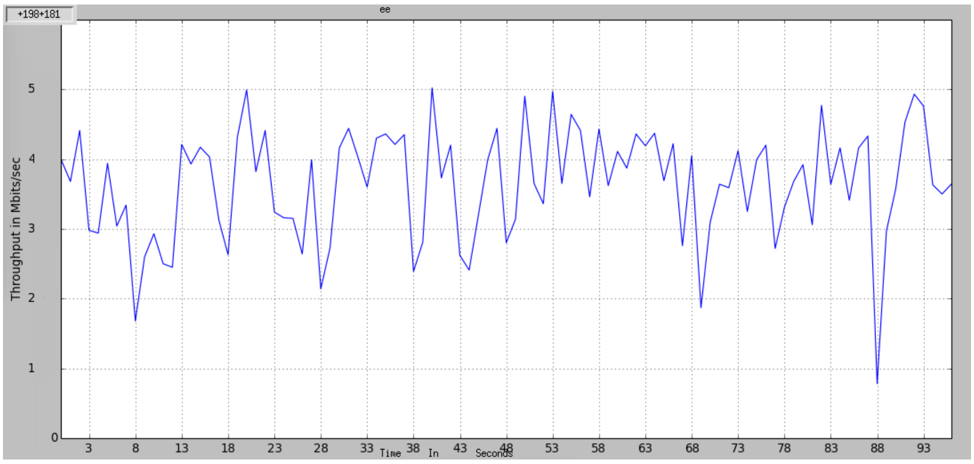


Figure 5.14: The graph shows the throughput over Wi-Fi on an OVS when ports are being changed.

As seen in Figure 5.15 the changes made in `add_tunne_improved.py` made a huge difference in throughput. The reason I did not pay attention to the order the commands were executed in the beginning, was that executing an ssh command took less than a second when there is no traffic on the switch.

5.2.9 Solution

After discovering the security issues with the PSK renewal I notified the security team at OVS, and got the following reply:

' .. I think that the solution to the first problem you reported would be IPsec shunt policy. Basically a low priority IPsec drop policy added by:

ip xfrm policy add src 0.0.0.0/0 dst 0.0.0.0/0 proto gre dir out action block priority 2147999999
Can you try it out and let me know what you think? Though this IPsec policy needs refinement so that "type=gre" tunnels could still get through. Perhaps we could match against skb_mark where it would be set to 1 for IPsec case.'

This worked. No packet leaked. The problem with this xfrm rule is that you are unable to send traffic that does not use IPsec.

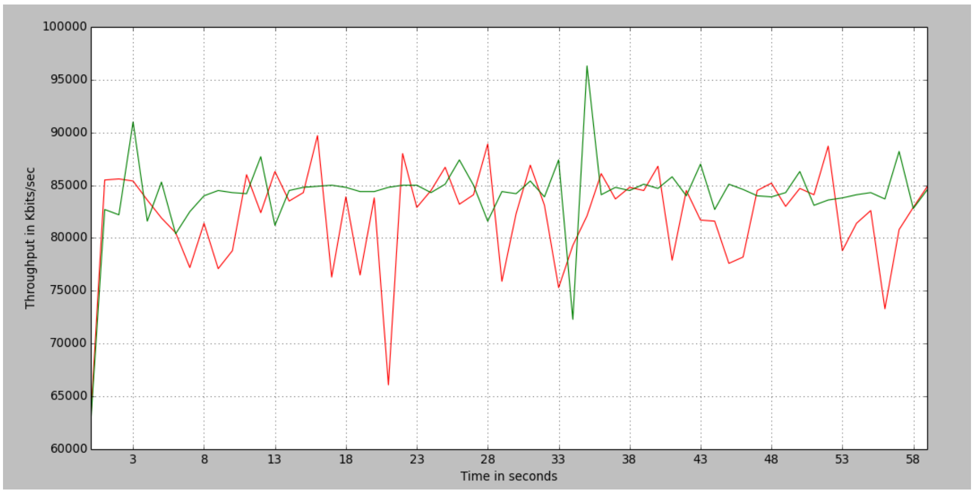


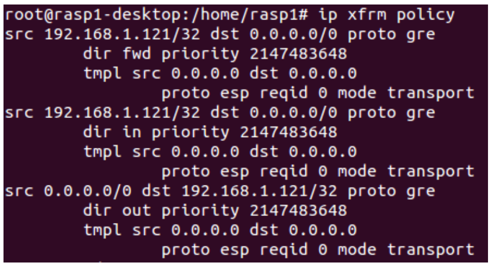
Figure 5.15: The red graph shows the throughput over Ethernet on an OVS when used ports are being switched after they are added on both switches. The green graph is added for reference to shows the throughput over IPsec when ports are not being changed or switched. Ports are being changed every 5 seconds.

5.2.10 Encryption policy

Looking at the original xfrm poliy in Figure 5.16 and Figure 5.17 it seems to be that the policy is based only on IP address. This explains why deleting an IPsec port with remote destination IP x would stop all other traffic from being encrypted. Consequently, this would also mean that an OVS could not have a GRE tunnel and IPsec tunnel open to the same remote IP address at the same time. This assumption was tested by sending traffic between the two Raspberry Pi devices over an IPsec tunnel. Then adding a GRE tunnel, and adding OpenFlow rules to only use the GRE port. The result was connections loss. In Wireshark it was observed that only ESP packets were being transmitted. When deleting the IPsec port the GRE tunnel started working again. The opposite order was also tested: Using GRE, then IPsec, and finally GRE. The result was that switching back to GRE did not work until the IPsec port was deleted.

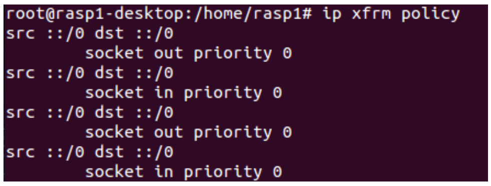
5.2.11 Verifying changes

As seen earlier, OVS does not always behave as according to the configurations. Therefore, I did some tests to verify that the ports added actually were used. I also used `dump-flows` to verify that the correct OpenFlow rules were being used. This can be seen in Figure 5.18. Additionally, I tried changing the key on the receiver side to verify that packets with an incorrect key are not accepted. The result was that the



```
root@rasp1-desktop:/home/rasp1# ip xfrm policy
src 192.168.1.121/32 dst 0.0.0.0/0 proto gre
        dir fwd priority 2147483648
        tmpl src 0.0.0.0 dst 0.0.0.0
                proto esp reqid 0 mode transport
src 192.168.1.121/32 dst 0.0.0.0/0 proto gre
        dir in priority 2147483648
        tmpl src 0.0.0.0 dst 0.0.0.0
                proto esp reqid 0 mode transport
src 0.0.0.0/0 dst 192.168.1.121/32 proto gre
        dir out priority 2147483648
        tmpl src 0.0.0.0 dst 0.0.0.0
                proto esp reqid 0 mode transport
```

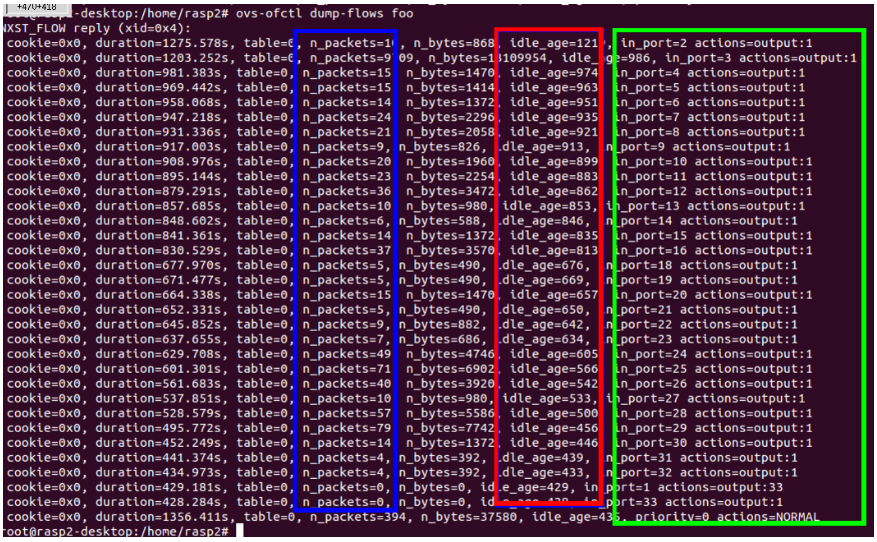
Figure 5.16: The original xfrm policy when creating an IPsec tunnel to 192.168.1.121



```
root@rasp1-desktop:/home/rasp1# ip xfrm policy
src ::/0 dst ::/0
        socket out priority 0
src ::/0 dst ::/0
        socket in priority 0
src ::/0 dst ::/0
        socket out priority 0
src ::/0 dst ::/0
        socket in priority 0
```

Figure 5.17: The xfrm policy when using a GREtunnel to 192.168.1.121

packets were not accepted by the receiver. Therefore, I conclude that dynamically adding new ports is a good way to renew the PSK being used.



```
[illegible]-desktop:/home/rasp2# ovs-ofctl dump-flows foo
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=1275.578s, table=0, n_packets=1[illegible], n_bytes=868, idle_age=121[illegible], in_port=2 actions=output:1
 cookie=0x0, duration=1203.252s, table=0, n_packets=9[illegible]09, n_bytes=1[illegible]109954, idle_age=986, in_port=3 actions=output:1
 cookie=0x0, duration=981.383s, table=0, n_packets=15, n_bytes=1470, idle_age=974, in_port=4 actions=output:1
 cookie=0x0, duration=969.442s, table=0, n_packets=15, n_bytes=1414, idle_age=963, in_port=5 actions=output:1
 cookie=0x0, duration=958.068s, table=0, n_packets=14, n_bytes=1372, idle_age=951, in_port=6 actions=output:1
 cookie=0x0, duration=947.218s, table=0, n_packets=24, n_bytes=2296, idle_age=935, in_port=7 actions=output:1
 cookie=0x0, duration=931.336s, table=0, n_packets=21, n_bytes=2058, idle_age=921, in_port=8 actions=output:1
 cookie=0x0, duration=917.003s, table=0, n_packets=9, n_bytes=826, idle_age=913, in_port=9 actions=output:1
 cookie=0x0, duration=908.976s, table=0, n_packets=20, n_bytes=1960, idle_age=899, in_port=10 actions=output:1
 cookie=0x0, duration=895.144s, table=0, n_packets=23, n_bytes=2254, idle_age=883, in_port=11 actions=output:1
 cookie=0x0, duration=879.291s, table=0, n_packets=36, n_bytes=3472, idle_age=862, in_port=12 actions=output:1
 cookie=0x0, duration=857.685s, table=0, n_packets=10, n_bytes=980, idle_age=853, in_port=13 actions=output:1
 cookie=0x0, duration=848.602s, table=0, n_packets=6, n_bytes=588, idle_age=846, in_port=14 actions=output:1
 cookie=0x0, duration=841.361s, table=0, n_packets=14, n_bytes=1372, idle_age=835, in_port=15 actions=output:1
 cookie=0x0, duration=830.529s, table=0, n_packets=37, n_bytes=3570, idle_age=813, in_port=16 actions=output:1
 cookie=0x0, duration=677.970s, table=0, n_packets=5, n_bytes=490, idle_age=676, in_port=18 actions=output:1
 cookie=0x0, duration=671.477s, table=0, n_packets=5, n_bytes=490, idle_age=669, in_port=19 actions=output:1
 cookie=0x0, duration=664.338s, table=0, n_packets=15, n_bytes=1470, idle_age=657, in_port=20 actions=output:1
 cookie=0x0, duration=652.331s, table=0, n_packets=5, n_bytes=490, idle_age=650, in_port=21 actions=output:1
 cookie=0x0, duration=645.852s, table=0, n_packets=9, n_bytes=882, idle_age=642, in_port=22 actions=output:1
 cookie=0x0, duration=637.655s, table=0, n_packets=7, n_bytes=686, idle_age=634, in_port=23 actions=output:1
 cookie=0x0, duration=629.708s, table=0, n_packets=49, n_bytes=4746, idle_age=605, in_port=24 actions=output:1
 cookie=0x0, duration=601.301s, table=0, n_packets=71, n_bytes=6902, idle_age=566, in_port=25 actions=output:1
 cookie=0x0, duration=561.683s, table=0, n_packets=40, n_bytes=3920, idle_age=542, in_port=26 actions=output:1
 cookie=0x0, duration=537.851s, table=0, n_packets=10, n_bytes=980, idle_age=533, in_port=27 actions=output:1
 cookie=0x0, duration=528.579s, table=0, n_packets=57, n_bytes=5586, idle_age=500, in_port=28 actions=output:1
 cookie=0x0, duration=495.772s, table=0, n_packets=79, n_bytes=7742, idle_age=456, in_port=29 actions=output:1
 cookie=0x0, duration=452.249s, table=0, n_packets=14, n_bytes=1372, idle_age=446, in_port=30 actions=output:1
 cookie=0x0, duration=441.374s, table=0, n_packets=4, n_bytes=392, idle_age=439, in_port=31 actions=output:1
 cookie=0x0, duration=434.973s, table=0, n_packets=4, n_bytes=392, idle_age=433, in_port=32 actions=output:1
 cookie=0x0, duration=429.181s, table=0, n_packets=0, n_bytes=0, idle_age=429, in_port=1 actions=output:33
 cookie=0x0, duration=428.284s, table=0, n_packets=0, n_bytes=0, idle_age=428, in_port=33 actions=output:1
 cookie=0x0, duration=1356.411s, table=0, n_packets=394, n_bytes=37580, idle_age=43[illegible], priority=0 actions=NORMAL
root@rasp2-desktop:/home/rasp2#
```

Figure 5.18: The result of `dump-flows` after adding ports while sending traffic. Each line shows an OpenFlow rule. In the blue rectangle, you can see the amount of packets sent over each OpenFlow rule. In the red, you can see the idle time, which means: seconds since last packet was sent over it. The green rectangle shows what port the packet should be forwarded to when matching on an in port. Port 1 is connected to the host. The third line from the bottom shows that the traffic sent from the host is forwarded to port 33 which is the IPsec port used at that time.

Chapter 6 Discussion

OpenFlow rule modification

From the experiments performed in chapter 5.2 it is shown that the best way to renew a PSK is by creating new IPsec tunnels and then modify the flows to use that tunnel. This can be done manually from the controller over SSH. In these experiments, the OpenFlow rules were installed proactively. Installing OpenFlow rules reactively would add significant overhead because the switch would have to pause traffic when querying the controller for the updated OpenFlow rule and then wait for the controller to add the new port. As seen in Figure 5.13 this could take between 6 and 13 seconds when under heavy traffic load.

Improved Security

As discussed in the introduction OVS takes over the IP stack when adding a physical interface. This limits the possibility to do multiple encapsulations in one switch. However, it has a benefit from a security point of view. It can be used to deal with ARP spoofing. When the physical interfaces are connected to OVS there are two ways to handle the ARP. The first option is just ignoring it and proactively sending the ARP entries to the switch, which would make ARP spoofing impossible. The second option is to handle incoming ARP packets in the controller. In this case, you would in principle be vulnerable to ARP spoofing. However, since you can programmatically control how to handle the packets, you could implement an intrusion detection system to detect malicious behavior.

Channel Noise

The experiments testing throughput and latency performed on the Raspberry Pi devices in 4.4, show that signal noise had a significant impact on both the throughput and latency. The noise was likely caused by other Wi-Fi networks running on overlapping networks. Running the experiments at nighttime gave more accurate and stable results, indicating that traffic generated by the devices connected to the other networks affects the performance. Using a Universal Serial Bus (USB) Wi-Fi dongle on the Raspberry Pi could have improved the achieved throughput. Either way, the results illustrate a lower bound for achievable performance with a wireless IPsec tunnel on a Raspberry Pi.

Infrastructure Mode

The AP I used was working in Infrastructure mode, which means that all the traffic has to go through the AP. However, when I ran Wireshark to listen to the traffic, it shows that the packets are sent to the IP and MAC destination of the receiver, and not the AP. This could mislead some to think that the devices are communicating through Peer-to-Peer mode when they are not.

In Table 4.3 and Table 4.4 we saw that the traffic running from a computer through two Raspberry Pi devices resulted in bad performance. The reason is most likely that the traffic had to go through the AP between each external interface. The results also illustrate the importance of repeating experiments on wireless traffic. Due to the nature of a wireless network, many uncontrolled factors can affect the results. Uncritically running only a few tests on each setup could, as shown in Table 4.3, have lead to the erroneous assumption that IPsec had a lower latency than GRE. Also, we saw that the population standard deviation was much greater than in the wireless experiments in Table 4.5 experiments. This could indicate that there is spread in the results, likely caused by something not working properly.

Physical Limitations

Comparing the throughput over a wired network in Table 4.7 with the throughput in the virtual network in Table 4.2, we see that IPsec has a greater effect on the throughput in the virtual environment. When comparing the throughput in GRE, we see that the throughput is more than doubled in the virtual environment even though it goes through three tunnels as opposed to one in the setup used in Table 4.7. This could indicate that the limiting factor could be the AP or the physical cable and not the encapsulation in itself.

Comparing Table 4.7 with Table 4.6 we see that both encapsulation and IPsec has a significantly larger effect on throughput on a wireless network compared to wired. GRE decreases throughput with 6.3% on a wireless network compared to 2.67% over a wired network. IPsec decreases throughput with 10.27% on a wired network compared to 40.98% on a wireless. Yet, it has to be taken into consideration that this is a 750 ,- NOK device, and it is not specialized to work as a router or switch.

Latency

Comparing the wireless latency in Table 4.5 with the wired latency in Table 4.8, we see that both encapsulation and encryption affects the latency more on the wireless network. However, looking at **mdev** we see that the population standard deviation also is much greater over wireless. Additionally, the average times measured on the wireless network are within the **mdev** of `normal`. Thus, in my test environment, GRE and IPsec did not have a significant impact on latency. Most likely because there were uncontrolled factors that affected the results more. On the wired network, we clearly see that IPsec adds latency. GRE also does, but only 5.93% compared to the 63.26% with IPsec.

Wired Throughput

It became clear from observing `top` that CPU was the bottleneck when using IPsec on a wired network. What has to be taken into consideration is that each of the Raspberry Pi devices were not just forwarding the traffic but also generating and receiving the traffic. On a quad core processor, the maximum usage on one core will be shown as 25% in `top` in Solarix mode, as discussed in the introduction. Figure 4.4 shows that ksoftirqd is maxing out the CPU on one core, and yet the total used CPU was 32 %. It is reasonable to think that using threading would increase the throughput with IPsec, and that the performance seen in Table 4.7 is comparable to what could be expected from a single core unit.

Uneven CPU

Both in the virtual environment, and on the Raspberry Pi devices it was observed that the receiving side uses more CPU power than the sender side. In the virtual environment, it was noticeable when using GRE, and when using IPsec. However, on the Raspberry Pi it was only noticeable with IPsec. When we look at the throughput in Table 4.2 versus Table 4.6 it seems that other factors limited the throughput before the CPU became relevant in IPsec. Why receiving traffic is more costly I was unable to figure out. However, it should be considered when designing a network, to avoid unnecessary bottlenecks. Iperf used more CPU on the server side, although as seen in Figure 4.4 it was running on a different core. Additionally, the R2 (decapsulating/ decrypting side) used more CPU than R1 in the virtual environment, and both were just forwarding the packets.

Secure Switch

Looking at how we can prevent a switch from accessing the sent data, we saw that the switch can still control how it forwards data that is already encrypted in an IPsec tunnel. ESP used in IPsec in OVS does not do any integrity check on the outer header, so the packet can be routed through a network that changes the MAC and IP headers, allowing a certain amount of SDN control in that you can still customize your forwarding rules based on the IP and MAC address. If you have packet on a subnet, you could treat the most significant bits in the IP address as a VLAN tag by changing it and creating customized forwarding rules for that part of the address. However, this does not solve the problem with a switch generating malicious traffic because the header is still not integrity checked. Therefore, the best way to secure a switch is to have IPsec tunnels between each switch, and use a double IPsec tunnel, meaning encapsulating a packet twice, if you want to make sure that the switch can not read the data.

Pox and Python

Pox is written in Python. Python uses an interpreter and is a relatively slow language. This should be taken into consideration when looking at the overhead added by the operations performed in the controller. However, this has not been a part of the scope in this thesis. Additionally, the OpenFlow operations carried out by the controller have been relatively simple.

Pox is considered a good controller for learning and research. However, when working with a real network I would recommend trying out other controllers, like Floodlight. It runs on Java which is faster than Python. Also, when adding the IPsec handling to the controller, the workload will increase. Then, a faster language could have an impact on performance. Finally, I would recommend using a controller platform that provides a Graphical User Interface (GUI), which will make the configuration on the controller faster and easier.

IP in IP

As we saw in Table 4.2 and Table 4.1, double encapsulation does add significant overhead to both latency and throughput. The interesting thing, however, is that the double encapsulation had a different effect on latency and throughput. The latency was smaller when using a tunnel between each host, but the throughput was greater. Increasing throughput when doing a double encapsulation makes sense because less encryption and decryption operations are done in the double encapsulation:
ipsec-between-ipsec: C1 encrypts, R1 decrypts and then encrypts, the same with R2 and C2 finally decrypts.
ipsec- in - ipsec : C1 encrypts, R1 encypts, R2 decrypts, C2 decrypts.
R1 and R2 need to do respectively one extra decryption and encryption in ipsec-between-ipsec. Why this does not lead to a higher latency, is still unclear. Although looking at Table 4.1 we see that the difference in average time is 0.243 milliseconds. With both tests having a standard deviation of 1.1 milliseconds the difference is to

small to be considered relevant in a test lasting for less than 30 seconds. To further investigate if there is a difference, a longer test should be performed.

Security Policy

The security issue shown in 5.2.1 regarding unencrypted packets leaking is a serious problem. The xfrm configuration did solve the problem but limits the switch only to use IPsec. What this also shows, is that OVS needs more security auditing. There might very well be more undiscovered vulnerabilities.

The problem with removing IPsec on all switches when one port with the same remote IP is removed shown in 5.2.2 illustrate a second important point. OVS is not designed with the purpose of changing PSK and having multiple tunnels to the same IP address. We saw in 5.2.10 that the xfrm policy only allows one policy per IP address, putting a limitation on OVS to either having only encrypted traffic or only encapsulated traffic.

IPsec in IPsec

Using IPsec in IPsec would also allow for switches to encrypt all its traffic without having to check if the packet is already encrypted, which would save CPU power and decrease latency. Also, as opposed to just forwarding an IPsec tunnel, switches could use all the fields used in SDN without having access to the packet data. A switch could decrypt the outer IPsec layer, then use and modify those headers, before encrypting it with IPsec again, and forwarding it.

Additionally, this could be useful if the encrypted data is so valuable that it is desired to encrypt the data as a shared secret. A scenario where this could be applicable is in a military setting or a financial institution. Trusting one person with the encryption of the traffic sent over a public network could be a risk. Encrypting it twice with different (presumably) PSKs, set by two switches handled by different people, would require both to be compromised in order to decrypt the traffic. Finally, this principle could be applied to cope with hardware failure.

Lastly, it can work as a countermeasure against a malicious switch. Routers are known to contain bugs [CR08], and malicious switches in SDN can potentially take down large parts of a network by either dropping packets or sending them to the wrong destination [KF15]. With IPsec in IPsec, a malicious switch would not be able to do anything with the originally sent packet. Secondly, switching PSK in adjacent switches would prevent it from successfully sending any traffic.

Chapter 7 Conclusion

In this thesis, I have illustrated that it is fully possible to create an SDN that can dynamically set up encryption to separate dynamic virtual networks by extending an SDN controller. I have shown that OVS can be used as an OpenFlow switch that also encrypts the traffic using IPsec with a PSK distributed by the SDN controller. In a virtual environment, it has been shown that OVS can encrypt a packet multiple times using different PSK, and also what is required to make this work. The biggest challenge is the handling of the IP stack in combination with the virtual tunnels controlled by OVS. Also, the MTU has to be considered, as encryption and encapsulation increase the packet size.

Using Raspberry Pi devices, the double packet encapsulation was performed in conjunction with a virtual environment. Dynamic use of encryption and PSK distribution was tested on a wireless network between two Raspberry Pi devices. Additionally, the performance and security aspects have been tested. The results show that using encryption does affect the throughput and that there are security issues that should be considered. Different methods for changing the PSK have been tested out from a security perspective and showed that adding new port first, then changing the OpenFlow rule is the best way to achieve this. Additionally, it has been demonstrated that if you change the PSK used on a port, it pauses all traffic going through any port on the switch until the ISAKMP is completed. It has been shown that deleting an IPsec port, stops encryption of all tunnels with the same remote IP address. Changing PSK by reconfiguring the interface was shown to leak some packets during the transmission. This was also tested in a virtual environment, to verify that it was not a hardware issue. Countermeasures using xfrm were shown to stop all unencrypted packets from leaving the host. A simple replay attack was tested on the GRE and IPsec tunnels. The GRE tunnel was vulnerable, but the IPsec tunnel was not. Some bugs in the OVS were discovered and reported. It was shown that OVS does not always apply the configuration, even if the console says it does.

Using SSH with Paramiko worked well to perform remote commands including PSK exchanges, except when adding over 36 IPsec ports while running a heavy traffic load over a wired network. Some framework needed to be built up to integrate the SSH commands with the SDN controller.

This thesis concludes that OVS has the potential to be used as a secure SDN switch. Although, there is still some security aspects that needs to be fixed before OVS can be considered secure. In the meantime, some additional configurations can be used to patch the problem.

7.1 Future Work

7.1.1 Native support of datapath encryption

For this thesis, I have used a separate SSH connection between the controller and switch to manage the configurations of the GRE and IPsec_GRE tunnels. However, it should be possible to use the existing secured OpenFlow connection between the switch and controller to handle this communication as well. This would make it easier to extend the controller-switch configuration possibilities and a step towards dynamic encryption of SDN.

OpenFlow only handles messages and configurations related to the data forwarding plane. A suggestion for future work would be to extend the OpenFlow protocol to manage the SAD and SPD between the switches. An argument against OpenFlow handling is that it would be switch specific, and therefore different for each switch software. However, as shown in this thesis, it is possible to create a framework that handles the security of the data plane with Pox and OVS. Thus, it could be possible to integrate OVS control into OpenFlow. Open vSwitch is an open source virtual switch licensed under the open source Apache 2.0 license. Doing this would require one to modify the OpenFlow, OVS, and Pox source code. Note that any other controller software could be used. If OVS is chosen as switch software, it would be useful to look at ways to integrate OVSDB. Creating an SDN architecture that enforces datapath encryption would be helpful because it would enable a free SDN architecture that could securely be deployed in wireless and uncontrolled environments. Making this an integrated easily configurable setting would help increase the security and work against the tendency seen where encryption is becoming optional to enable easier configuration.

7.1.2 Improving documentation

There is room for improvement when it comes to documenting the configuration of open vSwitch. There is a configuration cookbook for setting up GRE tunnels with OVS but when it comes to IPsec tunnels, there is only the 'Open_vSwitch database schema' [ovsb]. Because of this, the process of getting the IPsec tunnels to work consisted of trail and error. I discovered by chance that you need to run the command `sudo racoon` before adding the IPsec tunnels in order for them to work, and I have yet to find this documented anywhere. It hit me that, there, in general was little documentation on the setup related to IPsec_GRE, compared to the other tunneling protocols. If we want to motivate people to secure their network, it is paramount that we make it easy to configure. Additionally, documentation and information about how OVS works under the hood would have helped a lot.

7.1.3 Fix bugs

The bugs in OVS that were discussed in Chapter 5 that definitely should be fixed. As mentioned, I have sent one bug report that can be found here: http://comments.gmane.org/gmane.linux.network.openvswitch.general/12771. The security issues were sent to security@openvswitch.org and are not publicly listed. The issues with the encryption in the IPsec tunnels in OVS are being looked into by the security team. The current mail thread can be seen in the appendix.

7.1.4 Hack OVS routing

As I have mentioned in this thesis, OVS takes over the control of the interfaces, but still relies on the hosts IP stack for tunneling. This created some challenges in the setup and limitations regarding what a single OVS can do. I would suggest as future work to look further into how the internal forwarding in OVS works to see if there are alternative methods, or hacks that can affect how OVS forwards the traffic. What would be interesting is to enable a single OVS instance to do both forwarding on the hosts physical interfaces, and at the same time use those interfaces for GRE tunnels. I would suggest looking at ways to use namespacing to solve this.

7.1.5 Additional testing

There are many scenarios regarding SDN and IPsec that could and should be researched. The performance tests could be run on the 5 Giga hertz band to see if that would give any improvements. There were some varying results when adding ports over ssh. There seemed to be some issues when adding multiple ports over SSH without a pause between the SSH commands. Adding PSK on ports without deleting them could be a potential point for DOS attacks. Using an old PSK could be used to force the switch to decrypt the traffic and therefore using significant CPU power.

References

[cis] "wan design guide". https://www.cisco.com/c/dam/en/us/td/docs/solutions/CRD/Jul2015/CRD-WAN_Design_Jul2015.pdf. Accessed: 11 June 2016.

[CR08] Matthew Caesar and Jennifer Rexford. Building bug-tolerant routers with virtualization. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 51–56. ACM, 2008.

[git] "Github reposetory containing files". https://github.com/Steffb/MstThesis. Accessed: 11 June 2016.

[GK14] Prasad Gorja and Rakesh Kurapati. Extending open vswitch to l4-l7 service aware openflow switch. In *Advance Computing Conference (IACC), 2014 IEEE International*, pages 343–347. IEEE, 2014.

[gre] "rfc 2784 - Generic Routing Encapsulation (GRE)". https://tools.ietf.org/html/rfc2784. Accessed: 11 June 2016.

[ipea] "iPerf 3 user documentation". https://iperf.fr/iperf-doc.php. Accessed: 11 June 2016.

[ipeb] "What is iPerf". https://iperf.fr/, note = Accessed: 11 June 2016.

[ipsa] "HMAC-MD5 IP Authentication with Replay Prevention". https://tools.ietf.org/html/rfc2085. Accessed: 11 June 2016.

[ipsb] "IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap". https://tools.ietf.org/html/rfc6071. Accessed: 11 June 2016.

[ipsc] "Other Packages Related to openvswitch-ipsec". https://packages.debian.org/sid/openvswitch-ipsec. Accessed: 11 June 2016.

[JASD12] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 127–132. ACM, 2012.

[KF15] Andrzej Kamisiński and Carol Fung. Flowmon: Detecting malicious switches in software-defined networks. In *Proceedings of the 2015 Workshop on Automated Decision Making for Active Cyber Defense*, pages 39–45. ACM, 2015.

[KKS13] Rowan Kloti, Vasileios Kotronis, and Paul Smith. Openflow: A security analysis. In *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pages 1–6. IEEE, 2013.

[ksoa] "chapter 10: : Interrupt Handling". https://static.lwn.net/images/pdf/LDD3/ch10.pdf. Accessed: 11 June 2016.

[ksob] "ksoftirqd — Softirq daemon". http://www.ms.sapientia.ro/~lszabo/unix_linux_hejprogramozas/man_en/htmlman9/ksoftirqd.9.html. Accessed: 11 June 2016.

[mina] "Mininet overview". http://mininet.org/overview/. Accessed: 11 June 2016.

[minb] "Mininet VM Images". https://github.com/mininet/mininet/wiki/Mininet-VM-Images. Accessed: 11 June 2016.

[net] "ip-netns - process network namespace management". http://man7.org/linux/man-pages/man8/ip-netns.8.html. Accessed: 11 June 2016.

[ofaa] "Flowgrammable, instructions actions". http://flowgrammable.org/sdn/openflow/actions/. Accessed: 11 June 2016.

[ofab] "Mininet overview". http://opennetsummit.org/archives/apr12/hoelzle-tue-openflow.pdf. Accessed: 11 June 2016.

[ofsa] "6.1.1 Controller-to-Switch". https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf. Accessed: 11 June 2016.

[ofsb] "OpenFlow Switch Specification, section 5.1 Pipeline Processing". https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf. Accessed: 11 June 2016.

[ofsc] "OpenFlow Switch Specification, section 6.3.3 encryption". https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf. Accessed: 11 June 2016.

[ofsd] "OpenFlow Switch Specification, section A.2.3.7 Flow Match Fields". https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf. Accessed: 11 June 2016.

[ofv] "OpenFlow Vulnerability Assessment". http://www.ljean.com/files/vulnerability_analysis.pdf. Accessed: 11 June 2016.

[onf] "Software-Defined Networking (SDN) Definition". https://www.opennetworking.org/sdn-resources/sdn-definition. Accessed: 11 June 2016.

[ope] "Software-Defined Networking: The New Norm for Networks". https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf. Accessed: 11 June 2016.

[ovsa] "Open vSwitch site". http://openvswitch.org/, note = Accessed: 11 June 2016.

[ovsb] "Open_vSwitch manual". http://openvswitch.org/ovs-vswitchd.conf.db.5.pdf. Accessed: 11 June 2016.

[ovsc] "ovs-ofctl manual". http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt. Accessed: 11 June 2016.

[para] "Paramiko Client doc on Github". https://github.com/paramiko/paramiko/blob/master/paramiko/client.py. Accessed: 11 June 2016.

[parb] "Welcome to Paramiko". http://www.paramiko.org/. Accessed: 11 June 2016.

[pin] "Linux ping reference". http://linux.about.com/od/commands/l/blcmdl8_ping.htm. Accessed: 11 June 2016.

[pox] "POX Wiki". https://openflow.stanford.edu/display/ONL/POX+Wiki. Accessed: 11 June 2016.

[PSY$^+$12] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A security enforcement kernel for openflow networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 121–126. ACM, 2012.

[ras] "RASPBERRY PI 3 MODEL B". https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. Accessed: 11 June 2016.

[rfc] "Key and Sequence Number Extensions to GRE". https://tools.ietf.org/html/rfc2890. Accessed: 11 June 2016.

[rou] "LINKSYS EA2700 N600 DUAL-BAND SMART WI-FI WIRELESS ROUTER". http://www.linksys.com/fi/p/P-EA2700/#product-features. Accessed: 11 June 2016.

[Sam15] Dominik Samociuk. Secure communication between openflow switches and controllers. *AFIN 2015*, page 39, 2015.

[SG12] Seungwon Shin and Guofei Gu. Cloudwatcher: Network security monitoring using openflow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?). In *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pages 1–6. IEEE, 2012.

[top] "Linux top reference". http://linux.about.com/od/commands/l/blcmdl1_top.htm. Accessed: 11 June 2016.

[vbo] "Virtualbox Virtual networking". https://www.virtualbox.org/manual/ch06.html#network_internal. Accessed: 11 June 2016.

Chapter 8

Appendix

Gmail

Steffen Birkeland <steffenf.birkeland@gmail.com>

Re: [ovs-security] Ipsec_gre leaking

6 e-poster

Jesse Gross <jesse@kernel.org> 25. mai 2016 kl. 22.23
Til: Steffen Fredrik Birkeland <steffefb@stud.ntnu.no>
Kopi: "security@openvswitch.org" <security@openvswitch.org>

On Tue, May 24, 2016 at 2:15 AM, Steffen Fredrik Birkeland
<steffefb@stud.ntnu.no> wrote:
> Hi,
>
>
> I am currently working with open vswitch for my master thesis and I have
> discovered some problems with the Ipsec_gre tunnelling option in vswitch.

Thank you for the report. We will do an analysis and get back to you
if we need more information or with proposed resolutions.

Jesse Gross <jesse@kernel.org> 27. mai 2016 kl. 22.13
Til: Steffen Fredrik Birkeland <steffefb@stud.ntnu.no>, Ansis Atteka <aatteka@vmware.com>
Kopi: "security@openvswitch.org" <security@openvswitch.org>

On Tue, May 24, 2016 at 2:15 AM, Steffen Fredrik Birkeland
<steffefb@stud.ntnu.no> wrote:
> Hi,
>
>
> I am currently working with open vswitch for my master thesis and I have
> discovered some problems with the Ipsec_gre tunnelling option in vswitch.
>
>
> 1. If you change the psk used the switch will leak a packet during the
> process.
>
> I ping through the tunnel while changing the key, and one packet always gets
> through unencrypted.
>
>
> I use the command:
>
> ovs-vsctl set interface <portname> options:psk=<new psk > to do this.
>
>
> 2. when multiple Ipsec_gre tunnels on a vswitch changing the psk on one
> port will also pause the traffic on the other ports until the ISAKMP
> completes.
>
>
> 3. If I have mulitple Ipsec_gre tunnels deleting one of them will result in
> the other stopping using ipsec, and just becoming a normal gre tunnel.
>
>
>
> I don't know if thees are known issues, I can provide pcap and log files if
> this is interesting.
>
> I use ovs-vsctl (Open vSwitch) 2.4.0

Steffen, thanks again for reporting the issue. We've been
investigating the problems that you've described and also discussing
the appropriate actions for any releases.

Ansis is the primary author of the IPsec code in Open vSwitch and has

been working on possible solutions. He has a couple follow up
questions for you, so I've added him to this thread.

Steffen Birkeland <Steffefb@stud.ntnu.no> 29. mai 2016 kl. 22.35
Til: Jesse Gross <jesse@kernel.org>
Kopi: Ansis Atteka <aatteka@vmware.com>, "security@openvswitch.org" <security@openvswitch.org>

Ok,
I am happy to help!

Steffen
[Sitert tekst skjult]

Ansis Atteka <aatteka@vmware.com> 29. mai 2016 kl. 23.08
Til: Steffen Birkeland <Steffefb@stud.ntnu.no>, Jesse Gross <jesse@kernel.org>
Kopi: "security@openvswitch.org" <security@openvswitch.org>

Hi Steffen,

Thanks for reporting this issue.

I think that the solution to the first problem you reported would be IPsec shunt policy. Basically a low priority IPsec drop policy added by:

```
# ip xfrm policy add src 0.0.0.0/0 dst 0.0.0.0/0 proto gre dir out action block priority 2147999999
```

Can you try it out and let me know what you think? Though this IPsec policy needs refinement so that "type=gre" tunnels could still get through. Perhaps we could match against skb_mark where it would be set to 1 for IPsec case.

Regarding third issue. I wasn't able to reproduce it with a simple "ovs-vsctl del-port ipsec_gre0" command. Could you provide exact commands that you used to set up tunnels and then delete them?

Thanks,

Ansis

From: Steffen Birkeland <Steffefb@stud.ntnu.no>
Sent: Sunday, May 29, 2016 1:35 PM
To: Jesse Gross

8.1 add_tunnel.py

```
import paramiko

import time

class sshInfo:
    def __init__(self ,ip_address , user, password, ID_name):
      self.user = user
      self.ip_address = ip_address
      self.ID_name = ID_name
      self.password = password


name_to_sshInfo = {
'p1': sshInfo('192.168.1.146','rasp1','reverse', 'rasp1'),
'p2': sshInfo('192.168.1.122','rasp2','reverse', 'rasp2')
}

def exec_ssh_command(sshInfo, command):
      ssh = paramiko.SSHClient()
      ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
      ssh.connect(sshInfo.ip_address, username=sshInfo.user,
          password=sshInfo.password)
      if(command[:3]=='ovs'): command= 'sudo '+command
      stin, out, err = ssh.exec_command(command)
      #print '[exec_ssh_command (in)] %s'%(command)
      #print '[exec_ssh_command (out)] %s'%(out.readlines())

      #display_attr(out)
      #display_attr(err)
      if(err.read()): print '[exec_ssh_command (error)] %s
          !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
          Error'%(err.readlines())
      ssh.close()

exec_ssh_command(name_to_sshInfo['p1'], 'ovs-vsctl add-br foo;
    ovs-vsctl add-port foo tap -- set interface tap type=internal; ip
    netns add ns1; ip link set tap netns ns1 ; ip netns exec ns1
    ifconfig tap up ; ip netns exec ns1 ifconfig tap inet 11.0.0.1 ;
    ip netns exec ns1 bash')

exec_ssh_command(name_to_sshInfo['p2'], 'ovs-vsctl add-br foo;
    ovs-vsctl add-port foo tap -- set interface tap type=internal; ip
    netns add ns2; ip link set tap netns ns2 ; ip netns exec ns2
    ifconfig tap up ; ip netns exec ns2 ifconfig tap inet 11.0.0.2 ;
    ip netns exec ns2 bash')



for i in range(6,1100):
        newkey= 'secretKey'+str(i)
        newPort= 'ipsecPort'+str(i)
        unixtime= str(time.time())
        print '%s : Change keys to %s'%(unixtime, newkey)
        exec_ssh_command(name_to_sshInfo['p1'], 'ovs-vsctl add-port
            foo '+newPort+' -- set interface '+newPort+'
            type=ipsec_gre options:remote_ip=192.168.1.121
            options:key='+str(i)+' options:psk=%s'%(newkey))
        exec_ssh_command(name_to_sshInfo['p1'], 'ovs-ofctl add-flow
            foo in_port=1,actions:output='+str(i))
        exec_ssh_command(name_to_sshInfo['p1'], 'ovs-ofctl add-flow
            foo in_port='+str(i)+',actions:output=1')

        exec_ssh_command(name_to_sshInfo['p2'], 'ovs-vsctl add-port
            foo '+newPort+' -- set interface '+newPort+'
            type=ipsec_gre options:remote_ip=192.168.1.143
            options:key='+str(i)+' options:psk=%s'%(newkey))
        exec_ssh_command(name_to_sshInfo['p2'], 'ovs-ofctl add-flow
            foo in_port=1,actions:output='+str(i))
        exec_ssh_command(name_to_sshInfo['p2'], 'ovs-ofctl add-flow
            foo in_port='+str(i)+',actions:output=1')


        print '%s : Sucsessfully changed keys '%(unixtime)
        time.sleep(5)
        #raw_input("Press Enter to add tunnel...")


print 'Done'
```

8.2 add_tunnel_improved.py

```

import paramiko

import time

class sshInfo:
    def __init__(self ,ip_address , user, password, ID_name):
      self.user = user
      self.ip_address = ip_address
      self.ID_name = ID_name
      self.password = password





exec_ssh_command(name_to_sshInfo['p1'], 'ovs-vsctl add-br foo;
    ovs-vsctl add-port foo tap -- set interface tap type=internal; ip
    netns add ns1; ip link set tap netns ns1 ; ip netns exec ns1
    ifconfig tap up ; ip netns exec ns1 ifconfig tap inet 11.0.0.1 ;
    ip netns exec ns1 bash')
```

```
exec_ssh_command(name_to_sshInfo['p2'], 'ovs-vsctl add-br foo;
    ovs-vsctl add-port foo tap -- set interface tap type=internal; ip
    netns add ns2; ip link set tap netns ns2 ; ip netns exec ns2
    ifconfig tap up ; ip netns exec ns2 ifconfig tap inet 11.0.0.2 ;
    ip netns exec ns2 bash')



for i in range(2,1100):
        newkey= 'secretKey'+str(i)
        newPort= 'ipsecPort'+str(i)
        unixtime= str(time.time())
        print '%s : Change keys to %s'%(unixtime, newkey)
        exec_ssh_command(name_to_sshInfo['p1'], 'ovs-vsctl add-port
            foo '+newPort+' -- set interface '+newPort+'
            type=ipsec_gre options:remote_ip=192.168.1.121
            options:key='+str(i)+' options:psk=%s'%(newkey))
        exec_ssh_command(name_to_sshInfo['p2'], 'ovs-vsctl add-port
            foo '+newPort+' -- set interface '+newPort+'
            type=ipsec_gre options:remote_ip=192.168.1.143
            options:key='+str(i)+' options:psk=%s'%(newkey))

        exec_ssh_command(name_to_sshInfo['p1'], 'ovs-ofctl add-flow
            foo in_port=1,actions:output='+str(i))
        exec_ssh_command(name_to_sshInfo['p1'], 'ovs-ofctl add-flow
            foo in_port='+str(i)+',actions:output=1')


        exec_ssh_command(name_to_sshInfo['p2'], 'ovs-ofctl add-flow
            foo in_port=1,actions:output='+str(i))
        exec_ssh_command(name_to_sshInfo['p2'], 'ovs-ofctl add-flow
            foo in_port='+str(i)+',actions:output=1')


        print '%s : Sucsessfully changed keys '%(unixtime)
        time.sleep(5)
        #raw_input("Press Enter to add tunnel...")


print 'Done'
```

8.3 mycontroller.py

This is the controller code. Most of the functions are loaded from helper_functions which 500 lines of code. It can be found here https://github.com/Steffb/MstThesis/blob/master/Pox/helper_functions.py

```
from helper_functions import *


log = core.getLogger()

# This is the main controller that sets up the switches
# The commented out functions are specific exeperiment setup that can
    be found in helper functions.

# This acts as a frame for setting up the switches, all the code
    actually doing the operations are in helper_functions.py


_flood_delay = 0



class LearningSwitch (object):
  """ This class represents an switch instance """

  def __init__ (self, connection, transparent):
    # Switch we'll be adding L2 learning switch capabilities to
    self.connection = connection
    self.transparent = transparent
    try:
      connections[dpid_to_name[connection.dpid]]= connection


    except:
      print 'Unable to add the connection for %d'%(connection.dpid)
      for p in connection.features.ports:
        print 'The HW_addr is: '+str(p.hw_addr)


    #Exp 1
    #connect_gre_tunnels(connection)

    #Exp 2
    #connect_gre_in_gre(connection)

    #Exp 3
    #connect_ipsec_gre_tunnels(connection)

    #Exp 4
    #ipsec_connect_gre_in_gre(connection)

    #Exp 5
    #rasp_connect_gre_tunnels(connection)

    #Exp 6
    #rasp_connect_ipsec_gre_tunnels(connection)
    if(connection.dpid in dpid_to_name.keys() and
        dpid_to_name[connection.dpid] == 'p1'):



    if(connection.dpid in dpid_to_name.keys() and
        dpid_to_name[connection.dpid] == 'p2'):


    if(connection.dpid in dpid_to_name.keys() and
        dpid_to_name[connection.dpid] == 'R1'):

      pass
    if(connection.dpid in dpid_to_name.keys() and
        dpid_to_name[connection.dpid] == 'R2'):

      pass
    if(connection.dpid in dpid_to_name.keys() and
        dpid_to_name[connection.dpid] == 'R3'):
      pass
    if(connection.dpid in dpid_to_name.keys() and
        dpid_to_name[connection.dpid] == 'C1'):
      pass

    if(connection.dpid in dpid_to_name.keys() and
        dpid_to_name[connection.dpid] == 'C2'):
      pass

      #create_flow(connection, in_port,out_port):



    # Our table
    self.macToPort = {}

    # We want to hear PacketIn messages, so we listen
    # to the connection
    connection.addListeners(self)

    # We just use this to know when to log a helpful message
    self.hold_down_expired = _flood_delay == 0

    #log.debug("Initializing LearningSwitch, transparent=%s",
    #           str(self.transparent))

  def _handle_PacketIn (self, event):
    # Triggered when switch doesn't have a matching flowrule
    print '[packet_in] from '+dpid_to_name[event.dpid]
    packet = event.parsed
    print '[packet_in] content:\n '+str(packet )


    connection= event.connection




class l2_learning (object):
  """
  Waits for OpenFlow switches to connect and makes them learning
      switches.
  This is the running object

  _handle_connection is running once for each switch connecting

  """
  def __init__ (self, transparent):

    core.openflow.addListeners(self)
    self.transparent = transparent

  def _handle_ConnectionUp (self, event):

    log.debug("Connection %s" % (event.connection,))
    LearningSwitch(event.connection, self.transparent)

    try:
      print dpid_to_name[event.dpid]+ ' connected'
    except:
      print str(event.dpid)+ ' was is not in local dict'






def launch (transparent=False, hold_down=_flood_delay):
  """
  Starts an L2 learning switch.
  """
  #THis needs to run when endc1 and endc2 are connecting!
  print ' Not connecting endc1 and endc2 !!!!!!!!!!!!!!!! '
  #connect_ends()

  print 'not adding arp and route to rasps'
  #connect_raspberries()

  #exp1 creates gre tunnels between c1,c2,r1,r2
  #gre_between_all()
```

```
#exp2
#gre_in_gre()

#exp3
#ipsec_gre_between_all()

#exp4
#ipsec_gre_in_gre()

#exp5
#rasp_gre_between_all()

#exp6
#rasp_ipsec_gre_between_all()

#exp7
#rasp_to_rasp_double_tunnel()

#exp8
#rasp_to_rasp_verify_ipsec()


try:
  global _flood_delay
  _flood_delay = int(str(hold_down), 10)
  assert _flood_delay >= 0
except:
  raise RuntimeError("Expected hold-down to be a number")
print 'running core'
core.registerNew(l2_learning, str_to_bool(transparent))
```