CHALMERS

SSH over UDP

Master of Science Thesis in the Programmes
Secure and Dependable Computer Systems
Networks and Distributed Systems

MAGNUS ULLHOLM KARLSSON

MD. AHASAN HABIB

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, July 2010

SSH over UDP

Magnus Ullholm Karlsson,
Md. Ahasan Habib

Examiner: TOMAS OLOVSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden July 2010

TO MY FAMILY

-MAGNUS ULLHOLM KARLSSON

TO MY PARENTS AND WIFE

-MD. AHASAN HABIB

Acknowledgements

This thesis could not have been prepared without the excellent supervision of Tomas Olovsson. The necessary resources and information have been supplied by Chalmers University of Technology. Therefore, we want to thank Tomas Olovsson and Chalmers University of Technology.

Abstract

The SSH protocol provides many invaluable network features over encrypted channels. In version 4.3 of the OpenSSH implementation, VPN functionality is also supported, where actual IP packets from other applications are captured and tunneled via OpenSSH to the remote location. OpenSSH is using TCP consistently for all its network connections and thus for its VPN feature. This causes the VPN feature to tunnel one TCP connection within another TCP connection. Many sources say that TCP in TCP tunneling, under realistic conditions, can give rise to conflicts between the two TCP implementations and that TCP in TCP should be avoided. Many SSH and SSL VPN solutions use this functionality anyway and it seems to work.

To see whether a UDP based solution would perform better than a TCP based solution on links experiencing packet loss, we have modified the OpenSSH implementation by adding support for a UDP base connection to its VPN functionality. The modification was tested and compared to the original implementation using a test network, in which packet loss was emulated. The performance of the implementations is compared in terms of bandwidth for different rates of packet loss. We have shown that a UDP based solution performs slightly better than a TCP based solution. The most gain in performance, from using a UDP base connection, was detected when ACKs belonging to the tunneled connection where lost.

Keywords: VPN, SSH, OpenSSH, UDP, TCP in TCP, packet loss

Index

Chapter 1, Introduction

1.1 Background

A Virtual Private Network, VPN, can be created in order to connect hosts over an underlying public network, using confidential connections. Typically, the VPN connections consist of encrypted tunnels using standardized block ciphers to encrypt the tunnelled traffic. A VPN is a logical network on top of an already existing network and there are different VPN solutions that work on different layers in the OSI model. In the tunnels, the forwarded traffic is encrypted and sent through the tunnel using the beneath layers in the OSI model.

A very useful feature of a VPN is that it can be used to connect separate private networks by using gateways with VPN functionality. The traffic can then be encrypted, forwarded through the tunnel and introduced to the remote network. Because the VPN is working on top of the already available network, there is no need for extending or changing the physical network configuration. It also offers private connections over the Internet without expensive service contracts with the ISP for private leased lines between the sites.

Instead a VPN is typically established with software in the VPN hosts. The software is either implemented as part of the operating system or running as an application in the operating system. There are many open source application layer solutions available, e.g OpenVPN, OpenSSH and CIPE. Also, IPsec is supported by many operating systems and is working on the network layer. The cost-efficiency combined with the high confidentiality have led to an increased popularity of VPNs, especially in the corporate sector.

One of the open source VPN solutions is OpenSSH, which is an implementation of the SSH protocol. The SSH protocol [1] uses the concept of encrypted logical channels between a server and a client. SSH consistently relies on TCP for transporting encrypted data to the other side.

In version 4.3, OpenSSH introduced a new VPN feature to the implementation [2]. The VPN feature makes it possible to forward IP traffic between the server and the client, encapsulated in SSH packets. After introducing the VPN feature, the OpenSSH implementation still uses TCP for all its traffic. Because of this, the OpenSSH solution uses TCP for the base connection of any VPN it is creating. The encrypted VPN traffic will therefore be encapsulated into TCP segments. If the IP traffic that SSH is forwarding contains TCP segments, it will result in tunnelling TCP segments in other TCP segments, i.e. TCP in TCP [3] also called TCP stacking [4].

The TCP protocol has extensive algorithms to ensure correct delivery of the data. Having two TCP connections stacked together will thus force the algorithms of both TCP connections to work in parallel. TCP was not designed to work this way and problems are likely to occur in different situations.

The retransmission problems, TCP meltdown and double retransmit, are problems caused by tunneling TCP in TCP. The problems can occur when both of the stacked connections are retransmitting packets. In previous work, related to TCP in TCP tunneling, it is not entirely clear, how severe the retransmission problems really are.

In [3], TCP meltdown is claimed to be a realistic problem and TCP in TCP should therefore be avoided. In [5] and [6] testing in physical systems has been conducted to see the effects on performance when tunneling TCP in TCP. In [5] the TCP meltdown is showed to not be a realistic problem in a real network with packet loss less than 10% and the loss in performance, goodput, is not noticeable.

In [6] it is showed that using a TCP in TCP tunnel, in normal cases, decreases the goodput slightly. On the other hand, the authors have showed that, when having long propagation delays in the network, the TCP tunnel actually increases the performance in comparison to not having the tunnel at all.

In [4] a large test is conducted using several available VPN solutions. The solutions are divided into a TCP and a UDP group depending on which protocol is used for the base connection. It is then confirmed that having multiple timers along the data packet path has the most profound diminishing effects on the data bandwidth. According to [4] the UDP based VPNs have 80 percent higher bandwidth utilization then the ones using TCP.

The fact that OpenSSH uses TCP as base connection for its VPN feature seems, according to some of the previous work, to be a problem. This is the main reason for starting this project.

1.2 Proposed solution

The problem caused by unnecessary retransmissions is indeed a problem but it also lightens a potential solution to the problem. Because the forwarded protocols, of course, cannot be changed, it falls to the VPN connection to be modified. For example by skipping mechanisms such as retransmissions and let the forwarded TCP connection take care of this, which it already does. This is also the approach of the existing VPN solutions that uses UDP for the VPN base connection.

One specialized VPN product is OpenVPN [7]. It can use either UDP or TCP in the base connection for the tunnel. OpenVPN is as the name imply open source and the payload data encryption and decryption is done with help of OpenSSL. Various cipher algorithms are supported, e.g. AES-128, and the user is encouraged to use the CBC cipher mode. Because of the usage of OpenSSL libraries, which also is extensively used by OpenSSH, the OpenVPN implementation have a few things in common with the OpenSSH implementation. It has therefore been useful to look at the OpenVPN implementation in order to find advice in how to proceed with the modification of OpenSSH.

1.3 Objectives

The main objectives of this project are the following:

- Create a modified version of OpenSSH v5.4 that allows a VPN to be established using a UDP base connection.
- Test the modified version, in a test network, and compare the performance to the original. The performance should be measured in terms of bandwidth during different rates of emulated packet loss.

1.4 Scope

A usable modification of the OpenSSH v5.4 is presented. The implementation have two modes and can use either a TCP base connection (STD) or a UDP base connection (UVPN). The modification can use CBC as cipher mode for the encryption. Support for the CTR mode, in the UVPN mode, is not in the scope of the project. Also the support for sequence numbers when using the UDP base connection is moved out of the scope because of time limitations.

The original and modified implementation are tested in terms of bandwidth during different rates of emulated packet loss. In each test, the packet loss was only unidirectional, but different tests had packet loss emulated in different directions. Tests where performed by dropping packets containing data as well as ACKs from the forwarded connection. Testing in more complex network scenarios such as delays, congestion or induced TCP meltdown are out of the scope of this project.

1.5 Related work

Here follows a summary of recent work related to this project.

"Why TCP Over TCP Is A Bad Idea" [3]

Here an explanation of the phenomenon TCP meltdown, caused by TCP in TCP, is given. The author states that VPNs using TCP as base connection should be avoided. The author also states that, from practical experience, the SSH implementation used as a VPN solution have showed to be fairly unusable. The problem of TCP in TCP is the author's incentive to start the CIPE project, also referred to in this report.

"Understanding TCP over TCP: Effects of TCP Tunneling on End-to-End Throughput and Latency" [6]

The authors of this paper present an investigation of the effects a TCP tunnel and the TCP parameter configuration has on the end-to-end performance. The performance is measured in terms of goodput between the two end hosts of a tunnel.

Under normal conditions a TCP tunnel is showed to degrade the end-to-end goodput. On the other hand, if the network propagation delay was large, a tunnel improved the goodput in comparison to not using a tunnel.

They have also investigated the SACK options effect on the end-to-end goodput and states that it solves the problem by increasing the goodput. They state that, when using tunnels, the socket buffer size in both the end hosts and tunnel connection have impact on the goodput. Using a small buffer size will lead to decreased end-to-end goodput.

"Tunneling TCP over TCP" [8]
Here the concept of TCP meltdown is explained. The authors explain how problems of tunneling TCP in TCP can be caused by VPN systems that use a Virtual Ethernet Interface, VEI. A theoretical analysis is done where the scenario causing TCP meltdown is given. In the analysis they start by analyzing the TCP meltdown scenario when both the tunnel connection and the forwarded connection use the same TCP implementation, namely the Reno alt. TAHOE implementation, for both connections.

As possible solutions to the TCP meltdown problem three alternatives are given. One is using UDP for the base connection. The other is to use the TAHOE implementation of TCP for the upper layer, the forwarded connection, and using the Reno implementation for the base connection. Last, they have presented a solution that involves the VEI to selectively discard unnecessary retransmissions arriving from the forwarded connection. Also, a practical test is done on the performance. In this test a comparison is done between using Reno for both connections, using TAHOE for both connection and to using the proposed mix of the two implementations.

As a result of the work the authors state that TCP in TCP is not as bad as it has been presented to be. Also it is mentioned that using UDP as the base connection may cause new firewall problems.

"Tunneling TCP over TCP – A study of a real system" [5]
The work focuses on the effects of TCP in TCP in VPNs. The aim was to test the impact on performance by using a Virtual Ethernet Interface, VEI, to tunnel a TCP connection through another TCP connection. The authors wanted to test the theoretical claims, that problems such as TCP meltdown will occur, causing decreased goodput.

Questions raised by the authors was:

- Is TCP in TCP an actual problem which would confirm the theoretical claims?
- If it is a problem, when do the problems occur?
- Is it realistic that the levels of packet loss will occur, which is needed for triggering the problem?

To investigate, a test system was created. The test system consisted of three computers, two of the computers where end hosts and the third computer acted as a forwarding router between the end hosts. The middle computer had the purpose of emulating packet loss that can occur on a real, bigger, network [5].

In the tests the emulator *Linux Traffic control Network Emulator*, tc Netem, was used. In the tests only unidirectional packet loss was emulated. Ethereal was used to monitor the traffic. The tests where performed using SSH as VPN solution. Two different VPN configurations where tested. One was created using the port forwarding feature of SSH. The other by using the IP forwarding feature.

The results of the simulations performed by the authors have showed that the TCP meltdown does not occur for realistic levels of packet loss, i.e. lower than 10% [5]. The decrease in bandwidth is only noticeable when the packet loss consists of ACKs and is approximately unchanged for packet loss of packets containing payload data [5].

"Virtual Private Networks: An Overview with Performance Evaluation" [4]
The authors have chosen some of the most popular open source Linux based VPN, solutions currently available. The solutions are divided into the groups, GRP_TCP and GRP_UDP depending on the protocol used for the base connections.

The results presented from the study are comprehensive. They not only investigate many different network performance metrics, but also investigate the available features and flexibility of the solutions.

The testing is done in a test bench of two Linux computers. In the report there is no mentioning of any packet loss emulation, thus it has to be assumed that no network disturbances, e.g. packet loss or congestion, occurred during the testing. The results show that using UDP is much more profitable, in terms of bandwidth utilization, latency and overhead.

"CIPE Project" [9]
The CIPE project is a developing project started by the author of [1]. It is aimed to create a lightweight VPN solution using UDP as base connection [9]. It re-implements the cipher algorithms and is thus not relying on any other existing encryption software. Supported ciphers are IDEA with a key length of 128 bits and Blowfish with 64-bit block size in CBC mode [9].

Also here, each packet is encrypted and decrypted independently, where the first block of each packet contains an IV. CIPE has been developed open source and documented which is believed to the most secure approach in crypto protocol development [9]. There are two ways of attack in CIPE one is attacks against the design of a system and another one, attacks against implementation errors. CIPE is given the bit better performance then IPSEC.

Chapter 2, Theory

This chapter presents some theory needed for the following parts of the report.

2.1 VPN

A Virtual Private Network, VPN, is a logical network connecting hosts in an underlying physical network. The VPN traffic is separated from any other network traffic by using encrypted tunnels between the VPN hosts. Typically in such a tunnel, the forwarded traffic is encapsulated into a solution-dependent packet format on which a block cipher is used to encrypt the traffic. As mentioned in the introduction, a VPN can work on different layers of the OSI model.

Figure 1 describes an example where a VPN client is connected to a VPN server which is part of a private network. The VPN client can forward traffic to the internal network behind the VPN server and vice versa.

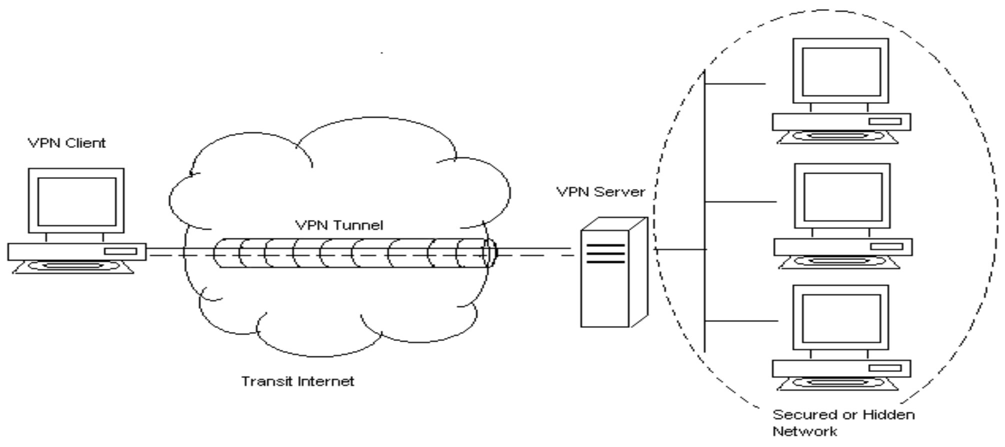


***Figure 1**: A VPN connection going over the Internet.*

Three common types of VPNs are Application Layer VPNs, Network Layer VPNs and Data-link Layer VPNs. Here follows a few characteristics of these types.

2.1.1 Application Layer VPNs

This type is working on the Application layer of the OSI model. Examples of solutions that are Application Layer VPNs are SSH, SSL, OpenVPN. Typically, the tunnelled traffic is encapsulated into application specific headers and then sent to the other side using the available Transport Layer Protocol, such as UDP or TCP.

2.1.2 Network Layer VPNs

This type of VPN is working on the Network layer of the OSI model. An example of Network Layer VPN is IPsec. The tunnelling is transparent to the applications working on the higher layers. Support for Network Layer tunnelling is implemented by the operating system of the host that is an end point of the tunnel.

2.1.3 Data-link Layer VPNs

Data-link layer VPNs are working on the Data-link Layer of the OSI model. They are most commonly used on top of PPP in order to secure modem based connections, although PPP actually encrypts the traffic. The most popular encapsulation protocols used for Data Link layer VPNs are PPTP, L2TP and L2F

2.2 Different VPN solutions

In this chapter, the existing VPN solutions OpenSSH, TLS, IPsec and OpenVPN are presented which are related to this project.

2.2.1 OpenSSH

OpenSSH is an implementation of the SSH protocol. SSH can be used to login securely to remote hosts using connections going over an insecure network. There are many more features that SSH can offer, for example TCP forwarding and IP forwarding.

The SSH protocol consists of three major parts [1], the Transport Layer Protocol [10], the User Authentication Protocol and the Connection Protocol [11]. The transport Layer Protocol maintains an encrypted connection, a tunnel, between the server side and the client side. It also handles server authentication. The User Authentication Protocol handles client authentication. The Connection protocol multiplexes the data for different logical bidirectional channels going between the two sides.

Logical channels are dedicated for different services used over a connection. For example, when forwarding TCP connections or IP traffic, there are two logical channels for the connection. One for the interactive terminal session and protocol messages and one for the tunnel forwarding the packets. All the data assigned to the logical channels are multiplexed through the single encrypted connection between the two sides, which is maintained by the SSH Transport Layer Protocol

The SSH Connection Protocol is using the SSH Transport Layer Protocol to encapsulate the channel data into SSH packets. Following is a visualization of an SSH packet encapsulating channel data.

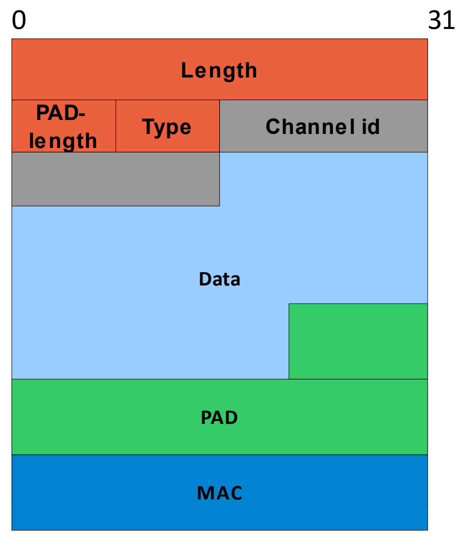


***Figure 2**: SSH packet used to send data for a logical channel.*

The SSH Transport Layer Protocol allows the use of many different symmetric key encryption algorithms. As stated in the standard [10], all implementations of SSH are required to support at least 3DES in CBC mode and recommended to support AES128 in CBC mode. The session keys are derived from a shared secret and an exchange hash resulting from a Diffie Hellman key exchange.

The standard also states that SSH works over any 8-bit clean binary transparent transport and that the underlying transport protocol should protect against transmissions errors, as such errors cause the SSH connection to terminate. The protocol assumes reliable transport of the data and it does not reinitialize the cipher on a per packet basis. The cipher is only reinitialized after a key exchange. Note: This is very important to consider since it does not tolerate any disturbance in the base connection.

The OpenSSH implementation uses TCP for all its transport. As of version 4.3 it supports VPN tunnelling where a tunnel can be established to tunnel either Layer 2 (Ethernet) or Layer 3 (IP) traffic. For example, two security gateways can be connected by an SSH tunnel, tunnelling IP traffic. IP datagrams routed to one of the gateways are then forwarded through the tunnel to later be routed to the final destination.

In order to setup a tunnel using OpenSSH, the client must first authenticate itself in the same way as when creating a regular SSH session. Upon successful authentication the tunnel will be created using a virtual network interface, a tun-device, on each side. Everything that is routed to the virtual interface on the host is then encapsulated in SSH packets, encrypted, possibly compressed and sent to the other side of the tunnel. Figure 3 shows a tunnelled IP datagram inside an SSH packet.

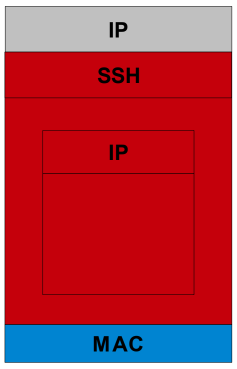


***Figure 3**: A tunnelled IP packet in an SSH packet. Red denoting encrypted information.*

The symmetric encryption of the data is done by libraries supplied by OpenSSL. The algorithms supported by OpenSSH are AES, 3DES, Blowfish, Arcfour. The cipher modes supported are CBC and CTR.

2.2.2 Internet Protocol Security (IPsec)

IPsec is a suite of protocols that provides security at the network layer [12]. It offers protocols for confidentiality and integrity for the individual datagrams. In order to send secure datagrams, a unidirectional logical channel, a security association (SA), is created for the given direction.

An SA is created and managed by using either, the Internet Key Exchange (IKE) algorithm which is the default key exchange, or by using the Internet Security Association and Key Management Protocol (ISKMP). If both sides want to send data two SAs with opposite directions must be used.

IPsec includes the Authentication Header (AH) protocol and the Encapsulation Security Payload (ESP) protocols for the transportation of datagrams. Either the AH protocol or the ESP protocol can be used in the transportation.

The AH protocol offers integrity for the datagrams by using a message digest derived from the IP payload, including the AH header, and portions of the IP header. It does not encrypt the payload and thus does not support confidentiality for the datagram. The ESP protocol offers both integrity and confidentiality. The integrity offered is somewhat more limited than for the AH, since nothing from the IP header is included when calculating the message digest.

An SA can only use one of AH or ESP, but by using two SAs in the given direction, both AH and ESP can be applied to the stream[13]. When using ESP, the Payload data and the ESP trailer, containing information about the transport protocol, is encrypted.

ESP is designed for symmetric key encryption. The encryption algorithm may use a cipher mode that needs synchronisation, e.g. if CBC mode is used the algorithm needs to be synchronized with an IV.

In order for ESP to tolerate packet loss or out-of-order arrival of packets, synchronization must be done individually for each packet, i.e. the synchronization data must be carried in each packet[14]. If CBC is used, it means that an explicit IV must be prepended the payload data.

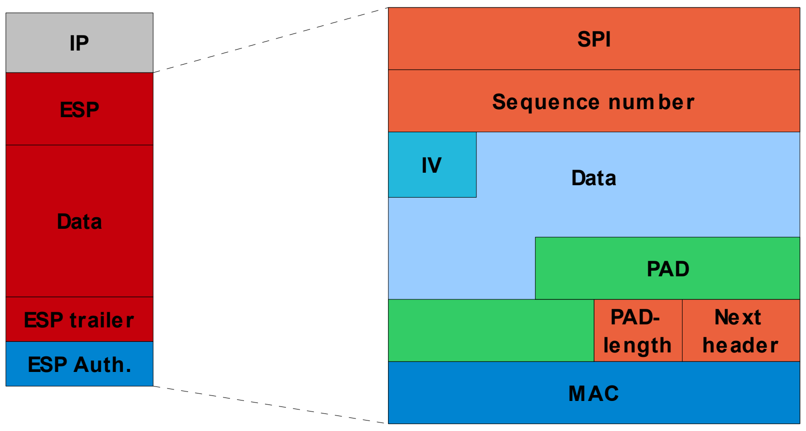


Figure 4: *To the left, an IP datagram containing an ESP header. To the right, the fields in an ESP header including an IV for the cipher.*

Tunnel mode

There are two types of SAs defined in IPsec. One type used for the transport mode and the other for the tunnel mode. Both types can be used with either the AH or ESP protocol. IP datagrams that are to be tunnelled are then encapsulated within the data field of an AH or ESP header in an outer IP datagram. If ESP is used the whole tunnelled IP datagram will be encrypted. The following figure visualizes a tunnelled IP packet encapsulated in an ESP header inside an outer IP datagram.

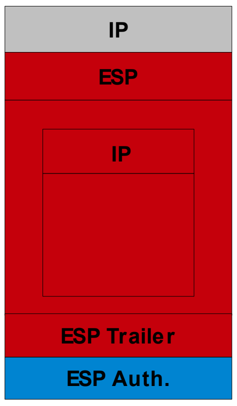


Figure 5: A tunnelled IP packet when using a tunnel mode SA and ESP.

2.2.3 TLS and OpenSSL

The Transport Layer Security (TLS) protocol is a standardized extension of the Secure Socket Layer (SSL). It is designed to achieve privacy and data integrity between two communicating applications [15]. The TLS protocol includes the TLS Record protocol which is designed to be used on top of a reliable transport layer protocol, such TCP. It takes care of the transportation of messages to the other side. This includes, fragmenting, compressing, encrypting and sending the TLS messages, as well as the corresponding processes when working on the receiving side.

A secure connection is initialized with the TLS handshake protocol, which works on top of the TLS Record Protocol. This protocol supports secure and reliable authentication of the peers identity and negotiation of shared secret for creating symmetric session keys. The protocol uses public key encryption, such as RSA or DSA etc.[15]. When a connection is established, data will be encrypted with the chosen symmetric key encryption algorithm. All block cipher encryption is done in CBC mode[15]. As of version 1.2 an unpredictable explicit IV is also created for each packet [15] in order to add security against certain chosen ciphertext attacks.

OpenSSL is an open source implementation of the SSL and TLS protocols and it also supplies a general purpose cryptographic library. For example, it supports generation of cryptographically secure pseudo random numbers and several symmetric key encryption algorithms.

2.2.4 OpenVPN

OpenVPN is a specialized product for establishing VPN tunnels. OpenVPN can be used with either TCP or UDP as base connection.

There are two modes for authentication in OpenVPN. One is Static key mode and the other is TLS mode. In Static Key mode, a pre-shared key is derived and shared between the two end hosts of the tunnel, before the tunnel is started [7]. The key contains four independent key parts. In TLS mode, an SSL session is established by using certificates from both sides to achieve bidirectional authentication [7]. The session is then used to exchange random key material to create symmetric keys to be used for the encryption of tunnelled packets.

The TLS authentication requires a reliable transport layer protocol when communicating between the two sides. When OpenVPN is used with a UDP base connection, a reliable layer is added on top of the UDP connection, in order to support reliable transport for the authentication messages. The reliable layer is not used for tunnelling the forwarded network traffic. The OpenSSL library is used for symmetric key encryption of the VPN packets. The following figure visualizes the structure of an OpenVPN packet.

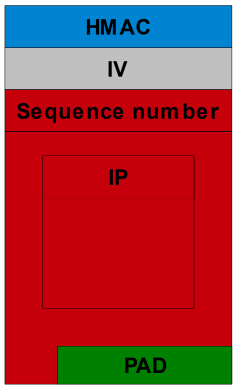


Figure 6*: An OpenVPN packet containing a tunnelled IP packet.*

In OpenVPN each packet has its own randomly generated IV and its own padding. After the IV follows the so called encrypted envelope. The encrypted envelope consists of a 64 Bit sequence number and the payload, which is an IP packet if IP traffic is being tunnelled. After the data follows a PAD. Both the encryption and PAD is done with help of the OpenSSL libraries.

2.2.5 CIPE

In the CIPE project, a VPN solution is implemented using UDP as a base connection. The block ciphers supported by CIPE are IDEA with the key length of 128 bits and Blowfish with 64 bit blocks size in CBC mode[9]. The cipher is synchronized by explicitly appending an IV to each packet. The cipher algorithms supported by CIPE where specially implemented in the CIPE project. The potential risks of reimplementing such algorithms have been compensated by keeping the development completely open to public review.

2.2.6 Relation to this project

The operation of each of the given solutions are very similar to each other. IPsec, OpenVPN and CIPE can send the encapsulated packets without a reliable transport protocol to the other side. TLS is able to tunnel IP traffic and is extended, in version 1.2, to use explicit IVs in order to add security. Also the OpenSSL libraries are extensively used by both OpenSSH and OpenVPN. Therefore, IPsec, OpenVPN, CIPE and TLS have similarities to OpenSSH and the modified version of OpenSSH. All the solutions, excluding OpenSSH, are using explicit IVs in the VPN packet format.

2.3 TCP as base connection

As mentioned in the introduction, using a TCP base connection in a VPN results in TCP stacking when a TCP connection is being tunnelled. Different sources say that TCP stacking may cause conflicts between the protocol mechanisms in the base connection and the forwarded connection and the result may be something often referred to as TCP meltdown. As explained in [3], TCP meltdown can occur when the forwarded connection, inside the VPN, have shorter RTO then the base connection.

For example, this could occur if multiple connections are tunnelled over the base connection. Assume that the forwarded connection *f* has been able to send data undisturbed over the base connection and is then quiet while other connections are sending data. Before *f* continues to send data a disturbance, e.g. packet loss, has occurred causing the base connection to trigger congestion control. The forwarded connection *f* will thus have a larger sender window and shorter RTO than the base connection.

If now a packet is lost again by the base connection, *f* will stall and retransmit its entire retransmit queue before the base connection has even retransmitted its lost packet. The forwarded connection *f* will thus build up lots of unnecessary retransmissions over the base connection which will further stall the data stream of *f*. The data stream of *f* will thus be completely stalled until the base connection has managed to empty the retransmission queue. This may take some time depending on how big the difference of RTO and the sender window size was. If the RTO timers have the same value of the two connections a lost segment in the base connection will only cause one unnecessary retransmission in the forwarded connection. This is called double retransmit and is less severe than TCP meltdown.

2.4 Block ciphers

Block ciphers are normally used for encryption of payload in an encrypted connection. This is because payload encryption needs to be done fast over a large quantity of data. Public key encryption is good for handling keys and signatures but it is generally slower than symmetric key encryption. Therefore public key encryption is normally used for initiation purposes , e.g. key exchange or authentication, whereas Block ciphers are used for data encryption. OpenSSH supports the following Block ciphers [1].

- Blowfish - block sizes
- 3DES - block sizes
- AES - 128-, 192-, 256-bits block sizes
- Arcfour (RC4)

Cipher modes

The Electronic Code Book (ECB) mode is the most straight forward way of using a Block cipher. Each block of data is in this mode encrypted individually [16]. The vulnerability in this, is that if an attacker for some time has access to both plaintext and the corresponding ciphertext, it is possible to create a list of ciphertexts and corresponding plaintexts. If enough {plaintext, ciphertext} pairs are found, a future encryption with the cipher, assuming the keys are not changed, can then be translated (decrypted) [16]. In order to avoid this confidentiality-weakness, more complex cipher modes can be used. In SSH the supported cipher modes are CBC or CTR [17].

CBC mode

The Cipher Block Chaining (CBC) mode uses feedback from the previously encrypted cipher block when encrypting a block. An encrypted block C does thus not only depend on the key k, but also on the data in the previous block. Because of this, no {plaintext, ciphertext} pairs found can be used to translate any future cipher block.

The encryption and decryption in CBC mode is done according to the following equations.

Encryption **Decryption**

$$\begin{cases} C_1 = E_k(IV \oplus P_1) \\ C_i = E_k(C_{i-1} \oplus P_i) \end{cases} \qquad \begin{cases} P_1 = IV \oplus D_k(C_1) \\ P_i = C_{i-1} \oplus D_k(C_i) \end{cases}$$ [16] (E1, E2)

An Initialization Vector (IV) is used as the first block in the data stream and the cipher stream. Because of the IV, all the real blocks in the stream can be processed in exactly the same way and the IV will synchronise the encryption and decryption.

During the encryption and decryption, the process depends on everything that have been previously processed since the IV was specified and the cipher was initialized. The IV and the previously processed blocks, make up what in this thesis is refereed to as crypto-state. If a block in the cipher was to be manipulated or removed, the dependencies in the cipher will cause the decryption to fail, starting from the position of that block. One can say that any alteration of the cipher will distort the crypto-state starting from the alteration.

CTR Mode

In Counter (CTR) mode, a counter is used which is incremented as the cipher is processed. The counter X is stored in a variable of equal size as the block size of the cipher, e.g. a block size of 128 or 256 bits. The data stream is split up into pieces of 8 bits. In order to encrypt a piece P of data, the counter is first incremented and then encrypted by the underlying cipher algorithm.

The CTR mode is working as a stream cipher by XORing the first 8 bits O of the output from the encryption algorithm with the 8 bits of data. For the encryption of the next piece of data the the above procedure is repeated. Exactly the same procedure is done for the decryption but on the cipher text. The general procedure is explained by the following equations.

Encryption **Decryption**

$$\begin{array}{ll} (1):\ X_j = X_{j-1} + 1 & (1):\ X_j = X_{j-1} + 1 \\ (2):\ O_j = L_8(E_k(X_j)) & (2):\ O_j = L_8(E_k(X_j)) \\ (3):\ C_j = P_j \oplus O_j & (3):\ P_j = C_j \oplus O_j \end{array}$$ [16] (E3,E4)

In the encryption and decryption, there are no dependencies of the previous data in the cipher. Therefore, it is possible to use parallelism in the encryption and decryption. This is not possible in CBC mode when the blocks must be processed in order.

2.5 UDP as base connection

By using UDP for the VPN base connection, the retransmission problems double retransmit and TCP meltdown will no longer be present. A forwarded connection will not be TCP in TCP, but instead TCP in UDP. UDP does not support any retransmission mechanisms, so it will only be the forwarded TCP connection that will do retransmissions. Another gain from using UDP, is the smaller header size. UDP has a header size of 8 bytes, when TCP have a header size of 20 bytes. This will leave space for bandwidth improvement in the tunnel.

The benefits from using UDP as base connection seams to be great, but they do not come for free. UDP does not support reliable transport, as TCP does. This means, that UDP packets can be lost or arrive out-of-order at the receiver. By using UDP there is thus a great loss of reliability in the base connection.

In SSH, the encrypting and decrypting of the SSH packets is done as if they where a stream of data. The crypto-state is kept at the end hosts and will not be reinitialized before the rekeying process is triggered and new session keys are derived. Either the CBC mode or the CTR mode is used in the cipher process. The encryption and decryption of the packets is thus depending on all the previously processed packets in the packet stream.

The cipher process is relying on the base connection to deliver the data to the receiver exactly as they where sent. If the TCP base connection of OpenSSH should be just replaced with UDP without any further precautions, it would mean that the slightest disturbance in the UDP connection would distort the receivers crypto-state. All the decrypted packets after the disturbance would then be complete nonsense, and the VPN connection would be disabled.

OpenVPN has dealt with this problem by adding an IV to each encrypted packet [7]. If the VPN cipher is used in CBC mode the IV can be used to re-initialize the crypto-state for each packet. The dependence of any previous packet would then be removed. If a packet would be lost or arrive out-of-order it would not distort encryption or decryption of following packets. This way of solving the problem have also been adapted in this project.

OpenVPN also supports a reliable transport layer on top of the UDP connection to enforce reliability for the authentication process [7]. This is not done in this project. Instead the original TCP connection is preserved in parallel with the additional UDP connection. Our choice to use two parallel connection has later showed to give rise to certain problems, see chapter 4.3. Future work may therefore be to modify the implementation to use a similar solution as used in OpenVPN.

In the modified OpenSSH implementation, the IV is generated with the default PRNG supplied by OpenSSL, i.e. the `RAND_bytes(...)` function. It is then prepended the encrypted SSH packet. The IV is of equal length as the original cipher that is used by the original implementation and thereby ranging between the length of 8 to 64 bytes [1]. The modification is done to support CBC as cipher mode for the VPN connection. The CTR mode is not covered in this project and thus not supported by the modification. Possibly, a similar solution could be used for the CTR mode, but this is left for future work.

Probability of duplicated IVs

When explicitly prepending the IV to every SSH packet, a potential security hazard is that the same IV is used twice or more using the same keys. In the original OpenSSH implementation, two limits are defined for how much data can be encrypted with the same key set. One limit defines the number of blocks that can be encrypted and the other limit defines the number of SSH packets that can be encrypted. The limits are derived according to the following [17].

$$\text{max\_blocks} = \begin{Bmatrix} \textit{if} \quad \text{block size} \geq 16\,\textit{bytes} & : & 2^{\text{block size}\cdot 2} \\ \textit{else} & : & 2^{30}/\text{block size} \end{Bmatrix} \tag{E5}$$

$$\text{max packets} = 2^{31} \tag{E6}$$

Also a re-key limit can be set which can further limit the amount of blocks allowed.

$$\text{max\_block} = \mathit{min}(\ \text{rekey\_limit}\ ,\ \text{max\_blocks}\ ) \tag{E7}$$

The first limit to be reached, of max_blocks or max_packets, will dictate when the key set must be renewed.

The probability of having multiple IVs between rekeying can be calculated according to the birthday problem. When picking *n* elements from a set containing *H* different types of elements, the probability *p*(*n;H*) of having two or more elements of the same type can be calculated as follows [16]:

$$\begin{aligned} p(n;H) &= 1 - \left((1-\frac{0}{H})\cdot(1-\frac{1}{H})(1-\frac{2}{H})\cdot\ldots\cdot(1-\frac{n-2}{H})\cdot(1-\frac{n-1}{H}) \right) \\ &= 1 - \frac{1}{H^n}\prod_{i=0}^{n-1}(H-i) = 1 - \frac{H!}{(H-n)!\,H^n} \end{aligned} \tag{E8}$$

The above formula can be approximated to [16]:

$$p(n;H) \approx 1 - \frac{1}{e^{(n^2/2\mathrm{H})}} \tag{E9}$$

In a VPN connection, IP packets will be forwarded, one in each SSH packet. The encrypted part in each SSH packet will therefore be of size [*SSH header length*]+*MTU*+[*PAD length*], when excluding the MAC which is not encrypted. The SSH header length is 6 bytes and MTU is the MTU value for the tun-device. The size of encrypted data is approximated to the MTU value in the following calculations. This will only make the resulting probability value more pessimistic, which may not be bad when estimating security.

The maximum number of packets allowed by the limits can be approximated to *n* according to E10. *H* in E11 is the possible combinations of IVs.

$$\begin{aligned} \text{implicit max packet limit} &\leq \mathit{min}\left(2^{31}, \frac{\text{max\_blocks} \cdot \text{block\_size}}{\text{MTU}} \right) \\ &\leq \frac{\text{max\_blocks} \cdot \text{block\_size}}{\text{MTU}} = n \end{aligned} \tag{E10}$$

$$H = 2^{8\text{block-size}} \tag{E11}$$

CASE: block size < 16 bytes

The exponent in equation E9 will in this case be according to E12.

$$n^2/2\mathrm{H} < \frac{2^{30}}{\mathrm{MTU}\cdot 2^{(8\mathrm{block\_size}+1)}} \tag{E12}$$

Increasing the block size will decrease E12 and thus decrease the probability of duplicated IVs. In the following graph the probability is showed for different MTU values.

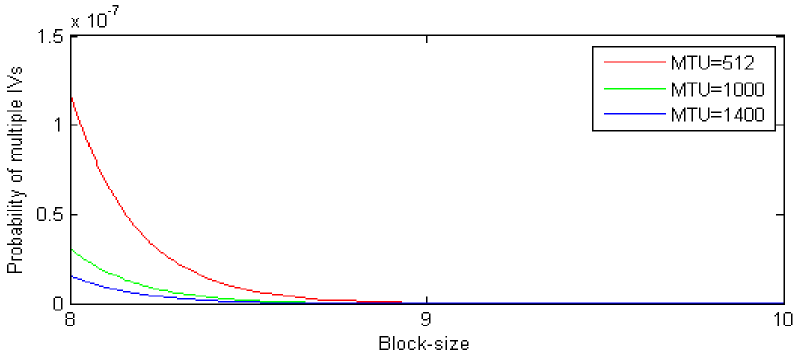


***Figure** 7: Graph of the probability vs. block size for duplicated IVs between rekeying. The block size ranging between 8 to10 bytes and MTU of 512, 1000 and 1400 bytes. Note, block sizes between 8 and 16 bytes are just theoretical and are not supported by the implementation.*

CASE: block size ≥ 16 bytes

In this case the exponent will be according to the following.

$$n^2/2\mathrm{H} < \frac{2^{(2\,\mathrm{block\_size})}\mathrm{block\_size}}{\mathrm{MTU}\cdot 2^{(8\mathrm{block\_size}+1)}} = \frac{1}{\mathrm{MTU}\cdot 2^{(6\,\mathrm{block\_size}-\ln(\mathrm{block\_size})+1)}} \tag{E13}$$

Similar to previous case, increasing the block size or increasing the MTU will decreases E13 and thus decreases the probability of duplicated IVs. Calculating the probability for a block size of 16 bytes and MTU of 512 bytes results in a probability of approximately $0\cdot 10^{-13}$ for duplicate IVs. For a block size of at least 16 bytes and MTU of at least 512 bytes the probability for an IV collision is very low.

Chapter 3, Method

The two main steps in this project are 1) modifying OpenSSH and 2) testing the implementation.

3.1 Modifying OpenSSH

At first, a study of the OpenSSH implementation was done in order to get understanding of how the implementation works and to estimate the extent of needed alterations. Already available VPN solutions using UDP as base connection, e.g. OpenVPN, was investigated. By combining this knowledge, together with the theory in Chapter 2, it was possible to derive a strategy for the modification. The strategy was later implemented and is explained, in detail, in Chapter 4.

3.2 Testing the implementation

A test bench where constructed in order to create an environment in which variable packet loss could be emulated. The modified implementation was then compared to the original implementation in terms of bandwidth. The bandwidth was measured after creating a tunnel in the test system, using the implementation to be tested. Samples of the bandwidth where taken while varying the packet loss probability. Also the raw bandwidth of the test bench where measured for different packet loss probabilities. The tests where repeated for different MTU values of the network path. The results of the tests where summarized and a comparison could be made between the two implementations.

Chapter 4, Implementing the Modification

This chapter explains the altering process of the OpenSSH source code. The alteration was done with the aim for OpenSSH to support VPN using a UDP base connection. The first sub-chapters deal with the necessary details of how the original implementation works. Then the implementation of the modification is explained.

4.1 The original design of OpenSSH

Here follows a detailed explanation of the necessary parts of the original implementation, in order to later explain the modification, in chapter 4.2. Because OpenSSH relies on parallel child processes rather than threads for the connections, it is only necessary to consider one connection when analysing and modifying the given modules.

4.1.1 Important code modules

The most important modules, related to this project, are packet.c/.h, channels.c/.h, serverloop.c/.h, and clientloop.c/.h. These modules are used to create the data flow between the client and the server. The misc.c/.h module is also important because it contains the function for initializing a tun-device, virtual network interface, in the system.

A connection, between the server and the client, can be visualized with the modules shown in the following figure.

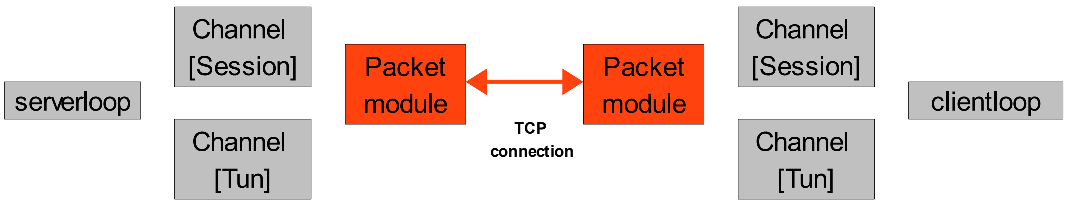


Figure 8*: The important modules involved in the communication between server and client. The packet module is in red to indicate that encryption is taking place and the connection to the client is thus encrypted when the VPN connection is initialized.*

- **packet.c/.h:** The packet module implements a big part of the SSH Transport Protocol. SSH packets can be created, encrypted, decrypted and parsed by the functions supplied in this module. The module has input and output buffers where the incoming SSH packets and the outgoing SSH packets are stored. The packets in these buffers are all encrypted.

  In order to be able to encrypt or decrypt the packets the packet module stores the session keys for the connection. Cipher contexts and above specified buffers are stored in the session_state struct defined in the packet module.

- **channels.c/.h:** The channel module implements a big part of the SSH Connection Protocol. It supplies methods for creating and using different types of logical channels.

For a connection, there might be more than one channel, depending on the requested service. Each channel is given a channel ID and the data associated with each channel is multiplexed through the encrypted connection, supplied by the packet module.

The channel module thus uses the packet module for encapsulating the channel data into SSH packets. The state of a channel is stored in the Channel struct. Variables stored in the state are for example channel ID, channel input and output buffers.

- **serverloop.c/.h:** The server loop contains the main loop, on the server side, for an established connection. For a VPN connection, server_loop2() is used, maintaining a connection by triggering methods in the different code modules and mastering the data flow of the application.
- **clientloop.c/.h:** The client loop is the clients counterpart to the server loop.
- **misc.h:** This module contains the tun_open(...) method, which is used to create a tunnel device in the underlying operating system.

4.1.2 Key exchange

The key exchange, kex, is a mechanism for establishing new session keys for a connection. It is used in the establishment of a new connection. It is also done for an already established connection if the limits are reached for how much data the keys are allowed to encrypt before they have to be changed.

The main module for the key exchange mechanism is kex.c. The key exchange is triggered by calling the kex_send_kexinit(...) function which sends a kexinit message. The SSH message type of the kexinit is SSH2_MSG_KEXINIT and it contains the preferred and supported algorithms to be used for key exchange, encryption, compression etc.

During the establishment of a new connection, the server and client sends their kexinit messages at the same time. In an already established connection, the rekeying process is triggered by the side that needs new keys. It enters rekeying mode and sends a kexinit message to the other side. Then, the other side will also enter its rekeying mode and answer with its own kexinit message.

In the kexinit, each side denotes the preferred (guessed) algorithm in each category [10]. Upon input of a kexinit message, the kex_input_kexinit is called by dispatch. Then, kex_kexinit_finnish(Kex *) is called. From this the configuration is chosen by kex_choose_conf(Kex *). After this the method corresponding to the chosen algorithm is used.

The OpenSSH implementation supports the following key exchange algorithms:

• Diffie Hellman group 1, sha1	(DH_GRP1_SHA1)
• Diffie Hellman group 14, sha1 ,	(DH_GRP14_SHA1)
• Diffie Hellman group exchange sha1 ,	(DH_GEX_SHA1)
• Diffie Hellman group exchange sha256,	(DH_GEX_SHA256)

The Kex data structure contains the private and public keys of the connection, as well as the session ID. It also contains fields for the new symmetric keys that are derived during the key exchange. The last field in the struct is a list of functions, where each function corresponds to a key exchange algorithm. Depending on the choice, the correct kex algorithm is called from the list by kex_kexinit_finnish(Kex *). When the algorithm has finished, the kex_finnish(Kex *) function is called. An SSH2_MSG_NEWKEYS message is sent to the other side, indicating that the keys for the given data direction is ready to be used.

4.1.3 Channel creation

When a VPN connection is created, two logical channels are used, one for the interactive session and one for the data tunnel. Throughout the following text, these channels will be referred to as the session channel and the tunnel channel. For the tunnel channel, the read and write file descriptors are set to the specified tun-device.

4.1.4 Forwarding IP packets

The forwarding procedure is done in steps where the IP packets traverse seven buffers on the way to the other side, starting with the input buffer of the tunnel channel in order to finally arrive on the output buffer of the tunnel channel on opposite side. Sender denotes the sending side of the tunnel and receiver denotes the receiving side of the tunnel.

1.	Channel module:	**input buffer**	(Sender)
2.	Packet module:	**outgoing_packet buffer**	(Sender)
3.	Packet module:	**output buffer**	(Sender)
4.	Socket		(Transmission)
5.	Packet module:	**input buffer**	(Receiver)
6.	Packet module:	**incoming_packet buffer**	(Receiver)
7.	Channel module:	**output buffer**	(Receiver)

Channels and datagram mode, Input Packets (Sender)

For the VPN tunnel, a channel of type SSH_CHANNEL_OPEN is used in datagram mode. In datagram mode, the boundaries for each IP packet are explicitly maintained and an input and an output filter are applied to the channel.

When IP traffic is tunneled over the channel, the input filter detects each IP packet and which version of IP it belongs to, i.e. version 4 or version 6. A 32-bit header is prepended to the IP packet indicating the IP version and the packet is placed in the channel input buffer. The packet is put onto the buffer, using the buffer_put_string(...) method. This method puts the packet, preceded by its length, onto the buffer in the format {len,data}. The actual format is then {len, {AF header, IP packet}}.

The channel module reads from the input buffer by using the buffer_get_string(...)**.** With this function, the data is fetched by first reading the length and then consuming the data. This way, the channel module can easily fetch one IP packet from the buffer. After putting the channel ID in the beginning of the SSH payload, the packet is put into a SSH packet by using the packet_put_string(...) supplied by the packet module. After this, the channel module calls the packet_send() function, which starts the encryption of the SSH packet. OpenSSH thus puts one IP packet in each SSH packet.

The Packet module, Creating SSH packet (Sender)

The packet module handles encryption, sending, receiving and decryption of SSH packets. It also supplies methods for reading and creating SSH packets.

An SSH packet is created in the outgoing_packet buffer. First, it must be initialized by packet_start(int type). This function will initialize a packet and put the type of the packet at the sixth byte of the packet. The first four bytes are reserved for the packet length field and the fifth byte is reserved for the PAD length field.

The data can be appended to the packet by using the packet_put_...(...) functions. For example packet_put_string(...) will put a chunk of data in the packet preceded with a data length field, similarly to buffer_put_string(...).

After the packet is created, packet_send() is called. If SSH v2 is supported, which it must be in the VPN case, packet_send2() will be called. If rekeying is in process the packet will be enqueued until the rekeying process is done, unless the packet is part of the rekeying process. Otherwise, packet_send2_wrapped() is called which encrypts the packet. The following figure visualizes the flow of data, SSH packets, between the session state variables.

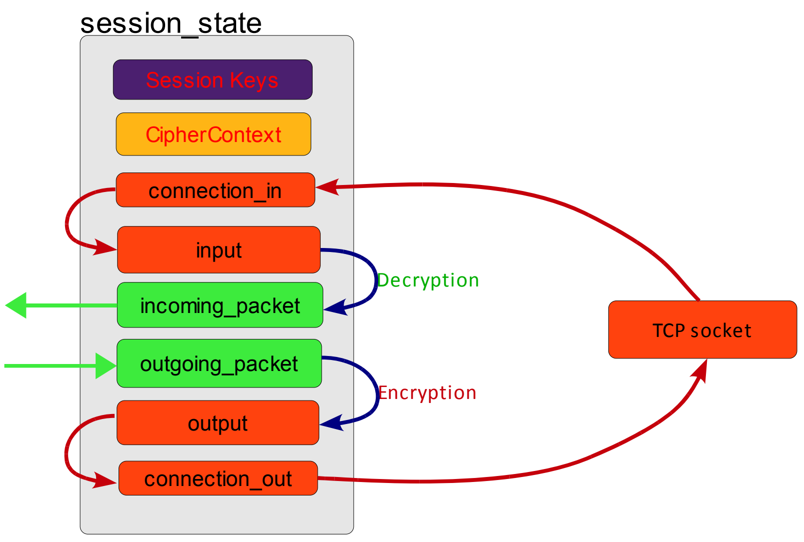


***Figure 9**: The session state data structure maintaining the state of the packet module. The arrows indicate the data flow, in the VPN case containing IP packets, between the objects. Green colour denotes unencrypted data, red denotes encrypted data and blue denotes data being encrypted or decrypted.*

If compression is used, the SSH packet residing in the outgoing_packet buffer, is compressed with the selected method. Before encryption is done a PAD is added to the packet so that the length is equal to an integer amount of cipher blocks. The PAD must be minimum 4 bytes and as much as 255 bytes [10].

Encryption is done over the SSH packet and the PAD and the encrypted packet is stored in the output buffer. The MAC is calculated on the plaintext packet and PAD, still residing in the outgoing_packet buffer. The MAC is appended to the encrypted packet in the output buffer [10] [17]. The following figure shows the layout of the SSH packet, PAD and MAC.

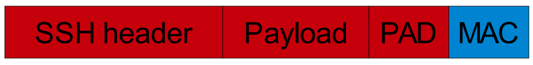


***Figure 10**:, An encrypted packet on the input and output buffer of the packet module. Red is indicating encrypted data and blue is indicating the MAC[10]*

In terms of crypto-state, the stream of packets is treated as a data stream. The cipher context is not re-initialized between encryption or decryption of individual packets, as mentioned in chapter 2.2.1.

Both CBC and CTR cipher modes are supported by OpenSSH. In each of the modes dependencies are thus created, either to the previously processed SSH packets or the amount of previously processed data. This is not a problem since transmission of the encrypted packets is reliable. TCP will ensure that the encrypted SSH packets arrive to the other side and that they arrive in order.

Transport using TCP

Because the original OpenSSH implementation uses TCP for transporting the encrypted data to the other side, it does not need to keep track of the boundaries of the SSH packets in the input and output buffers. The encrypted packets are fetched from the output buffer of the packet module as a continues stream of data.

The TCP protocol will then put the data into appropriate sized segments and send it over the connection to the other side of the connection. The TCP protocol takes the current MTU, on the network interface into account and no IP fragmentation will therefore occur. The receiving side reads the data on the socket as a stream and puts it onto the input buffer of the receiver packet module.

Packet module, reading SSH packet (Receiver)

From the input buffer, a packet is read by calling either packet_read_seqnr(...) or packet_read_poll_seqnr(...). The first function is typically used when waiting for a certain type of packet, for example in the rekeying process. Otherwise, when handling normal data packets, the second function will be used. This function then calls packet_read_poll2(...) . This function then decrypts the packet as it is fetched from the input buffer and puts it in the incoming_packet buffer.

The MAC is recomputed to check integrity and the PAD is removed. Also the fields for packet length, PAD length and type are removed and the function returns the type of the packet. If compression is used, the packet is decompressed.

Now, the data of the packet is ready to be read. In normal input packet processing, for example that of the a data channel, the packet_read_poll_seqnr(...) will be called from dispatch_run(...), which also calls the proper channel function to deal with the data of the packet. This will typically be repeated until all the packets in the input buffer of the packet module are processed.

Channels and Datagram mode, output packets (Receiver)

When dispatch_run(...) is called for an SSH packet containing regular channel data, it calls channel_input_data(). This function first fetches the channel ID to demultiplex the data to the correct channel. The data is then fetched to the output buffer of the correct channel.

The data is put onto the channel output buffer by using buffer_put_string(...). The IP packet that has been sent from the channel input buffer, of the sender side, has now arrived on the channel output buffer, on the receiver side. Still the boundaries of the IP packet is kept. The sys_tun_output filter is used to make sure that a complete IP packet has arrived before it is written to the tun-device. The AF header that was prepended by the input filter on the other side will be removed and each complete IP packet is sent to the tun device of the receiver host. The packet have now arrived on the other end of the tunnel.

4.2 The modification

In this chapter, a proposed and implemented modification of OpenSSH version 5.4p1 is presented. The chapter begins with defining the strategy. Then, the implementation of the modification is explained. The extended VPN functionality using UDP as base connection is hereafter denoted by UVPN.

4.2.1 Strategy

The main strategy can be divided into two points:

- When using the UVPN extension, there should be a TCP connection in parallel with the UDP connection. The TCP connection takes care of the standard SSH communication, hereafter denoted by STD, i.e. key exchanges, remote window adjustments, and the SSH session which also is present in parallel with the VPN tunnel. The UDP connection should be the base connection for the VPN tunnel.
- The dependencies between SSH packets, in CBC mode, should be removed by the use of an explicit randomized IV prepended to each SSH packet, similar to OpenVPN.

4.2.2 Modified modules

The modules that where modified are particularly packet.c/.h and channels.c. Also serverloop.c, clientloop.c, buffer.c and kex.c where slightly altered. Entirely new modules added are misc_uvpn.c/h supplying some miscellaneous methods and also packet_queue.c/.h supplying a queuing library for handling queues of packets.

4.2.3 Modifying packet.c/.h

The packet module contains the session keys, as well as the input and output buffers for the connection to the other side. In order to support UDP traffic, it was necessary to introduce a set of new or duplicated objects. Other than new file descriptors for the UDP socket, the choice of implementation led to the introduction of the following main objects:

- **Additional cipher contexts**: The explicit IV should be used to reinitialize the cipher contexts in order to remove the dependencies between the SSH packets. In order to do this, it is necessary to have a separate cipher contexts for the UVPN connection not to disturb the crypto-state of the STD connection.

- **Additional input buffer and output buffer**: Because the STD and UVPN packets are encrypted with different cipher contexts, it is necessary to store the encrypted SSH packets in different buffers. It is therefore necessary to have new input and output buffers for the UVPN connection.

- **Additional session keys**: The use of separate cipher contexts made it natural to also use separate key-sets for the connections. Using different keys for the UDP connection and the TCP connection can in this way offer more security compared to just copying the session keys.

In order to achieve this the original session_state struct was split up and extended. The additional data structures created are S_State, U_State and C_State. The C_State data structure contains the general communication variables which are needed for both connections. These variables are file descriptors for the socket, cipher contexts for sending and receiving, session keys etc:

```
typedef struct c_state{
      int initialized;

      int connection_in;
      int connection_out;

      CipherContext receive_context;
      CipherContext send_context;

      Newkeys *newkeys[MODE_MAX];

      struct packet_state p_read, p_send;
      u_int64_t max_blocks_in, max_blocks_out;
      u_int32_t rekey_limit;
      u_int64_t max_packets;
} C_State;
```

***Code 1**: Definition, C_State, general for STD and UVPN connection.*

The reason to create the general data structure C_State was to optimize the heavily used functions, for encryption and decryption of packets. Both communication types now each have a variable of identical structure. This results in less conditional execution in the functions, which is good for the performance of the application.

The S_State is a data structure dedicated for keeping the state of the STD connection and the U_State data structures keeps the state of the UVPN connection as shown below.

```
typedef struct std_state {
     int initialized;

     C_State com_state;
     Buffer input;
     Buffer output;

     u_int packet_discard;
     Mac *packet_discard_mac;
     u_char ssh1_key[SSH_SESSION_KEY_LENGTH];
     u_int ssh1_keylen;
} S_State;
```

Code 2*: Definition, S_State.*

Following is the definition of the U_State data structure.

```
typedef struct uvpn_state {
      int initialized;

     C_State com_state;

     PktQueue * input;
     PktQueue * output;
     PktQueue * pool[MODE_MAX];

     int mtu;
     int ready[MODE_MAX];
     struct sockaddr_in * local_host;
     struct sockaddr_in * remote_host;

} U_State;
```

Code 3*: Definition, U_State.*

There are some differences in the S_State and the U_State structures which made it necessary to create separate structures for the STD and UVPN communications. For example, the use of queues for the buffered VPN packets instead of buffers and the extra information needed, e.g MTU, end host addresses etc. As seen in the definition of U_State the UVPN connection has two pools of buffers, each being a queue. One pool is dedicated to the outgoing direction and the other for the incoming direction. Buffers can be fetched in order to place packets in either the input or the output queue. The differences aside, both the S_State and the U_State have a C_State variable which holds the general state variables.

The main state data structure, the session_state, now holds pointers to an S_State structure and a U_State structure, linking everything together. Following, is an excerpt from the session state definition.

```
struct session_state{
      S_State * std_state;
      U_State * uvpn_state;
            ...
      u_int max_packet_size;
      Buffer outgoing_packet;
      Buffer incoming_packet;
      Buffer compression_buffer;
            ...
      u_int packlen;
      int rekeying;
      int rekey_com;
            ...
      TAILQ_HEAD(, packet) outgoing;
};
```

***Code 4**: Partial definition, struct session_state.*

Putting it together, the following picture visualizes the organization of the different structures and buffers.

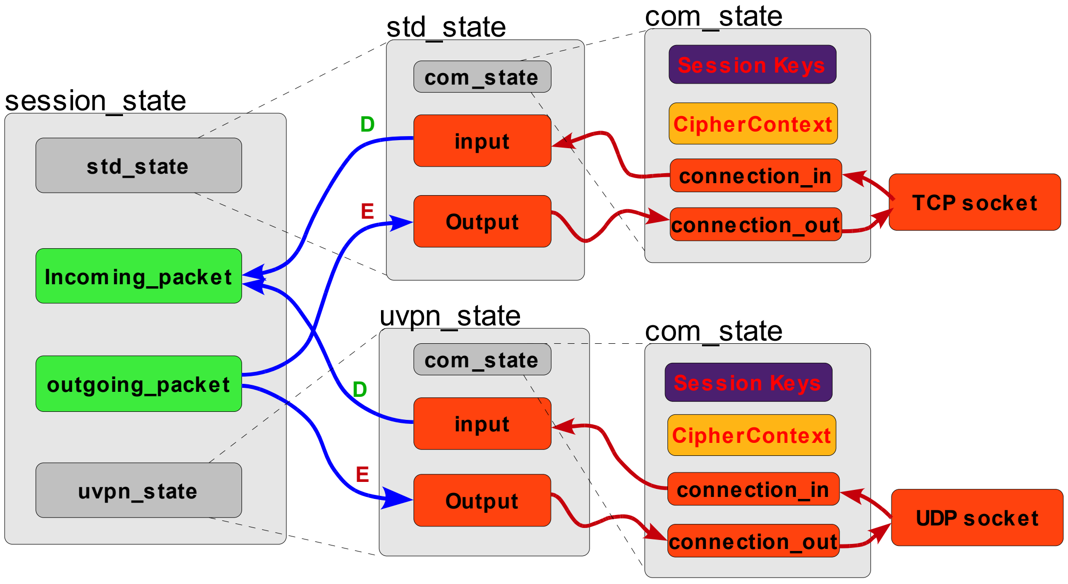


***Figure 11**: The organization of the new session state structure, located in packet.c, and its sub structures. Red color indicates encrypted data. Green indicates unencrypted data.*

The incoming_packet buffer and outgoing_packet buffer are now shared buffers by both the STD and UVPN communication. This was done to generalize the construction and decoding of

SSH packets from both connections. The creation and sending, reading and receiving is done by the same methods for both STD and UVPN connections. A new SSH packet type was introduced, which is SSH2_UVPN_DATA. When this type is used for a packet, set by packet_start(int type), the packet will automatically be sent over the UVPN connection, otherwise it will go over the STD connection. It was therefore no need to alter modules not using the UVPN connection and the alterations of the external modules where therefore kept to a minimum.

Encrypting packets

First, the packet is initialized by the calling module and data is appended to it. The SSH packet then resides in the outgoing_packet buffer. To encrypt the packet the packet_send() function is called which then calls packet_send2(). The type of the packet is then fetched. If rekeying is in the process and the packet is not part of the key exchange, the packet will be enqueued on the outgoing queue until the key exchange is done. Otherwise, packet_send2_wrapped() is called. This function then handles the encryption of the packet.

The packet_send2_wrapped() function is modified to take com_type as an argument. The com_type is a flag that indicates which of the connections the packet belongs to. If the type fetched by packet_send2() is SSH2_UVPN_DATA, COM_UVPN is given as argument, any other case COM_STD is given. The two communication types COM_STD and COM_UVPN are defined in packet.h.

When a UVPN packet is to be encrypted, a packet queue entry, QPkt, is fetched from the pool dedicated to the outgoing direction. A pointer, obuf, to the buffer of the QPkt is then set. In the case of STD, the same pointer is set to point to the output buffer in order to generalize the execution. The IV length is fetched from the cipher context. Then an IV is created by put_rand_iv(...) which uses RAND_bytes(...) supplied by OpenSSL. The IV is then added to the buffer, prepending the SSH packet and the cipher context is initialized with the generated IV. After that the SSH packet is encrypted in the same way as for STD packets. Also here a pointer to the com_state is kept to keep the conditional execution to a minimum.

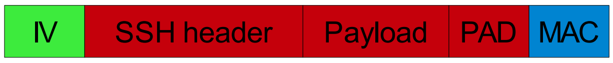


***Figure 12**: showing an encrypted UVPN packet.*

Decrypting packets

When a VPN channel is established by OpenSSH, all the packets going through this channel are read by non-blocking functions. The dispatch module then calls packet_read_poll_seqnr(), when the packets coming from the other side are to be processed. The packet_read_poll_seqnr() is thus the only function that needs to be able to fetch and decrypt UVPN packets from the UVPN input queue. The packet_read_seqnr(), for example does not need to able to fetch UVPN packets.

The packet_read_poll_seqnr(...) therefore uses packet_read_poll2(...) when reading packets from the input buffer. The packet_read_poll2(int seqnr) was modified to take a second argument, com_type. Which com_type the function should be called with, is decided by packet_read_poll2_imux().

```
int packet_read_poll2_imux(u_int32_t *seqnr, int std_only)
{
	if(active_state->packlen==0)
	{
		if(  ( !uvpn_have_data_to_read() || std_only )
		     && std_have_data_to_read())
		     return packet_read_poll2(COM_STD,seqnr);

		else if(uvpn_have_data_to_read())
		     return packet_read_poll2(COM_UVPN,seqnr);
		else{

		     return SSH_MSG_NONE;
		}
	}
	else
	{
		return packet_read_poll2(COM_STD,seqnr);
	}
}
```

***Code 5** : packet_read_poll2_imux() definition.*

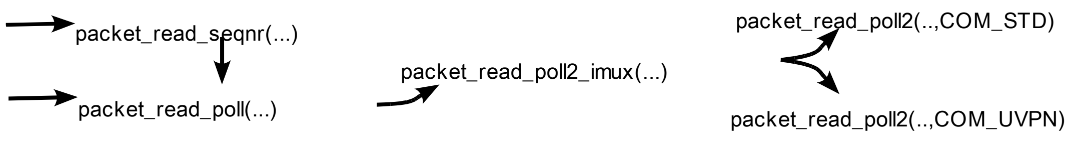
If there is a partial packet left in the incoming_packet buffer it is assumed that it belongs to the STD connection. The UVPN connection always processes the packets without splitting them, so a partial UVPN packet should never exist. Therefore, in case of partial packets, the function will be called with COM_STD.

UVPN packets are given precedence to make sure that no UVPN packets are left in the input buffer, if an SSH2_MSG_KEXINIT is received. The other side will never send any regular SSH packets after it has sent the kexinit. If UVPN packets are not given precedence it is possible that the UVPN packets, in the input queue, is delayed from being processed and the kexinit could then be processed out-of-order, in respect to the UVPN packets. This could cause UVPN packets to be residing in the input queue when the key exchange is started. The UVPN packets will not be read before the new keys have been set, which would not be possible because they are encrypted with the old keys.

By giving UVPN traffic precedence, the above scenario can, in most cases, be avoided. Although, the problem can still arise during heavy load on the connection, as explained in 4.3.1.

The decryption of packets coming from the other side is handled by packet_read_poll2(...). When UVPN packets are read, first the length of the IV is fetched from the cipher context and then the IV is read from the UVPN packet and the cipher context is initialized with this IV. After that the decryption is done in the same way as for STD packets. During decryption, a pointer to the general com_state is kept which allows easy access to the cipher context and keys.

Putting it all together, the packet_read_poll(int * seqnr, int std_only) thus calls packet_read_poll2_imux(...) instead of packet_read_poll2(...) directly, which calls the following function with the correct com_type.



packet_read_seqnr(...)
packet_read_poll(...)
packet_read_poll2_imux(...)
packet_read_poll2(..,COM_STD)
packet_read_poll2(..,COM_UVPN)

Figure 13*: The call sequence of the read functions.*

The usage of Queues for buffering UVPN packets

Our choice to use queues for buffering input and output UVPN packets is only a matter of design. A single large buffer could also have been used, as it is for the STD connection and using buffer_put_string() to maintain packet limits.

Nevertheless, when using queues and initializing the buffer pools in each direction, a static number of queue entries are created. Each entry contains a small buffer initialized to a size, calculated to fit a packet under the current MTU. In this way, the buffers should never have to further allocate memory. It should only be a matter of handling pointers when fetching buffer space.

Transport using UDP

To begin with, port numbers must be negotiated for the UDP connection. On the server side, this is done by searching a defined interval of port numbers for unused ports. The port range is specified in the sshd_config file.

A new socket is in the server case initialized by calling create_uvpn_server_sock(). This funciton calls uvpn_create_port_range(). Both located in misc_uvpn.c/.h. A list of available ports in the specified port range is created and uvpn_get_free_port() is used to get the first available port in the list. If a race occurs and the port is taken by some other application or another tunnel, the bind(...) call will return with an error and the port is marked unavailable. The uvpn_get_free_port() is then called again for the next port in the list.

On the client side, the port number specified by using the additional flag -U <port-nr>, when calling SSH, will be used. It is then up to the user to check that the port is not used. The client adds the port number to the CHANNEL_OPEN_REQUEST sent to the server when requesting the channel for the VPN tunnel. The server then reads the port number and sends its own port number in the CHANNEL_OPEN_CONFIRM message. Then, both sides know each others port numbers.

Avoiding IP fragmentation

The UDP protocol does not segment the stream into properly sized segments, based on the current MTU of the network path. Instead when calling sendto(...), as much data as possible is put into one single UDP segment. If this is not controlled, it can cause extensive IP fragmentation on the network.

The original implementation of OpenSSH sends the whole output buffer in one call to send(...). This works because the original implementation relies on TCP to divide the data into properly sized segments. This has to be done differently when using the UDP protocol.

As explained before, the VPN channel is used in datagram mode. Therefore, each SSH packet will contain only one IP packet. The size of the IP packet can be controlled by setting the MTU of the tun-device. In our modified implementation, it is possible to specify a MTU value in the configuration file which will be set automatically on the tun-device. The UVPN communication will put one SSH packet in each UDP packet, so the MTU should be specified so that no IP fragmentation will be caused on the network.

On the server side, a range of allowed MTU values is specified in the sshd_config file. The ssh_config file of the client side specifies which exact MTU it will propose for the tunnel. The client adds the MTU value right before the port number, in the CHANNEL_OPEN_REQUEST message sent for the VPN channel. In this way, the MTU value is sent to the server. If the MTU value is within the allowed range, the MTU of the server tun-device is set to the same value. Otherwise, the connection is disconnected.

The MTU is set when calling packet_setup_uvpn_module(int pmin,int pmax,int tun,int mtu). The pmin and pmax argument specifies the range of port numbers. The tun specifying the ID of the tun-device dedicated to the connection and mtu the MTU value to set for the tunnel. The server calls this function with pmin<pmax , and the client calls it with pmin=pmax.

Separate session keys and key exchange

As mentioned earlier, each connection has its own key set, stored in the com_state struct. In order to make this possible, a small change has been made to the key exchange mechanism where an extra field is added to the kexinit packet, indicating which connection the keys belong to. The field is only added when a UVPN connection is active, proving that the other side also is the modified version. This way the support for old non-modified SSH implementations is preserved.

In the session_state there is a new flag, rekey_com, that is set to either COM_STD or COM_UVPN, depending on which connection triggered the key exchange. The channel dedicated to the tunnel sends its data with help of the packet UVPN module. This module has its own cipher context and its own keys.

The packet_need_rekeying() is extended to return true also if the UVPN module need new keys. It also takes a pointer to an integer, com_type, as argument in which it can specify the connection that needs rekeying. In each iteration of the server loop and client loop, the need for rekeying is checked. When need for new keys is detected, the rekeying mechanism for the correct com_type will be triggered. When the new keys are ready to be set, the update_keys_after_kex(int mode) will call set_newkeys(int com_type,int mode) with the com_type specified in rekey_com.

4.2.4 Modifying channel.c

In order for the channel module to know when to use the UVPN extension, the flag uvpn_extension, was added to the Channel struct. The flag is set explicitly, by either the server loop or the client loop, when creating the channel.

Sending data to the other side of the channel is done with channel_output_poll(). In the normal case, the function creates an SSH packet of type SSH2_CHANNEL_DATA, using the functions supplied by the packet module. To add the possibility to use the UVPN extension channel_output_poll(), the usage of type SSH2_UVPN_DATA was introduced.

Reading VPN data from the other side of the channel is done with channel_input_data(). This function is called by dispatch, succeeding a call to packet_read_poll(). Because the packet_read_poll(), as explained above, decide from which connection the packet should be read, channel_input_data() does not need to be concerned about this. It mearly reads whatever is in the incoming_packet buffer. Therefore, no modification for reading packets was necessary in channel.c.

Another modification needed was in channel_input_open_confirmation(...) which is a function called on the client side when an SSH2_MSG_OPEN_CONFIRM packet is received from the server. This message is sent when the server confirms the opening of a channel. If the channel should be used for UVPN data on the server side, the server loop is modified to send the port number that will be used by the server for the UDP connection.

The function channel_input_open_confirmation(...) was therefore modified to read the port number with packet_uvpn_get_remote_uport(), see Appendix B.1.

4.2.5 Modifying the Server loop and Client loop

Most of the modifications done in the server loop and client loop are very similar. One important modification was to include the file descriptors for the UDP connection in the select() call. Also, the introduced packet type SSH2_UVPN_DATA, had to be added to the dispatch group. The type is associated with the channel_input_data() function, which is also used for VPN data in the normal STD connection.

A difference between the server loop and the client loop, of concern for the modification, is that the client side is the one sending the tunnel forwarding request and the server is the one that receives the request. The client_request_tun_fwd(...) function was modified, see appendix B.3, to include the MTU and UDP port which are proposed by the client.

On the server side, the dispatch module triggers server_input_channel_open(...) when receiving the request. This function was modified to call server_request_tun(int uvpn) with an argument denoting if the UVPN extension should be used, see appendix B.2. For the case when the UVPN extension is to be used, the server_request_tun(...) function has been modified to read the proposed MTU and the remote UDP port sent by the client. After the MTU value and the UDP port number are read, the UVPN extension is initialized.

4.2.6 Modifying kex.c

Because there are two key sets in the modification, the connection triggering a key exchange must be specified. In order to do this the kex_send_kexinit(...) function was modified to take a second argument com_type. The com_type was added in an additional filed to the kexinit message, see appendix B.5. In kex_input_kexinit(...) this field is read and the rekey_com flag in the packet module is set accordingly.

4.2.7 Modifying buffer.c

In order to use small buffers for the queue elements in the the packet module, it was preferable to be able to set the initial size of the buffer. The buffer_init(Buffer * buffer) was modified and buffer_init_len(Buffer * buffer,u_int buffer_len) was added, see appendix B.4. It was then possible to use either buffer_init(...) or buffer_init_len(...) to initialize a buffer. The initial buffer size can therefore be set according to the MTU of the tun-device when creating the queue elements.

4.3 Unsolved problems in the implementation

A few unsolved problems where detected for the modified implementation. In 4.3.1 the problems associated with the key exchange procedure and the use of two parallel connections are described. In 4.3.2, a potential vulnerability is presented caused by the predictability of the base connection and the forwarded traffic.

4.3.1 Known problems related to the rekeying process

Problem 1

When using the UVPN extension, two key sets are used. It is not likely, but possible, that both the STD connection and the UVPN connection need to requests new keys in approximately the same time, from opposite sides, see following figure.

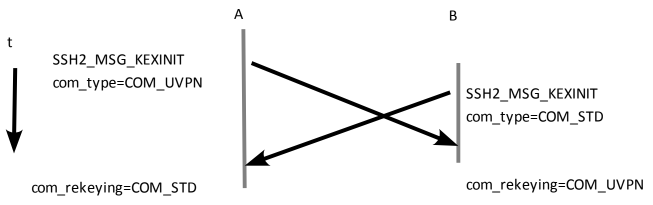


***Figure 14**: Problem 1. B sends a KEXINIT before it receives the kexinit sent by A for the opposite connection.*

The sender, A, of a kexinit sets the rekeying flag and the com_type indicating which of STD and UVPN that is rekeying. If the other side, B, manages to send a kexinit for the other connection during the time when the kexinit from A is travelling to B both sides will enter rekeying state but for different connections. The new keys will thus be set for STD at one side and for UVPN at the other, resulting in total packet corruption and disconnect.

Problem 2

During heavy load, packets delays may occur on the UDP connection. If the TCP socket sends the kexinit message, the delay can cause a UVPN packet sent before the kexinit, to arrive after the key exchange has begun. When this happens and the kexinit is for the UVPN connection, it causes the delayed packet to be undecipherable, since new keys are in use and the packet is encrypted with the old keys.

In the tests during heavy load on the tunnel, one corrupted packet was detected, for almost each key exchange. Because the modified implementation tolerates packet loss, this is not detectable for the user. Still, the packet loss is created by the application itself and must be considered to be a bug.

4.3.2 Security issue: predictability of VPN packets

In the VPN operation, each UDP packet contains exactly one SSH packet. Each SSH packet contains exactly one IP packet and so forth. The fields for packet length and PAD length in the SSH header can be predicted if the attacker have knowledge of the IP packet sizes that is forwarded over the connection. Also, the IP and TCP headers can be predicted. This leads to 46 bytes, if Ipv4, or 50 bytes, if Ipv6, predictable bytes in the beginning of each SSH packet. Lets denote the number of bytes by S.

Encryption and decryption in CBC mode, is done according to the equations (E1,E2). The predictable data in each SSH packet is larger than the block sizes supported by the implementation. It is possible to derive $D(C_i)=P_i \oplus C_{i-1}$ for $i \in I=[1,\lfloor S/\text{block\_size}\rfloor]$ in each packet and thus know the pairs $(P_i \oplus C_{i-1}, C_i)$ for $i \in I$. The attacker can therefore create an extensive list of {plaintext, ciphertext} pairs which can be used to crack the session keys. This together with the obtainable IV for each SSH packet makes it possible to derive plenty of blocks where IV plaintext and ciphertext is known.

In [18] a MITM attack on 3DES can be done from 3 known {plaintext, ciphertext} pairs, needing 256 units of storage and 2^{112} single encryption operations. More enhanced attacks are also presented in [18] where fewer steps are needed to the expense of more needed known pairs which for the attacker would not be a problem. For example using a block size of 128 bits (16 bytes) and considering the max_pack limit of at most 2^{31} packets, it should be possible to get 2-3 known pairs of blocks for each packet and 2^{32} pairs in total before new keys are set.

Proposed solution

By splitting a PAD of unpredictable length into two parts, a and b, and put part a in the beginning of the SSH packet and part b after the payload, it should remove the predictability. It would not be possible to predict the SSH header nor the beginning of the payload. Following, is a figure visualizing the structure of the SSH packet when applying the proposed modification.

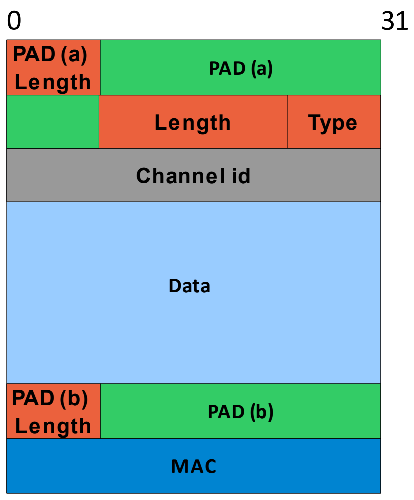


Figure 15*: The modified SSH packet in order to avoid predictability. The PAD divided into part a and part b which are preceded with length fields.*

This modification was not done in this project but is recommended for future work.

4.4 Sequence numbers

The modified implementation, presented in this report, does not use sequence numbers for the SSH packets going over the UVPN connection. The original OpenSSH implementation does not append sequence numbers in the packets, but instead stores a synchronized pair of sequence numbers for sending and receiving in the end hosts. The way of using the sequence numbers does not tolerate packet loss or out-of-order arrival of packets. Therefore, the original mechanism can not be used in the UVPN case.

By not using sequence numbers, the implementation is open for replay attacks. Any UVPN packet can be copied and retransmitted by an attacker and the application would not detect it.

The remedy of this problem is a field for a 32-bit sequence number in the UVPN packet header. The sender places the sequence number in this field and it will be included in the MAC calculation and the encryption. The receiver retrieves the sequence number by reading the field after decrypting the packet. In the receiver side, a sequence window should be maintained, in order to detect possible replay attacks.

Because of time limitations, this strategy was not fully implemented and is therefore excluded in the presented implementation.

Chapter 5, Testing

In the testing, the bandwidths of the tunnels where tested in order to compare the performance/goodput of the modified and non-modified SSH implementation. To evaluate the bandwidth available in the VPN tunnels, iperf was used. Packet loss has been emulated in the network by using Linux Network Traffic Control, tc, together with netem. To monitor the behaviour on the network, Wireshark was used.

First, the bandwidth of the test bench was tested to find an upper bound of what could be expected. It also shows the relation of the performance between the tunnels and the network, which could be general for other network settings.

Later, the bandwidths of the VPNs created with the original implementation and the modified implementation where tested.

5.1 The test bench

The test bench consisted of three desktop computers named Host A, B and C. On host A, the SSH server was running. Host B, was acting as a router between the two sub-nets network A and B, see figure 15. On Host C, the SSH client was running. The main purpose of the router was to be the base for the emulation software used in the tests.

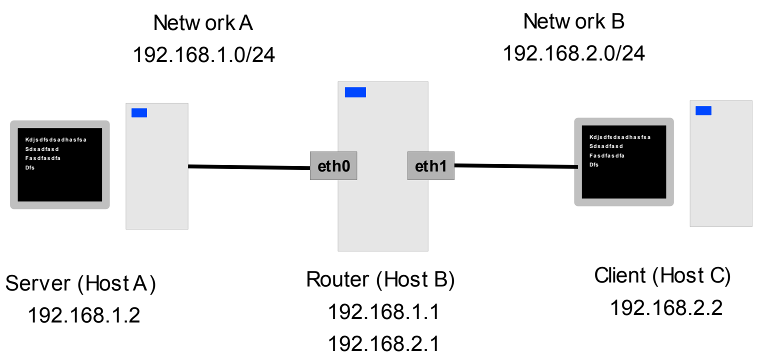


***Figure 16**: The test bench, showing the Server host, the Router host and the Client host.*

Network facts:

- $0.36 \text{ ms} < RTT_{AC} < 0.56 \text{ ms}$
- Measured bandwidth: ~95 Mbit/s (100Mbit/s network)

The configuration of host A and C:

- CPU: Intel Core 2 duo 3.0 GHz
- Operating system: Ubuntu (Linux)
- Network cards: 1Gbit(eth0)
- Max TCP sender window size: 4,194,304 bytes (~4MiB)

The configuration of Host B:

- CPU: AMD Athlon X2 2Ghz
- Operating system: Debian (Linux)
- Network cards: 1x 100Mbit(eth0) and 1x 1Gbit(eth1).

The bandwidth of the network was 100Mbit/s, due to the 100Mbit network card in host B. The test bench was isolated from other networks. The lack of direct access to the internet and the large set of applications available in the Debian offline installation, led to the use of Debian instead of Ubuntu in the router host.

5.2 Emulating packet loss with Linux Network Traffic Control, tc

Linux Network Traffic Control, tc, was used as the main network emulator in the tests [19].

The following command has been used to initialize packet loss on the router host:

```
tc qdisc add eth<x> root netem loss <percentage>
```

In the tests, data packets where sent strictly in one way, from the server host to the client host. The above command will apply the emulation on the egress traffic on the given interface. Applying the emulation on eth1 causes data packets to be lost during the tests. Emulating on eth0 causes ACKs to be lost.

5.3 Testing the bandwidth with iperf

iperf was used to derive the bandwidth for first the test bench and then for the established VPN tunnels. Each test was divided into two sub-tests. In the first sub-test, packet loss was emulated on traffic coming from the server. In the second sub-test, the emulation was instead done on traffic coming from the client. Each sub-test tested the bandwidth for packet loss probabilities increasing, in steps of 1, from 0 to 10 percent. For each percentage value, iperf was used five times during a 30 second period. The arithmetic mean of the these results was then used.

Three tests where conducted, where the first test was testing the bandwidth of the test bench without any tunnel. The second and third test where testing the bandwidth of VPN tunnels created with the original implementation and the modified implementation of OpenSSH respectively.

5.4 Emulating packet loss with Simple Packet Filter, *spf*

In order to verify our results from packet loss, we also repeated all tests with another application, Simple Packet Filter (spf). spf was used to emulate packet loss independently from tc. This was done to rule out that any errors had been caused by the use of tc for the emulation.

The spf filter uses two tun-devices in order to read IP packets from the kernel and, if they are not dropped, writes the packets back to the kernel. With spf the packet loss emulation was moved out from the kernel and gave more control over exactly what happened in the execution. The following figure visualizes the filter, filtering a packet going from Network A to B.

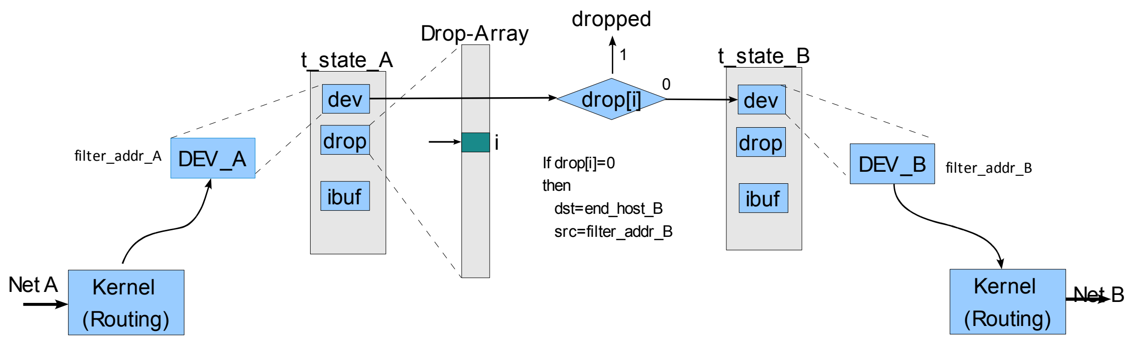


***Figure 17**: Filtering of packets, by spf, going from Network A to Network B.*

Whether a packet should be dropped or not, is decided by an randomized drop array. In order for the kernel to route the IP packets, first through the filter and then to the end host, the source and destination address of the packet is changed by the filter. At first, this was tried by using iptables, but problems seemed to arise when using NAT on the tun-devices. Instead, the network address is changed in spf directly. After the address has been changed, the IP checksum and the checksum for the transport layer protocol is updated and the packet is given back to the kernel.

5.5 Test results

In this part, the result from the testing is presented and analysed. In the tests, when using a tunnel, the MTU value of the tunnel is set to 1400 in order to give space for the encapsulating IP, TCP and SSH header. When testing only the test bench the MTU value of the interface where also set to 1400, to give a similar condition.

Dropping acknowledgements (ACKs)

The graph below shows the results from emulating packet loss on eth0. In this test ACKs where dropped. The spf filter was used to create control points which are used to verify the correctness of the test results by using a second packet loss emulator. The control points are in the graph denoted by triangles and are located sufficiently close to the results of the main tests to verify the correctness.

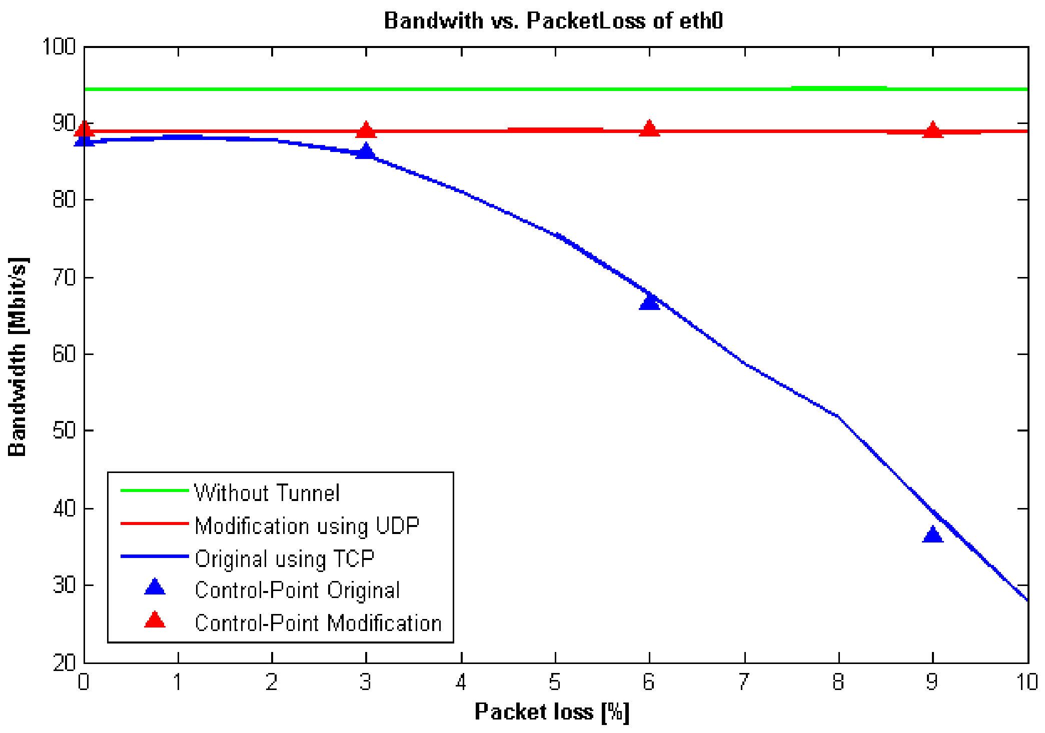


***Figure 18**: Result when emulating packet loss on eth0 of the router causing ACKs from the forwarded connection to be lost. Green=Test bench without tunnel. Blue=VPN tunnel using the original implementation. Red=VPN tunnel using the modified implementation.*

The result from the original implementation (TCP base connection) stands out from the modified implementation and the test system, as the packet loss increases. This is an effect of TCP in TCP tunnelling. The TCP base connection treats all the forwarded IP packets as data, whether it contains a TCP ACK or not.

Normally, in a TCP connection, a lost ACK is not retransmitted because the following ACK will cover for the lost ACK. This, because ACKs are cumulative and acknowledges all previously sent packets with lower sequence numbers then the ACK acknowledgement number.

Because the TCP base connection treats the ACKs of the forwarded connection as data, it will retransmit all the lost ACKs from the forwarded connection. Also, the base connection will stall the reconstruction of the packets succeeding the lost ACK, i.e. including succeeding ACKs, in the forwarded connection. Therefore, not only will it retransmit the non-needed ACKs it will also stall the forwarded connection until the ACKs are successfully retransmitted. This causes degraded performance when packet loss occurs. In the tests of the modified implementation using UDP base connection and the test bench without tunnel this does not occur because there exist no TCP stacking.

Dropping packets containing Data

In the following graph packet loss is applied on eth1 of the router host. This will thus cause packets containing data to be dropped.

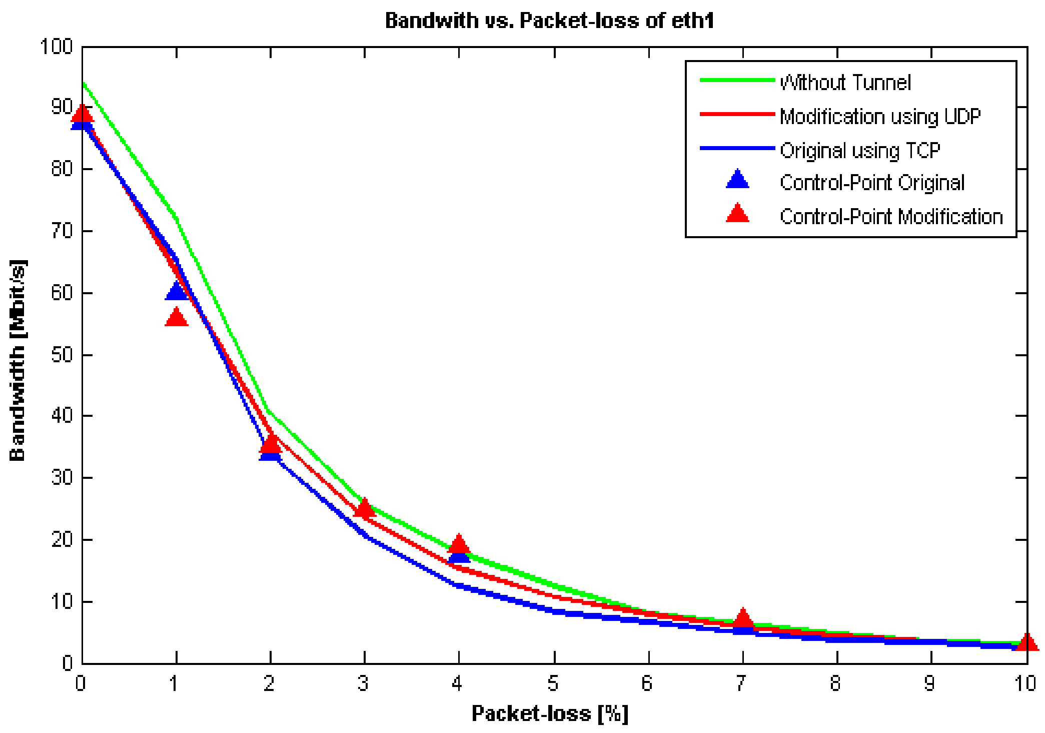


***Figure 19**: Result when emulating packet loss on eth1 of the router. This causes data packets to be lost. Green=Test bench without tunnel. Blue=VPN tunnel using the original implementation. Red=VPN tunnel using the modified implementation.*

Increasing the packets loss from 0 to 1 percent decreases the performance with approximately 30 percent. The huge decrease in performance was more drastic than what was expected. Because of this, a few steps where taken to rule out any basic problems that could cause the drastically decreasing bandwidth.

Steps to detect the cause of the excessive decrease in performance

First, it was confirmed that the Selective ACK (SACK) option was used by the operating system during the tests. To further make sure that only the lost packets where retransmitted, Wireshark was used to monitor the behaviour during packet loss. The investigation showed that only the correct packets where retransmitted.

Second, control points where created by repeating the tests for a chosen set of packet loss rates using spf in order to rule out any unforeseen errors from the emulation using tc. As seen in figure 19, the control points created when using spf follows the result of the main tests when using tc. It is therefore not likely that any problems where introduced in the packet loss emulation.

Third, in order to fully rule out that the router host was causing the unexpected behaviour, the router host was completely removed from the test system. Tests where then performed on the system, only consisting of the two end hosts.

This time, the tests where performed when emulating packet loss only in the data direction. The packet loss where thus emulated on the interface of Host A. To use tc for emulating packet loss on the end host of a connection and not on a bridge or router is not recommended in [20]. The packet loss created on the end host can be reported in the local system and TCP may act differently, to what it would have done if the packet loss where occurring on the network itself. Nevertheless, whether the results are exactly correct or not, it will give a hint to whether or not the router host where causing the pessimistic results. Following is the graph showing the results from emulating packet loss on the data packets.

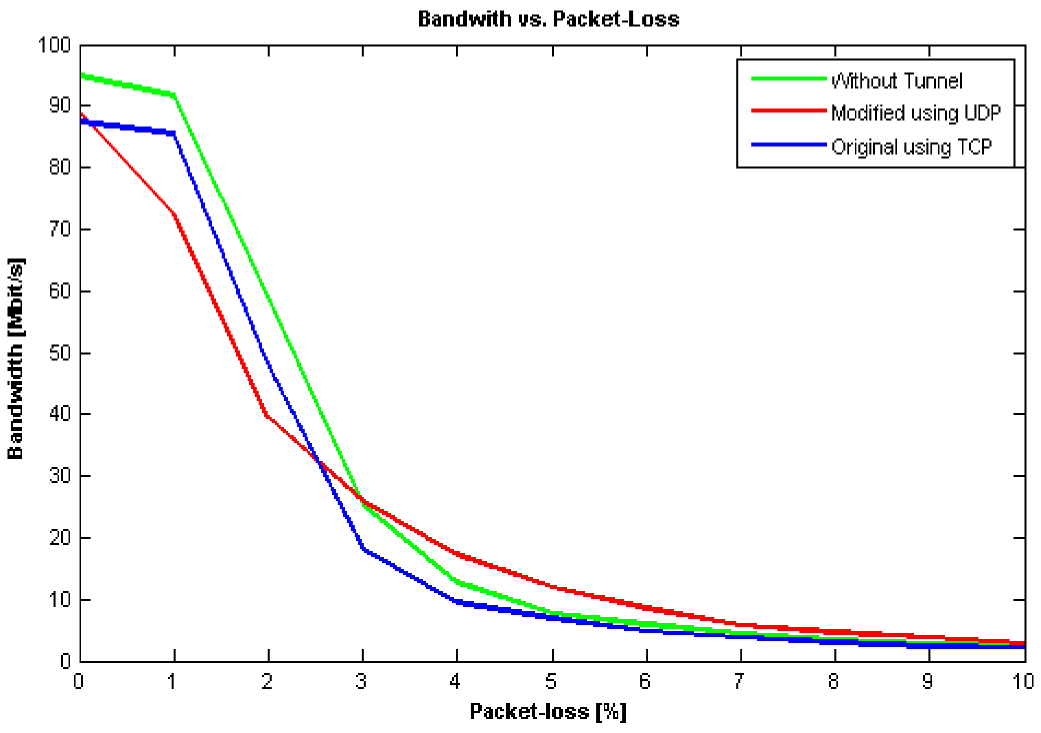


***Figure 20**: Result of tests when applying packet loss in the data direction. Here only the two end hosts where used in the test. The packet loss emulation was applied to the network interface of Host A. Green=Test bench without tunnel. Blue=VPN tunnel using the original implementation. Red=VPN tunnel using the modified implementation.*

The test show a slower decrease of the bandwidth for packet loss rates up to one percent. This may be caused by the TCP implementation, detecting the packet loss and directly retransmits the lost packets. After one percent the bandwidth starts to decrease drastically. Overall, the results are similar to the results when using the router host. Therefore, if there is an error in the test system, it does not seem to be in the router host.

Last, to rule out iperf to be the cause of any error, LinuxTTCP was used to test the available bandwidth. The result from LinuxTTCP was approximately the same.

Using the results

It is hard to find a reason for the overly drastic decrease in bandwidth when increasing the packet loss of forwarded data packets. Therefore, when no errors where found, the results are used to compare the performance of the implementations.

The results show that, in the case of lost data packets, the performance is only slightly better when using a UDP base connection, in comparison to using a TCP base connection. The biggest improvement came when ACKs where lost and the negative effects of TCP in TCP was the most clearest.

Unknown performance in case of TCP meltdown

The TCP meltdown scenario has not been detected during the tests and no attempt to trigger the scenario has been made. Therefore, it is not known, from the tests, what the differences in performance would be for such a scenario.

Chapter 6, Conclusions

6.1 Result

In terms of implementation effort, we have showed that it is feasible to modify the original OpenSSH v5.4p1 implementation to use UDP as base connection for its VPN feature. The code module that needed the most alterations where packet.c. If the modified packet.c/.h and the new packet_queue.c/.h and misc_uvpn.c/.h are supplied there are only moderate alterations needed in channels.c/.h, serverloop.c/.h, clientloop.c/.h, buffer.c/.h and kex.c/.h.

The tests performed on the modified and original implementation show that the most gain from using UDP for the base connection is when ACKs, belonging to the forwarded connection, are lost. As the packet loss rate increases, at least within 0 to 10 percent, the bandwidth is unchanged for the modified implementation, using a UDP base connection, whereas the original implementation, using a TCP base connection, suffers from decreased bandwidth.

When emulating packet loss in the data direction, there was not much difference between using a UDP base connection or a TCP base connection. The UDP based solution where performing only slightly better than the TCP based.

Another result worth mentioning, when not considering packet loss, is that the available bandwidth in the tunnels turned out to be very large in relation to the available bandwidth in the network. Approximately 90 percent of the bandwidth available in the test network have also been available in the tunnels.

6.2 Detected problems

As the implementation was carried out, we showed that the choice of using two parallel connections, i.e. a TCP connection for protocol messages and a UDP connection for the VPN data, can in some situations introduce problems related to the SSH rekeying mechanism, see chapter 4.3. The result is a few undecipherable packets, directly after the rekeying which can be viewed as packet loss introduced by the application itself. The packet loss is not detectable by the user and because rekeying is done relatively seldom there will be no noticeable performance loss. Still, this problem must be considered to be a bug in the application. A similar strategy to what is used for OpenVPN, namely using only one UDP connection, is therefore a better choice for the modification and is recommended as future work.

Another problem was detected when testing the bandwidth for different rates of packet loss on the packets containing forwarded data packets. The bandwidth was dropping more drastically then expected when the packet loss increased. Although trying to find an explanation for the unexpected performance drop, the reasons was not found in this project. A more thorough study of what was causing the drop in performance is therefore recommended as future work.

6.3 Recommendations for future work

The following is a list of recommendations for continued work on this subject.

- Adding a mechanism for sequence numbers in the UVPN connection, possibly as described in chapter 4.4 in order to prevent replay attacks.
- Remove the predictability of VPN packets by changing the structure of the SSH packet when using the VPN feature, see chapter 4.3.2.
- Implementation of a reliable transport layer on top of the UDP connection, only to be used by SSH protocol messages similar to OpenVPN [7]. In this way, only one connection is needed that uses UDP between the server and client. As a secondary benefit, it would not be necessary to use separate keys and cipher contexts. The rekeying conflict presented in chapter 4.3 would in that case not occur.
- A more thorough test when emulating packet loss, and possibly more complex network scenarios, is also needed to properly analyse the difference in performance. In this thesis, we have not been able to create detected occurrences TCP meltdown. Creating a scenario when TCP meltdown is induced, would make an interesting opportunity to test the differences in performance from using UDP instead of TCP for base connection.

References

[1] T. Ylonen, *"Secure Shell (SSH) Protocol Architecture"*, RFC4251, January 2006

[2] *"OpenSSH"*, [Accessed: March 16, 2010], Available: http://www.openssh.com/ .

[3] O. Titz, *"Why TCP Over TCP Is A Bad Idea"*, [Accessed: March 16, 2010] Available: http://sites.inka.de/sites/bigred/devel/tcp-tcp.html.

[4] Shashank Khanvilkar, Ashfaq Khokhar, *"Virtual Private Networks: An Overview with Performance Evaluation"*, Communications Magazine, IEEE, Volume 42, Issue 0, 2004, Pages: 146-154.

[5] Faruk Dabak, Sameer Panjwani, *"Tunneling TCP over TCP – A Study Of A Real System"*, Department of Computer Engineering, Chalmers University of Technology, Gothenburg 2006.

[6] Honda, Ohsaki, Imase, Ishizuka *"Understanding TCP over TCP: Effects of TCP Tunneling on End-to-End Throughput and Latency".*

[7] OpenVPN Technologies, "Security Overview: OpenVPN cryptographic layer", [Accessed: March 16, 2010]. Available: http://www.openvpn.net.

[8] Berg Canberg, Jaya Dhanesh, *"Tunneling TCP over TCP"*, Department of Computer Engineering, Chalmers University of Technology, Gothenburg 2004.

[9] "*CIPE (Crypto IP Encapsulation)"*, [Acessed: May 5,2010]. Available: http://sites.inka.de/~bigred/devel/CIPE-Protocol.txt

[10] T. Ylonen, *"Secure Shell (SSH) Transport Layer Protocol"*, RFC4253.

[11] T. Ylonen, *"Secure Shell (SSH) Connection Protocol"*, RFC4254.

[12] James F. Kurose, Keith W. Ross, *Computer Networking: A top down approach featuring the Internet*, Third Edition, Pearson Education, Inc. United States of America, 2005.

[13] S. Kent, K. Seo, "Security Architecture of the Internet Protocol",RFC4301, December 2005.

[14] S. Kent, "IP Encapsulating Security Payload (ESP)", RFC4303, December 2005.

[15] T. Dierks, E. Rescorla, *"The Transport Layer Security (TLS) Protocol"*, Version 1.2, RFC 5246, August 2008.

[16] Wade Trappe, Lawrence Washington, *"Introduction to Cryptography with Coding Theory"*, Second Edition, Washington, Pearson Education, Inc. 2006.

[17] Source code, OpenSSH implementaion.

[18] Stefan Lucks, *”Attacking Triple Encryption”*, Theorestische Informatik Universität Manheim, Germany, 1998.

[19] Ariane Keller, *Manual tc Packet Filtering and netem*, ETH Zurich, July 20, 2006

[20] Linux Foundation, “*netem”*, [Accessed: May 25, 2010], Available: http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

Appendix A - Test results

A.1 Main tests

Emulator: tc , bandwidth measurement: iperf , test-time: 30 s

Original Implementation												
Pkt-loss (%)	**eth0**						**eth1**					
	#1	**#2**	**#3**	**#4**	**#5**	**AM**	**#1**	**#2**	**#3**	**#4**	**#5**	**AM**
0	87.4	87.6	87.6	87.6	87.6	**87,56**	87.6	87.6	87.6	87.6	87.6	**87.6**
1	88.2	88.1	88.2	88.1	88	**88.12**	63.8	64.7	66.7	65	65.5	**65.14**
2	87.8	87.8	88	87.5	88	**87.82**	32.6	33.2	31.9	34.8	36.4	**33.78**
3	85.7	87.1	86.4	86.2	83.8	**85.84**	18.3	20.5	20.9	21	22	**20.54**
4	82.4	82.7	81.9	83.5	74.8	**81.06**	12.5	11.9	12.7	12	13.1	**12.44**
5	73.9	77.5	76.8	80.3	68.9	**75.48**	7.41	7.77	7.83	9.84	8.46	**8.26**
6	65.6	72	65.9	67.1	68.2	**67.76**	6.11	7.15	6.35	6.95	6.69	**6.65**
7	49.8	68.1	57.9	54.2	64	**58.8**	4.45	5.42	4.49	5.4	4.53	**4.86**
8	46.4	51	53.9	49.3	58.3	**51.78**	3.35	3.79	3.38	3.44	4.09	**3.61**
9	49.6	25.7	44.7	34.2	43	**39.44**	4.11	3.11	3.13	3.44	2.47	**3.25**
10	25.7	24.4	34.9	27.2	27	**27.84**	2.4	2.44	2.14	2.55	1.91	**2.29**

Modified Implementation												
Pkt-loss (%)	**eth0**						**eth1**					
	#1	**#2**	**#3**	**#4**	**#5**	**AM**	**#1**	**#2**	**#3**	**#4**	**#5**	**AM**
0	88.9	88.8	88.9	88.9	89	**88.9**	89	88.9	88.9	88.9	88.9	**88.92**
1	89	88.9	88.9	88.9	88.9	**88.92**	63.5	63.6	63.6	62.4	63.2	**63.26**
2	89	88.9	88.9	88.9	88.9	**88.92**	37.5	36	38	37.9	37.6	**37.4**
3	89	89	88.9	88.9	88.9	**88.94**	22.7	21.8	25.1	23.3	23.8	**23.34**
4	89	88.9	88.9	88.9	88.9	**88.92**	14.1	15	16.1	15.8	15.7	**15.34**
5	89	89	88.9	89	88.9	**88.96**	11.5	10.6	11.2	10	9.76	**10.61**
6	89	88.9	88.9	88.9	88.9	**88.92**	8.03	8.49	8.24	6.96	7.94	**7.93**
7	89	88.9	88.9	88.9	88.9	**88.92**	6.07	5.92	5.85	5.38	5.81	**5.81**
8	89	89	88.9	88.6	89	**88.9**	4.49	3.5	4.55	4.64	4.24	**4.28**
9	89	89	88.9	88.9	88.4	**88.84**	3.69	3.32	3.51	3.29	2.82	**3.33**
10	89	88.9	89	88.9	88.9	**88.94**	2.44	2.36	2.66	2.34	2.88	**2.54**

Test-Bench												
Pkt-loss (%)	**eth0**						**eth1**					
	#1	**#2**	**#3**	**#4**	**#5**	**AM**	**#1**	**#2**	**#3**	**#4**	**#5**	**AM**
0	94.5	94.2	94.2	94.2	94.2	**94,26**	94.5	94.2	94.3	94.3	94.2	**94.3**
1	94.5	94.3	94.3	94.3	94.3	**94.34**	72.6	71.4	71.2	72.1	72.1	**71.88**
2	94.5	94.3	94.2	94.2	94.3	**94.3**	42.5	40.1	38.4	39.5	41.6	**40.42**
3	94.5	94.3	94.2	94.2	94.3	**94.3**	23	26.2	24.5	23	28.6	**25.06**
4	94.5	94.3	94.6	94	94.3	**94.34**	18.1	17.1	17.9	18.6	18	**17.94**
5	94.5	94.3	94.2	94.3	94.2	**94.3**	11.8	13.1	12.5	12.6	12.2	**12.44**
6	94.5	94.2	94.3	94.3	94.3	**94.32**	7.95	8.89	7.31	7.63	8.45	**8.05**
7	94.5	94.3	94.3	94.2	94.3	**94.32**	5.71	6.29	6.7	6.24	6.33	**6.25**
8	94.5	94.2	94.2	94.5	94.5	**94.38**	3.66	4.7	5.15	4.54	5.84	**4.78**
9	94.5	94.3	94.3	94.3	94.3	**94.34**	3.5	3.3	2.72	3.15	4.04	**3.34**
10	94.5	94.3	94.3	94.2	94.3	**94.32**	3.24	2.82	2.59	2.65	3.39	**2.94**

A.2 Control-points

Emulator: spf , bandwith measurement: iperf , test-time: 30 s

Original Implementation												
Pkt-loss (%)	**BtoA eq. eth0**						**AtoB eq. eth1**					
	#1	**#2**	**#3**	**#4**	**#5**	**AM**	**#1**	**#2**	**#3**	**#4**	**#5**	**AM**
0	87.6	87.6	87.4	87.8	87.7	**87.62**	87.6	87.6	87.4	87.8	87.7	**87.62**
1	-	-	-	-	-	**0**	60.3	61.5	59.2	58.8	60.3	**60.02**
2	-	-	-	-	-	**0**	33.5	34.5	34.1	33.8	33.5	**33.88**
3	86.5	85.7	86.1	86.4	85.6	**86.06**	24.9	25.2	24.5	24.4	25.3	**24.86**
4	-	-	-	-	-	**0**	17.3	17.7	17.7	17.6	16	**17.26**
6	59.6	65.5	66.3	67.8	73.9	**66.62**	-	-	-	-	-	**0**
7	-	-	-	-	-	**0**	5.72	6.15	6.33	5.85	5.82	**5.97**
9	38.3	35.5	36.7	32	38.7	**36.24**	-	-	-	-	-	**0**
10	-	-	-	-	-	**0**	3.27	3.13	2.68	3	3.33	**3.08**

Modified Implementation												
Pkt-loss (%)	**BtoA eq. eth0**						**AtoB eq. eth1**					
	#1	**#2**	**#3**	**#4**	**#5**	**AM**	**#1**	**#2**	**#3**	**#4**	**#5**	**AM**
0	88.8	89	89	89	89	**88.96**	88.8	89	89	89	89	**88.96**
1	-	-	-	-	-	**0**	56.8	54.8	55.7	56.4	54.9	**55.72**
2	-	-	-	-	-	**0**	34.4	34.7	36.3	36.6	34.7	**35.34**
3	88.8	88.9	88.9	89	89	**88.92**	25.8	23.8	25	24.4	25.3	**24.86**
4	-	-	-	-	-	**0**	19.2	18.8	18.8	19.4	18.8	**19**
6	89	89	89	89	89	**89**	-	-	-	-	-	**0**
7	-	-	-	-	-	**0**	7.67	6.53	6.86	7.26	7.23	**7.11**
9	89	89	88.8	89	88.9	**88.94**	-	-	-	-	-	**0**
10	-	-	-	-	-	**0**	2.89	3.22	3.49	2.96	3.14	**3.14**

Appendix B – Excerpts from modified modules

B.1 Excerpts from the modified channel.c

```
void channel_output_poll()
{
      ...
      if (c->datagram) {
            if (len > 0) {
                  u_char *data;
                  u_int dlen;
                  data = buffer_get_string(&c->input,
                      &dlen);
                  if(c->uvpn_extension)
                        packet_start(SSH2_UVPN_DATA);
                  else
                        packet_start(SSH2_MSG_CHANNEL_DATA);
                  packet_put_int(c->remote_id);
                  packet_put_string(data, dlen);
                  packet_send();
                  c->remote_window -= dlen ;
                  xfree(data);
            }
            continue;
      }
      ...
}
```

Code B1: Excerpt from `channel_output_poll()`.

```
void channel_input_open_confirmation()
{
      debug2("channel %d: open confirm rwindow %u rmax %u", c->self,
          c->remote_window, c->remote_maxpacket);
      if(c->uvpn_extension)
      {
            /* Retreive the remote port and start the UVPN module */
            debug("channel %d: open confirm remote port received",c-
>self);
            packet_uvpn_get_remote_uport();
            packet_init_uvpn_module();
```

```
        }
}
```

Code B2 *: excerpt from channel_input_open_confirmation(...). The highlighted code-area is used to set the remote UDP port and to initialize the UVPN extension.*

B.2 Excerpts from the modified server_loop.c

```
static void server_input_channel_open(...)
{
        ...
        if (strcmp(ctype, "session") == 0) {
                c = server_request_session();
        } else if (strcmp(ctype, "direct-tcpip") == 0) {
                c = server_request_direct_tcpip();
        } else if (strcmp(ctype, "tun@openssh.com") == 0) {
                c = server_request_tun(0);
        } else if (strcmp(ctype, "uvpn_tun")==0){
                c = server_request_tun(1);
        }
        ...
}
```

Code B3*: Modification of the server_input_channel_open(...) to be able to handle the incoming tunnel-request using the UVPN extension.*

```
static Channel * server_request_tun(int uvpn)              (line: 998)
{
        ...
        if(uvpn)
        {
                c = channel_new("tun", SSH_CHANNEL_OPEN, sock, sock, -1,
                        CHAN_UDP_WINDOW_DEFAULT, CHAN_UDP_PACKET_DEFAULT, 0,
                                "tun", 1);
                c->datagram = 1;
                c->uvpn_extension=1;
                mtu = packet_get_int();
                if(!(options.uvpn_mtu_min <= mtu && mtu <=
                                options.uvpn_mtu_max))
                    packet_disconnect("Unsupported MTU!, should be withing
[%d,                 %d]", options.uvpn_mtu_min, options.uvpn_mtu_max);

                uvpn_sock = packet_setup_uvpn_module(options.uvpn_port_min,
                                             options.uvpn_port_max,
                                             tun,mtu);

                packet_uvpn_get_remote_uport();
                packet_init_uvpn_module();
        }
        else
        {
                c = channel_new("tun", SSH_CHANNEL_OPEN, sock, sock, -1,
                        CHAN_TCP_WINDOW_DEFAULT, CHAN_TCP_PACKET_DEFAULT, 0,
                                "tun", 1);
                c->datagram = 1;
        }
        ...
}
```

Code B4*: Exceprt from the server_request_tun(...) funciton. The high-lighted code-area is used to setup a channel and initialize the UVPN extension.*

```
static void process_input(fd_set * readset)
{
        ...
        if (uvpn_sock!=-1 && FD_ISSET(uvpn_sock, readset))
        {
                packet_uvpn_receive_pkts();
        }
        ...
}
```

Code B5*: Excerpt from the process_input function. Here, if UDP packets are available to be read, the packet_uvpn_reaceive_pkts() will be called to read all the packets.*

```
static void server_init_dispatch_20(void)
{
        ...
        dispatch_set(SSH2_UVPN_DATA,&channel_input_data);
}
```

Code B6*: Associates the channel_input_data function to the UVPN packet-type in the dispatch group.*

```
static void wait_until_can_do_something(...)
{
        if (compat20) {
                ...
                FD_SET(connection_in, *readsetp);
                if(uvpn_sock!=-1)
```

```
                        FD_SET(uvpn_sock,*readsetp);

        }
        ...
        if (packet_have_data_to_write())
                FD_SET(connection_out, *writesetp);
        if(uvpn_sock!=-1 && packet_uvpn_have_data_to_write() )
        {
                        FD_SET(uvpn_sock,*writesetp);
        }
        ...
}
```

***Code B7**: Adding the file-descriptors for the UVPN connection. Similarly i done in client_loop.c.*

B.3 Excerpts from the modified client_loop.c

```
int client_request_tun_fwd(...)
{
        ...
        if(options.use_uvpn)
        {
                c = channel_new("tun", SSH_CHANNEL_OPENING, fd, fd, -1,
                        CHAN_UDP_WINDOW_DEFAULT, CHAN_UDP_PACKET_DEFAULT, 0,
                                        "tun", 1);
                c->datagram = 1;
                c->uvpn_extension=1;
                uvpn_sock =
                        packet_setup_uvpn_module(options.uvpn_local_port,
                                        options.uvpn_local_port,
                                        local_tun,options.uvpn_mtu);
        }
        else
        {
                c = channel_new("tun", SSH_CHANNEL_OPENING, fd, fd, -1,
                        CHAN_TCP_WINDOW_DEFAULT, CHAN_TCP_PACKET_DEFAULT, 0,
                                        "tun", 1);
                c->datagram = 1;
        }

#if defined(SSH_TUN_FILTER)
        if (options.tun_open == SSH_TUNMODE_POINTOPOINT)
                channel_register_filter(c->self, sys_tun_infilter,
                  sys_tun_outfilter, NULL, NULL);
#endif

        packet_start(SSH2_MSG_CHANNEL_OPEN);
        if(options.use_uvpn)
                packet_put_cstring("uvpn_tun");
        else
                packet_put_cstring("tun@openssh.com");
        packet_put_int(c->self);
        packet_put_int(c->local_window_max);
        packet_put_int(c->local_maxpacket);
        packet_put_int(tun_mode);
        packet_put_int(remote_tun);
        if(options.use_uvpn)
        {
                packet_put_int(options.uvpn_mtu);
                packet_uvpn_put_local_uport();
        }
        packet_send();
        return 0;
}
```

Code B8*: modified client_request_tun_fwd(...)*

B.4 Excerpts from the modified buffer.c

```
void buffer_init(Buffer *buffer)
{
      const u_int len = 4096;

      buffer_init_len(buffer,len);
}
void buffer_init_len(Buffer *buffer,u_int len)
{
      buffer->alloc = 0;
      buffer->buf = xmalloc(len);
      buffer->alloc = len;
      buffer->offset = 0;
      buffer->end = 0;
}
```

Code B9*:* `buffer_init(...)` *is divided into two functions where* `buffer_init(...)` *is used as normal but the possibility is added for external* *modules to use buffer_init_len(...,u_int_len). It is then possible to specify an explicit buffer-length for the buffer.*

B.5 Excerpts from the modified Kex.c

```
void kex_send_kexinit(...,int com_type)
{
        ...
        for (i = 0; i < KEX_COOKIE_LEN; i++) {
                if (i % 4 == 0)
                        rnd = arc4random();
                cookie[i] = rnd;
                rnd >>= 8;
        }
        packet_start(SSH2_MSG_KEXINIT);
        if(packet_uvpn_active())
                packet_put_int(com_type);
        packet_put_raw(buffer_ptr(&kex->my), buffer_len(&kex->my));
        packet_send();
        debug("SSH2_MSG_KEXINIT sent");
        if(com_type!=-1)
                packet_set_rekeying(com_type);
        ...
}
void kex_input_kexinit(...)
{
        ...
        debug("SSH2_MSG_KEXINIT received");
        if (kex == NULL)
                fatal("kex_input_kexinit: no kex, cannot rekey");

        if(packet_uvpn_active())
        {
                com_type = packet_get_int();
                if(com_type != -1)
                        packet_set_rekeying(com_type);
        }
        else
              packet_set_rekeying(COM_STD);
        ...
  }
```

Appendix C – Modified packet.c

```
#include "includes.h"

#include <sys/types.h>
#include "openbsd-compat/sys-queue.h"
#include <sys/param.h>
#include <sys/socket.h>
#ifdef HAVE_SYS_TIME_H
# include <sys/time.h>
#endif

#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>

#include <errno.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>

#include <openssl/err.h>
#include <openssl/rand.h>

#include "xmalloc.h"
#include "buffer.h"
#include "packet.h"
#include "crc32.h"
#include "compress.h"
#include "deattack.h"
#include "channels.h"
#include "compat.h"
#include "ssh1.h"
#include "ssh2.h"
#include "cipher.h"
#include "key.h"
#include "kex.h"
#include "mac.h"
#include "log.h"
#include "canohost.h"
#include "misc.h"
#include "ssh.h"
#include "roaming.h"
#include "packet_queue.h"
#include "misc_uvpn.h"

#ifdef PACKET_DEBUG
#define DBG(x) x
#else
#define DBG(x)
#endif

#define PACKET_MAX_SIZE (256 * 1024)
#define MAX_QUEUE_SIZE 500

/* Limit for how many corrupted packets that is allowed to be
 * received back-to-back. */
#define MAX_STREAM_CORRUPTED 10

struct packet_state {
          u_int32_t seqnr;
          u_int32_t packets;
          u_int64_t blocks;
          u_int64_t bytes;
};

struct packet {
          TAILQ_ENTRY(packet) next;
          u_char type;
          Buffer payload;
};

/*
 * General state for both STD communication and UVPN communication.
 */
typedef struct c_state{

          int initialized;

          int connection_in;
          int connection_out;

          CipherContext receive_context;
          CipherContext send_context;

          Newkeys *newkeys[MODE_MAX];

          struct packet_state p_read, p_send;
          u_int64_t max_blocks_in, max_blocks_out;
          u_int32_t rekey_limit;
          u_int64_t max_packets;

} C_State;

/*
 * Specific state for the STD communication
 */
typedef struct std_state {

          int initialized;



          /* General comunicaiton state */
          C_State com_state;

          /* Input buffer for the communication */
          Buffer input;

          /* Output buffer for the communication */
          Buffer output;

          /* XXX discard incoming data after MAC error */
          u_int packet_discard;
          Mac *packet_discard_mac;

          /* Session key for protocol v1 */
          u_char ssh1_key[SSH_SESSION_KEY_LENGTH];
          u_int ssh1_keylen;


} S_State;

/*
 * Specific state for the UVPN communication
 */
typedef struct uvpn_state {

          /* All allocated, connection set,
           * cipher not necessarily ready */
          int initialized;

          /* General communication state */
          C_State com_state;

          /* Input queue for the communication */
          PktQueue * input;

          /* Input queue for the communication */
          PktQueue * output;

          /* Buffer pools for each direction containing
           * free buffers to be used*/
          PktQueue * pool[MODE_MAX];

          /* MTU for the tun-device, (the tunnel) */
          int mtu;

          /* Variable to detect if a stream of corrupted packets are sent to the
           * UDP socket. If so, the connection should be disconnected to divert
           * possible DoS attack. */
          int stream_corrupted_packets;

          /* Denoting if the uvpn connection is ready to send encrypted data in
           * a sertain direction , i.e. if keys and everything is ready.*/
          int ready[MODE_MAX];
	struct sockaddr_in * local_host;
	struct sockaddr_in * remote_host;


} U_State;

/*
 * State for the entire Packet-module
 */
struct session_state{

	/* State of the STD communication */
	S_State * std_state;

	/* State of the UVPN communication */
	U_State * uvpn_state;

	/* Protocol flags for the remote side. */
	u_int remote_protocol_flags;
	/*
	 * Flag indicating whether packet compression/decompression is
	 * enabled.
	 */
	int packet_compression;

	/* default maximum packet size */
	u_int max_packet_size;

	/* Buffer for the partial outgoing packet being constructed. */
	Buffer outgoing_packet;

	/* Buffer for the incoming packet currently being processed. */
	Buffer incoming_packet;

	/* Scratch buffer for packet compression/decompression. */
	Buffer compression_buffer;
	int compression_buffer_ready;

	/* Set to true if the connection is interactive. */
	int interactive_mode;

	/* Set to true if we are the server side. */
	int server_side;

	/* Set to true if we are authenticated. */
	int after_authentication;

	/* Used in packet_read_poll2() */
	u_int packlen;

	/* Used in packet_send2 */
	int rekeying;
	int rekey_com;

	int initialized;
	/* Used in packet_set_interactive */
	int set_interactive_called;

	/* Used in packet_set_maxsize */
	int set_maxsize_called;

	/* roundup current message to extra_pad bytes */
	u_char extra_pad;

	int keep_alive_timeouts;

	/* The maximum time that we will wait to send or receive a packet */
	int packet_timeout_ms;

	TAILQ_HEAD(, packet) outgoing;
};

static struct session_state *active_state, *backup_state;

static struct session_state *
alloc_session_state(void)
{
	struct session_state *s = xcalloc(1, sizeof(*s));

	s->std_state  = (S_State *) xcalloc(1, sizeof(S_State));
	s->uvpn_state = NULL;
```

```
          s->std_state->com_state.connection_in = -1;
          s->std_state->com_state.connection_out = -1;
          s->std_state->com_state.max_packets = (u_int64_t) 1 << 31;
          s->std_state->initialized=0;

          s->max_packet_size = 32768;
          s->packet_timeout_ms = -1;

          return s;
}
void alloc_uvpn_state(void)
{
          struct session_state * s = active_state;
          debug("allocating UVPN state");
          s->uvpn_state = (U_State *) xcalloc(1, sizeof(U_State));
          s->uvpn_state->com_state.connection_in = -1;
          s->uvpn_state->com_state.connection_out = -1;
          s->uvpn_state->com_state.initialized = 0;
          s->uvpn_state->com_state.max_packets = (u_int64_t) 1 << 31;
          s->uvpn_state->ready[MODE_IN]  = 0;
          s->uvpn_state->ready[MODE_OUT] = 0;
          s->uvpn_state->mtu = 0;
          s->uvpn_state->local_host = uvpn_allocate_sockaddr_in();
          s->uvpn_state->remote_host= uvpn_allocate_sockaddr_in();
}

void packet_set_rekeying(int com_type)
{
          debug("set rekeying %d",com_type);
          active_state->rekeying = 1;
          active_state->rekey_com = com_type;
}

int packet_uvpn_active()
{
          if(active_state->uvpn_state)
                    return 1;
          return 0;
}


/*
 * Sets the descriptors used for communication.  Disables encryption until
 * packet_set_encryption_key is called.
 */
void
set_connection(int ct, int fd_in, int fd_out)
{
          C_State * cs;

          Cipher *none = cipher_by_name("none");


          if (none == NULL)
                    fatal("packet_set_connection: cannot load cipher 'none'");
          if (active_state == NULL)
                    active_state = alloc_session_state();

          if(ct == COM_STD)
                    cs = &active_state->std_state->com_state;

          else if(ct == COM_UVPN)
          {
                    if(!active_state->uvpn_state)
                              fatal("uvpn_state is not allocated!");

                    cs = &active_state->uvpn_state->com_state;
          }
          else
                    fatal("App. Bug : set_connection() , bad com_type");

          cs->connection_in = fd_in;
          cs->connection_out = fd_out;
          cipher_init(&cs->send_context, none, (const u_char *)"",
              0, NULL, 0, CIPHER_ENCRYPT);
          cipher_init(&cs->receive_context, none, (const u_char *)"",
              0, NULL, 0, CIPHER_DECRYPT);
          cs->newkeys[MODE_IN] = cs->newkeys[MODE_OUT] = NULL;
          if(!cs->initialized)
          {
                    cs->initialized=1;
                    cs->p_send.packets = cs->p_read.packets = 0;
          }
          if(ct == COM_STD)
          {
                    buffer_init(&active_state->std_state->input);
                    buffer_init(&active_state->std_state->output);
          }
          else
          {

                    active_state->uvpn_state->input = packet_queue_create_queue();
                    active_state->uvpn_state->output = packet_queue_create_queue();

                    active_state->uvpn_state->pool[MODE_IN] =
                                        packet_queue_create_buf_pool(MAX_QUEUE_SIZE,
                                                  active_state->uvpn_state->mtu);
                    active_state->uvpn_state->pool[MODE_OUT]=
                                        packet_queue_create_buf_pool(MAX_QUEUE_SIZE,
                                                  active_state->uvpn_state->mtu);


          }

          if(!active_state->initialized)
          {
                    active_state->initialized = 1;
                    buffer_init(&active_state->outgoing_packet);
                    buffer_init(&active_state->incoming_packet);
                    TAILQ_INIT(&active_state->outgoing);
          }
}
/*
 * Sets the descriptors of the STD connection.
 *
 * (name unchanged to minimice external code alteration)
 */
void packet_set_connection(int fd_in, int fd_out)
{
          set_connection(COM_STD, fd_in, fd_out);
}

int packet_setup_uvpn_module(int pmin, int pmax ,int tun, int mtu)
{
          int socket;
          alloc_uvpn_state();
          if(!active_state->uvpn_state)
                    fatal("Could not allocate UVPN state");
          if(pmin < pmax)
          {
                    debug("Creating server UDP socket within port-range: [%d,%d]",pmin,pmax);
                    socket= create_uvpn_server_sock(
                              active_state->uvpn_state->local_host,pmin,pmax);
          }
          else if(pmin==pmax)
          {
                    debug("Creating UDP socket with port: [%d]",pmin);
                    socket = create_uvpn_sock(active_state->uvpn_state->local_host,
                                                            pmin);
          }
          else
                    fatal("packet_setup_uvpn_module(): Bad port range");

          active_state->uvpn_state->mtu = mtu;

          debug("Using UDP-port: %d",active_state->uvpn_state->local_host->sin_port);

          debug("Setting MTU: [tun%d , mtu%d]",tun,mtu);


          set_connection(COM_UVPN, socket ,socket);
          set_tun_mtu(tun,mtu);

          return socket;

}
/*
 * This method is called to start the module
 */
void packet_init_uvpn_module()
{
          if(!active_state->uvpn_state)
                    fatal("App Bug : uvpn state is not allocated");
          if(active_state->uvpn_state->initialized)
                    fatal("App Bug: uvpn_state is already initialized!");

          if(check_uvpn_addr_conf(active_state->uvpn_state->local_host,
                                        active_state->uvpn_state->remote_host)==-1)
                    fatal("Bad UDP address configuration");

          active_state->uvpn_state->initialized=1;

}



void
packet_set_timeout(int timeout, int count)
{
          if (timeout == 0 || count == 0) {
                    active_state->packet_timeout_ms = -1;
                    return;
          }
          if ((INT_MAX / 1000) / count < timeout)
                    active_state->packet_timeout_ms = INT_MAX;
          else
                    active_state->packet_timeout_ms = timeout * count * 1000;
}

static void
packet_stop_discard(void)
{
          if (active_state->std_state->packet_discard_mac) {
                    char buf[1024];

                    memset(buf, 'a', sizeof(buf));
                    while (buffer_len(&active_state->incoming_packet) <
                        PACKET_MAX_SIZE)
                              buffer_append(&active_state->incoming_packet, buf,
                                  sizeof(buf));
                    (void) mac_compute(active_state->std_state->packet_discard_mac,
                        active_state->std_state->com_state.p_read.seqnr,
                        buffer_ptr(&active_state->incoming_packet),
                        PACKET_MAX_SIZE);
          }
          logit("Finished discarding for %.200s", get_remote_ipaddr());
          cleanup_exit(255);
}

static void
packet_start_discard(Enc *enc, Mac *mac, u_int packet_length, u_int discard)
{
          if (enc == NULL || !cipher_is_cbc(enc->cipher))
                    packet_disconnect("Packet corrupt");
          if (packet_length != PACKET_MAX_SIZE && mac && mac->enabled)
                    active_state->std_state->packet_discard_mac = mac;
          if (buffer_len(&active_state->std_state->input) >= discard)
                    packet_stop_discard();
          active_state->std_state->packet_discard = discard -
              buffer_len(&active_state->std_state->input);
}

/* Returns 1 if remote host is connected via socket, 0 if not. */

int
packet_connection_is_on_socket(void)
{
          struct sockaddr_storage from, to;
          socklen_t fromlen, tolen;
          C_State * cs = &active_state->std_state->com_state;

          /* filedescriptors in and out are the same, so it's a socket */
          if (cs->connection_in == cs->connection_out)
                    return 1;
          fromlen = sizeof(from);
          memset(&from, 0, sizeof(from));
          if (getpeername(cs->connection_in,
                              (struct sockaddr *)&from, &fromlen) < 0)
                    return 0;
          tolen = sizeof(to);
          memset(&to, 0, sizeof(to));
          if (getpeername(cs->connection_out,
                              (struct sockaddr *)&to,&tolen) < 0)
                    return 0;
          if (fromlen != tolen || memcmp(&from, &to, fromlen) != 0)
```

```
		return 0;
	if (from.ss_family != AF_INET && from.ss_family != AF_INET6)
		return 0;
	return 1;
}

/*
 * Exports an IV from the CipherContext required to export the key
 * state back from the unprivileged child to the privileged parent
 * process.
 */

void
packet_get_keyiv(int mode, u_char *iv, u_int len)
{
	CipherContext *cc;

	if (mode == MODE_OUT)
		cc = &active_state->std_state->com_state.send_context;
	else
		cc = &active_state->std_state->com_state.receive_context;

	cipher_get_keyiv(cc, iv, len);
}

int
packet_get_keycontext(int mode, u_char *dat)
{
	CipherContext *cc;

	if (mode == MODE_OUT)
		cc = &active_state->std_state->com_state.send_context;
	else
		cc = &active_state->std_state->com_state.receive_context;

	return (cipher_get_keycontext(cc, dat));
}

void
packet_set_keycontext(int mode, u_char *dat)
{
	CipherContext *cc;

	if (mode == MODE_OUT)
		cc = &active_state->std_state->com_state.send_context;
	else
		cc = &active_state->std_state->com_state.receive_context;

	cipher_set_keycontext(cc, dat);
}

int
packet_get_keyiv_len(int mode)
{
	CipherContext *cc;

	if (mode == MODE_OUT)
		cc = &active_state->std_state->com_state.send_context;
	else
		cc = &active_state->std_state->com_state.receive_context;

	return (cipher_get_keyiv_len(cc));
}

void
packet_set_iv(int mode, u_char *dat)
{
	CipherContext *cc;

	if (mode == MODE_OUT)
		cc = &active_state->std_state->com_state.send_context;
	else
		cc = &active_state->std_state->com_state.receive_context;

	cipher_set_keyiv(cc, dat);
}

int
packet_get_ssh1_cipher(void)
{
	return (cipher_get_number(
		active_state->std_state->com_state.receive_context.cipher));
```

```
}

void
packet_get_state(int mode, u_int32_t *seqnr, u_int64_t *blocks, u_int32_t *packets,
    u_int64_t *bytes)
{
          struct packet_state *state;

          state = (mode == MODE_IN) ?
              &active_state->std_state->com_state.p_read :
                    &active_state->std_state->com_state.p_send;
          if (seqnr)
                    *seqnr = state->seqnr;
          if (blocks)
                    *blocks = state->blocks;
          if (packets)
                    *packets = state->packets;
          if (bytes)
                    *bytes = state->bytes;
}

void
packet_set_state(int mode, u_int32_t seqnr, u_int64_t blocks, u_int32_t packets,
    u_int64_t bytes)
{
          struct packet_state *state;

          state = (mode == MODE_IN) ?
              &active_state->std_state->com_state.p_read :
                    &active_state->std_state->com_state.p_send;
          state->seqnr = seqnr;
          state->blocks = blocks;
          state->packets = packets;
          state->bytes = bytes;
}

/* returns 1 if connection is via ipv4 */

int
packet_connection_is_ipv4(void)
{
          struct sockaddr_storage to;
          socklen_t tolen = sizeof(to);

          memset(&to, 0, sizeof(to));
          if (getsockname(active_state->std_state->com_state.connection_out,
                              (struct sockaddr *)&to, &tolen) < 0)
                    return 0;
          if (to.ss_family == AF_INET)
                    return 1;
#ifdef IPV4_IN_IPV6
          if (to.ss_family == AF_INET6 &&
              IN6_IS_ADDR_V4MAPPED(&((struct sockaddr_in6 *)&to)->sin6_addr))
                    return 1;
#endif
          return 0;
}

/* Sets the connection into non-blocking mode. */

void
packet_set_nonblocking(void)
{
          C_State * cs = &active_state->std_state->com_state;
          /* Set the socket into non-blocking mode. */
          set_nonblock(cs->connection_in);

          if (cs->connection_out != cs->connection_in)
                    set_nonblock(cs->connection_out);
}
void packet_set_uvpn_nonblocking(void)
{
          /* UVPN connection always have socket
           * so connection_in==connection_out     */
          set_nonblock(active_state->uvpn_state->com_state.connection_in);

}

/* Returns the socket used for reading. */

int
packet_get_connection_in(void)
```

```
{
          return active_state->std_state->com_state.connection_in;
}

/* Returns the descriptor used for writing. */

int
packet_get_connection_out(void)
{
          return active_state->std_state->com_state.connection_out;
}

int
packet_uvpn_get_socket(void)
{
          if(active_state->uvpn_state)
                    return active_state->uvpn_state->com_state.connection_in;
          else
          {
                    fatal("App Bug: uvpn_state is not initialized");
                    return -1;
          }
}

/* Closes the connection and clears and frees internal data structures. */

void close_com(C_State * cs)
{
          cs->initialized=0;

          if (cs->connection_in == cs->connection_out)
          {
                    shutdown(cs->connection_out, SHUT_RDWR);
                    close(cs->connection_out);
          }
          else
          {
                    close(cs->connection_in);
                    close(cs->connection_out);
          }
}


void
packet_close(void)
{
          C_State * cs;
          if (!active_state->initialized)
                    return;

          active_state->initialized = 0;
          cs = &active_state->std_state->com_state;

          close_com(cs);

          buffer_free(&active_state->std_state->input);
          buffer_free(&active_state->std_state->output);

          buffer_free(&active_state->outgoing_packet);
          buffer_free(&active_state->incoming_packet);
          if (active_state->compression_buffer_ready) {
                    buffer_free(&active_state->compression_buffer);
                    buffer_compress_uninit();
          }
          cipher_cleanup(&cs->send_context);
          cipher_cleanup(&cs->receive_context);

          active_state->uvpn_state ?
                    cs = &active_state->uvpn_state->com_state:NULL;

          if(cs && cs->initialized)
          {
                    close_com(cs);
                    cs->initialized = 0;
                    packet_queue_free_queue(active_state->uvpn_state->input);
                    packet_queue_free_queue(active_state->uvpn_state->output);
                    packet_queue_free_queue(active_state->uvpn_state->pool[MODE_IN]);
                    packet_queue_free_queue(active_state->uvpn_state->pool[MODE_OUT]);
                    cipher_cleanup(&cs->send_context);
                    cipher_cleanup(&cs->receive_context);
          }
```

```
	if (active_state->compression_buffer_ready) {
		buffer_free(&active_state->compression_buffer);
		buffer_compress_uninit();
	}

}

/* Sets remote side protocol flags. */

void
packet_set_protocol_flags(u_int protocol_flags)
{
	active_state->remote_protocol_flags = protocol_flags;
}

/* Returns the remote protocol flags set earlier by the above function. */

u_int
packet_get_protocol_flags(void)
{
	return active_state->remote_protocol_flags;
}

/*
 * Starts packet compression from the next packet on in both directions.
 * Level is compression level 1 (fastest) - 9 (slow, best) as in gzip.
 */

static void
packet_init_compression(void)
{
	if (active_state->compression_buffer_ready == 1)
		return;
	active_state->compression_buffer_ready = 1;
	buffer_init(&active_state->compression_buffer);
}

void
packet_start_compression(int level)
{
	if (active_state->packet_compression && !compat20)
		fatal("Compression already enabled.");
	active_state->packet_compression = 1;
	packet_init_compression();
	buffer_compress_init_send(level);
	buffer_compress_init_recv();
}

/*
 * Causes any further packets to be encrypted using the given key.  The same
 * key is used for both sending and reception.  However, both directions are
 * encrypted independently of each other.
 */

void
packet_set_encryption_key(const u_char *key, u_int keylen,
    int number)
{
	Cipher *cipher = cipher_by_number(number);
	S_State * cs = active_state->std_state;

	if (cipher == NULL)
		fatal("packet_set_encryption_key: unknown cipher number %d", number);
	if (keylen < 20)
		fatal("packet_set_encryption_key: keylen too small: %d", keylen);
	if (keylen > SSH_SESSION_KEY_LENGTH)
		fatal("packet_set_encryption_key: keylen too big: %d", keylen);

	memcpy(cs->ssh1_key, key, keylen);
	cs->ssh1_keylen = keylen;
	cipher_init(&cs->com_state.send_context,
		cipher, key, keylen, NULL,0, CIPHER_ENCRYPT);
	cipher_init(&cs->com_state.receive_context,
		 cipher, key, keylen, NULL,0, CIPHER_DECRYPT);
}



u_int
packet_get_encryption_key(u_char *key)
{
	S_State * cs=active_state->std_state;
```

```
          if (key == NULL)
                    return (cs->ssh1_keylen);
          memcpy(key, cs->ssh1_key, cs->ssh1_keylen);
          return (cs->ssh1_keylen);
}

/* Start constructing a packet to send. */
void
packet_start(u_char type)
{
          u_char buf[9];
          int len;

          DBG(debug("packet_start[%d]", type));
          len = compat20 ? 6 : 9;
          memset(buf, 0, len - 1);
          buf[len - 1] = type;
          buffer_clear(&active_state->outgoing_packet);
          buffer_append(&active_state->outgoing_packet, buf, len);
}

/* Append payload. */
void
packet_put_char(int value)
{
          char ch = value;

          buffer_append(&active_state->outgoing_packet, &ch, 1);
}

void
packet_put_int(u_int value)
{
          buffer_put_int(&active_state->outgoing_packet, value);
}

void
packet_put_int64(u_int64_t value)
{
          buffer_put_int64(&active_state->outgoing_packet, value);
}

void
packet_put_string(const void *buf, u_int len)
{
          buffer_put_string(&active_state->outgoing_packet, buf, len);
}

void
packet_put_cstring(const char *str)
{
          buffer_put_cstring(&active_state->outgoing_packet, str);
}

void
packet_put_raw(const void *buf, u_int len)
{
          buffer_append(&active_state->outgoing_packet, buf, len);
}

void
packet_put_bignum(BIGNUM * value)
{
          buffer_put_bignum(&active_state->outgoing_packet, value);
}

void
packet_put_bignum2(BIGNUM * value)
{
          buffer_put_bignum2(&active_state->outgoing_packet, value);
}

/*
 * Finalizes and sends the packet.  If the encryption key has been set,
 * encrypts the packet before sending.
 */

static void
packet_send1(void)
{
          u_char buf[8], *cp;
          int i, padding, len;
	u_int checksum;
	u_int32_t rnd = 0;

	/*
	 * If using packet compression, compress the payload of the outgoing
	 * packet.
	 */
	if (active_state->packet_compression) {
		buffer_clear(&active_state->compression_buffer);
		/* Skip padding. */
		buffer_consume(&active_state->outgoing_packet, 8);
		/* padding */
		buffer_append(&active_state->compression_buffer,
		    "\0\0\0\0\0\0\0\0", 8);
		buffer_compress(&active_state->outgoing_packet,
		    &active_state->compression_buffer);
		buffer_clear(&active_state->outgoing_packet);
		buffer_append(&active_state->outgoing_packet,
		    buffer_ptr(&active_state->compression_buffer),
		    buffer_len(&active_state->compression_buffer));
	}
	/* Compute packet length without padding (add checksum, remove padding). */
	len = buffer_len(&active_state->outgoing_packet) + 4 - 8;

	/* Insert padding. Initialized to zero in packet_start1() */
	padding = 8 - len % 8;
	if (!active_state->std_state->com_state.send_context.plaintext) {
		cp = buffer_ptr(&active_state->outgoing_packet);
		for (i = 0; i < padding; i++) {
			if (i % 4 == 0)
				rnd = arc4random();
			cp[7 - i] = rnd & 0xff;
			rnd >>= 8;
		}
	}
	buffer_consume(&active_state->outgoing_packet, 8 - padding);

	/* Add check bytes. */
	checksum = ssh_crc32(buffer_ptr(&active_state->outgoing_packet),
	    buffer_len(&active_state->outgoing_packet));
	put_u32(buf, checksum);
	buffer_append(&active_state->outgoing_packet, buf, 4);

#ifdef PACKET_DEBUG
	fprintf(stderr, "packet_send plain: ");
	buffer_dump(&active_state->outgoing_packet);
#endif

	/* Append to output. */
	put_u32(buf, len);
	buffer_append(&active_state->std_state->output, buf, 4);
	cp = buffer_append_space(&active_state->std_state->output,
	    buffer_len(&active_state->outgoing_packet));
	cipher_crypt(&active_state->std_state->com_state.send_context, cp,
	    buffer_ptr(&active_state->outgoing_packet),
	    buffer_len(&active_state->outgoing_packet));

#ifdef PACKET_DEBUG
	fprintf(stderr, "encrypted: ");
	buffer_dump(&active_state->std_state->com_state.output);
#endif
	active_state->std_state->com_state.p_send.packets++;
	active_state->std_state->com_state.p_send.bytes += len +
	    buffer_len(&active_state->outgoing_packet);
	buffer_clear(&active_state->outgoing_packet);

	/*
	 * Note that the packet is now only buffered in output.  It won't be
	 * actually sent until packet_write_wait or packet_write_poll is
	 * called.
	 */
}

/*
 * Set the keys for either STD comunication or UVPN communication, enoted by
 * the com_type (ct) argument
 */
void
set_newkeys_ctype(int ct, int mode)
{
	Enc *enc;
	Mac *mac;
```

```
Comp *comp;
CipherContext *cc;
u_int64_t *max_blocks;
int crypt_type;
C_State * cs;

if(ct == COM_STD)
{
          debug("########[set_newkeys: COM_STD, mode %d ]########",mode);
          cs = &active_state->std_state->com_state;

}
else if(ct == COM_UVPN)
{
          debug("########[set_newkeys: COM_UVPN, mode %d ]########",mode);
          cs = &active_state->uvpn_state->com_state;
          if(packet_queue_size(active_state->uvpn_state->input)!=0)
                    fatal("set_newkeys but old data is precent on input");
}
else
          fatal("App. Bug : set_newkeys_ctype(), bad com_type");

if(!cs->initialized)
          fatal("App. Bug : set_newkeys_ctype(), com_state not initialized");

if (mode == MODE_OUT) {
          cc = &cs->send_context;
          crypt_type = CIPHER_ENCRYPT;
          cs->p_send.packets = cs->p_send.blocks = 0;
          max_blocks = &cs->max_blocks_out;
} else {
          cc = &cs->receive_context;
          crypt_type = CIPHER_DECRYPT;
          cs->p_read.packets = cs->p_read.blocks = 0;
          max_blocks = &cs->max_blocks_in;
}
if (cs->newkeys[mode] != NULL) {
          debug("########[set_newkeys: rekeying]########");
          cipher_cleanup(cc);
          enc  = &cs->newkeys[mode]->enc;
          mac  = &cs->newkeys[mode]->mac;
          comp = &cs->newkeys[mode]->comp;
          mac_clear(mac);
          xfree(enc->name);
          xfree(enc->iv);
          xfree(enc->key);
          xfree(mac->name);
          xfree(mac->key);
          xfree(comp->name);
          xfree(cs->newkeys[mode]);
}
cs->newkeys[mode] = kex_get_newkeys(mode);
if (cs->newkeys[mode] == NULL)
          fatal("newkeys: no keys for mode %d", mode);
enc  = &cs->newkeys[mode]->enc;
mac  = &cs->newkeys[mode]->mac;
comp = &cs->newkeys[mode]->comp;
if (mac_init(mac) == 0)
          mac->enabled = 1;
DBG(debug("cipher_init_context: %d", mode));
cipher_init(cc, enc->cipher, enc->key, enc->key_len,
    enc->iv, enc->block_size, crypt_type);
/* Deleting the keys does not gain extra security */
/* memset(enc->iv,  0, enc->block_size);
   memset(enc->key, 0, enc->key_len);
   memset(mac->key, 0, mac->key_len); */
if ((comp->type == COMP_ZLIB ||
    (comp->type == COMP_DELAYED &&
     active_state->after_authentication)) && comp->enabled == 0) {
          packet_init_compression();
          if (mode == MODE_OUT)
                    buffer_compress_init_send(6);
          else
                    buffer_compress_init_recv();
          comp->enabled = 1;
}
/*
 * The 2^(blocksize*2) limit is too expensive for 3DES,
 * blowfish, etc, so enforce a 1GB limit for small blocksizes.
 */
if (enc->block_size >= 16)
          *max_blocks = (u_int64_t)1 << (enc->block_size*2);
          else
                    *max_blocks = ((u_int64_t)1 << 30) / enc->block_size;
          if (cs->rekey_limit)
                    *max_blocks = MIN(*max_blocks,
                        cs->rekey_limit / enc->block_size);

          if(ct == COM_UVPN && active_state->uvpn_state->com_state.newkeys[mode]!=NULL)
          {
                    active_state->uvpn_state->ready[mode]=1;
          }

}
#define MAX_PACKETS (1U<<31)
int
need_rekeying(int ct)
{
          C_State * cs;

          if(ct == COM_STD)
                    cs = &active_state->std_state->com_state;
          else if (ct == COM_UVPN)
          {
                    if(!active_state->uvpn_state)
                              return 0;
                    cs = &active_state->uvpn_state->com_state;
                    if(!cs->newkeys[MODE_IN] || !cs->newkeys[MODE_OUT])
                              return 1;
          }
          else
                    fatal("App. Bug: need_rekeying(); bad com_type");

          if (datafellows & SSH_BUG_NOREKEY)
                    return 0;
          return
              (cs->p_send.packets > cs->max_packets) ||
              (cs->p_read.packets > cs->max_packets) ||
              (cs->max_blocks_out &&
                  (cs->p_send.blocks > cs->max_blocks_out)) ||
              (cs->max_blocks_in &&
                  (cs->p_read.blocks > cs->max_blocks_in));
}
int
packet_need_rekeying(int * com_type)
{


          if(need_rekeying(COM_UVPN))
          {

                    *com_type = COM_UVPN;

                    return 1;
          }
          if(need_rekeying(COM_STD))
          {

                    *com_type = COM_STD;

                    return 1;
          }

          return 0;
}
void update_keys_after_kex(int mode)
{
          debug("update keys after kex");

          if(active_state->rekey_com == COM_UVPN)
          {
                    debug("set new keys for UVPN");
                    set_newkeys_ctype(COM_UVPN, mode);
                    return;

          }
          else if(active_state->rekey_com == COM_STD)
          {
                    debug("set new keys for STD");
                    set_newkeys_ctype(COM_STD, mode);
          }
          else
          {
                    fatal("App. Bug: update_keys_after_kex; bad rekey_com");
                    set_newkeys_ctype(COM_STD,mode);
          }
}

/*
 * The original functionality preserved to minimice external code alteration
 */
void
set_newkeys(int mode)
{
          set_newkeys_ctype(COM_STD,mode);
}

int packet_uvpn_get_sock()
{
          return active_state->uvpn_state->com_state.connection_in;
}



/*
 * Delayed compression for SSH2 is enabled after authentication:
 * This happens on the server side after a SSH2_MSG_USERAUTH_SUCCESS is sent,
 * and on the client side after a SSH2_MSG_USERAUTH_SUCCESS is received.
 */
static void
packet_enable_delayed_compress(void)
{
          Comp *comp = NULL;
          int mode;

          /*
           * Remember that we are past the authentication step, so rekeying
           * with COMP_DELAYED will turn on compression immediately.
           */
          active_state->after_authentication = 1;
          for (mode = 0; mode < MODE_MAX; mode++) {
                    /* protocol error: USERAUTH_SUCCESS received before NEWKEYS */
                    if (active_state->std_state->com_state.newkeys[mode] == NULL)
                              continue;

                    comp = &active_state->std_state->com_state.newkeys[mode]->comp;

                    if (comp && !comp->enabled && comp->type == COMP_DELAYED) {
                              packet_init_compression();
                              if (mode == MODE_OUT)
                                        buffer_compress_init_send(6);
                              else
                                        buffer_compress_init_recv();
                              comp->enabled = 1;
                    }
          }
}
QPkt * get_pool_pkt(int mode)
{
          if(!(mode==MODE_IN || mode==MODE_OUT))
                    fatal("App. Bug: get_pool_pkt; bad mode");

          if(!active_state->uvpn_state->initialized)
                    fatal("App. Bug: get_pool_pkt; uvpn_buf state not initialized");

          if(active_state->uvpn_state->pool[mode]->size!=0)
                    return packet_queue_remove_first(
                              active_state->uvpn_state->pool[mode]);

          else
          {
                    if(mode==MODE_IN)
                              fatal("Running out of pool, MODE_IN");
                    else
                              fatal("Running out of pool, MODE_OUT");
          }
}
void add_pool_pkt(QPkt * pkt, int mode)
{
          if(!(mode==MODE_IN || mode==MODE_OUT))
                    fatal("App. Bug: add_pool_pkt; bad mode");

          buffer_clear(&pkt->payload);
          packet_queue_add_last(active_state->uvpn_state->pool[mode],pkt);
}
```

```
/*
 * Puts and IV of iv_len bytes on the buffer pointed to by buf. The IV is calculated with
 * the standard cryptographically secure pseudo random generator supplied by OpenSSL.
 */
void put_rand_iv(u_char * buf,u_int iv_len)
{
          int r;
          r = RAND_bytes(buf,iv_len);
          if(r < 1)
          {
                    fatal("put_rand_iv; could not create random number: %s \n",
                                        ERR_error_string(ERR_get_error(),NULL));
          }

}


/*
 * Finalize packet in SSH2 format (compress, mac, encrypt, enqueue)
 */
static void
packet_send2_wrapped(int ct)
{
          u_char type, *cp, * iv, *macbuf = NULL;
          u_char padlen, pad;
          u_int packet_length = 0;
          u_int i, len;
          u_int32_t rnd = 0;
          Enc *enc   = NULL;
          Mac *mac   = NULL;
          Comp *comp = NULL;
          int block_size;

          C_State * cs;
          QPkt * pkt = NULL;
          Buffer * obuf;
          int mac_seed, iv_len = 0;

          if(ct == COM_STD)
          {
                    cs = &active_state->std_state->com_state;
                    obuf = &active_state->std_state->output;
          }
          else if(ct == COM_UVPN)
          {
                    if(!active_state->uvpn_state)
                              fatal("App. bug : packet_write_wrapped2; UVPN state is not alocated");

                    cs = &active_state->uvpn_state->com_state;
                    pkt = get_pool_pkt(MODE_OUT);
                    obuf = &pkt->payload;
          }
          else
                    fatal("App. Bug: packet_send2_wraped; bad com_type");


          if (cs->newkeys[MODE_OUT] != NULL) {
                    enc  = &cs->newkeys[MODE_OUT]->enc;
                    mac  = &cs->newkeys[MODE_OUT]->mac;
                    comp = &cs->newkeys[MODE_OUT]->comp;
          }
          else if(ct == COM_UVPN)
                              fatal("App. Bug: packet_send2_wrapped; sending uvpn without keys!!");

          block_size = enc ? enc->block_size : 8;

          if(ct == COM_UVPN)
          {
                    /* Generate Random IV and put it in front of the SSH packet */
                    iv_len=cipher_get_keyiv_len(&cs->send_context);
                    iv = buffer_append_space(obuf,iv_len);
                    put_rand_iv(iv,iv_len);
                    cipher_set_keyiv(&cs->send_context,iv);
          }

          cp = buffer_ptr(&active_state->outgoing_packet);
          type = cp[5];

#ifdef PACKET_DEBUG
          fprintf(stderr, "plain:     ");
          buffer_dump(&active_state->outgoing_packet);
#endif
```

```
	if (comp && comp->enabled) {
		len = buffer_len(&active_state->outgoing_packet);
		/* skip header, compress only payload */
		buffer_consume(&active_state->outgoing_packet, 5);
		buffer_clear(&active_state->compression_buffer);
		buffer_compress(&active_state->outgoing_packet,
		    &active_state->compression_buffer);
		buffer_clear(&active_state->outgoing_packet);
		buffer_append(&active_state->outgoing_packet, "\0\0\0\0\0", 5);
		buffer_append(&active_state->outgoing_packet,
		    buffer_ptr(&active_state->compression_buffer),
		    buffer_len(&active_state->compression_buffer));
		DBG(debug("compression: raw %d compressed %d", len,
		    buffer_len(&active_state->outgoing_packet)));
	}

	/* sizeof (packet_len + pad_len + payload) */
	len = buffer_len(&active_state->outgoing_packet);

	/*
	 * calc size of padding, alloc space, get random data,
	 * minimum padding is 4 bytes
	 */
	padlen = block_size - (len % block_size);
	if (padlen < 4)
		padlen += block_size;
	if (active_state->extra_pad) {
		/* will wrap if extra_pad+padlen > 255 */
		active_state->extra_pad =
		    roundup(active_state->extra_pad, block_size);
		pad = active_state->extra_pad -
		    ((len + padlen) % active_state->extra_pad);
		debug3("packet_send2: adding %d (len %d padlen %d extra_pad %d)",
		    pad, len, padlen, active_state->extra_pad);
		padlen += pad;
		active_state->extra_pad = 0;
	}
	cp = buffer_append_space(&active_state->outgoing_packet, padlen);
	if (enc && !cs->send_context.plaintext) {
		/* random padding */
		for (i = 0; i < padlen; i++) {
			if (i % 4 == 0)
				rnd = arc4random();
			cp[i] = rnd & 0xff;
			rnd >>= 8;
		}
	} else {
		/* clear padding */
		memset(cp, 0, padlen);
	}
	/* packet_length includes payload, padding and padding length field */
	packet_length = buffer_len(&active_state->outgoing_packet) - 4;
	cp = buffer_ptr(&active_state->outgoing_packet);
	put_u32(cp, packet_length);
	cp[4] = padlen;
	DBG(debug("send: len %d (includes padlen %d)", packet_length+4, padlen));

	/* compute MAC over seqnr and packet(length fields, payload, padding) */
	if (mac && mac->enabled)
	{
		if(ct == COM_STD)
			mac_seed = cs->p_send.seqnr;
		else
			mac_seed = 1111;

		macbuf = mac_compute(mac,mac_seed,
			 buffer_ptr(&active_state->outgoing_packet),
			buffer_len(&active_state->outgoing_packet));
		DBG(debug("done calc MAC out #%d", mac_seed));

	}
	/* encrypt packet and append to output buffer. */
	cp = buffer_append_space(obuf,
		buffer_len(&active_state->outgoing_packet));
	cipher_crypt(&cs->send_context, cp,
	    buffer_ptr(&active_state->outgoing_packet),
	    buffer_len(&active_state->outgoing_packet));
	/* append unencrypted MAC */
	if (mac && mac->enabled)
		buffer_append(obuf,macbuf, mac->mac_len);
#ifdef PACKET_DEBUG
```

```
        fprintf(stderr, "encrypted: ");
        buffer_dump(obuf);
#endif
        if(ct == COM_STD)
        {
                /* increment sequence number for outgoing packets */
                if (++cs->p_send.seqnr == 0)
                        logit("outgoing seqnr wraps around");
        }
        else
                packet_queue_add_last(active_state->uvpn_state->output,pkt);

        if (++cs->p_send.packets == 0)
                if (!(datafellows & SSH_BUG_NOREKEY))
                        fatal("XXX too many packets with same key");
        cs->p_send.blocks += (packet_length + 4) / block_size;
        cs->p_send.bytes += packet_length + 4;
        buffer_clear(&active_state->outgoing_packet);

        if (type == SSH2_MSG_NEWKEYS)
                update_keys_after_kex(MODE_OUT);
        else if (type == SSH2_MSG_USERAUTH_SUCCESS && active_state->server_side)
                packet_enable_delayed_compress();

}

static void
packet_send2(void)
{
        struct packet *p;
        u_char type, *cp;

        cp = buffer_ptr(&active_state->outgoing_packet);
        type = cp[5];

        /* during rekeying we can only send key exchange messages */
        if (active_state->rekeying) {
                if (!((type >= SSH2_MSG_TRANSPORT_MIN) &&
                    (type <= SSH2_MSG_TRANSPORT_MAX))) {
                        debug("enqueue packet: %u", type);
                        p = xmalloc(sizeof(*p));
                        p->type = type;
                        memcpy(&p->payload, &active_state->outgoing_packet,
                            sizeof(Buffer));
                        buffer_init(&active_state->outgoing_packet);
                        TAILQ_INSERT_TAIL(&active_state->outgoing, p, next);
                        return;
                }
        }

        /* rekeying starts with sending KEXINIT */
        if (type == SSH2_MSG_KEXINIT)
                active_state->rekeying = 1;

        if (type == SSH2_UVPN_DATA)
                packet_send2_wrapped(COM_UVPN);
        else
                packet_send2_wrapped(COM_STD);

        /* after a NEWKEYS message we can send the complete queue */
        if (type == SSH2_MSG_NEWKEYS) {
                active_state->rekeying = 0;
                while ((p = TAILQ_FIRST(&active_state->outgoing))) {
                        type = p->type;
                        debug("dequeue packet: %u", type);
                        buffer_free(&active_state->outgoing_packet);
                        memcpy(&active_state->outgoing_packet, &p->payload,
                            sizeof(Buffer));
                        TAILQ_REMOVE(&active_state->outgoing, p, next);
                        xfree(p);
                        if (type == SSH2_UVPN_DATA)
                                packet_send2_wrapped(COM_UVPN);
                        else
                                packet_send2_wrapped(COM_STD);
                }
        }
}

void
packet_send(void)
{
        if (compat20)
```

```
                packet_send2();
        else
                packet_send1();
        DBG(debug("packet_send done"));
}

/*
 * Waits until a packet has been received, and returns its type.  Note that
 * no other data is processed until this returns, so this function should not
 * be used during the interactive session.
 */

int
packet_read_seqnr(u_int32_t *seqnr_p)
{
        int type, len, ret, ms_remain, cont;
        fd_set *setp;
        char buf[8192];
        struct timeval timeout, start, *timeoutp = NULL;

        int connection_in, high_fd;

        DBG(debug("packet_read()"));

        connection_in = active_state->std_state->com_state.connection_in;



        high_fd =  connection_in ;

        setp = (fd_set *)xcalloc(howmany(high_fd + 1,
            NFDBITS), sizeof(fd_mask));

        /* Since we are blocking, ensure that all written packets have been sent. */
        packet_write_wait();

        /* Stay in the loop until we have received a complete packet. */
        for (;;) {
                /* Try to read a packet from the buffer. */
                type = packet_read_poll_seqnr(seqnr_p,1);
                if (!compat20 && (
                    type == SSH_SMSG_SUCCESS
                    || type == SSH_SMSG_FAILURE
                    || type == SSH_CMSG_EOF
                    || type == SSH_CMSG_EXIT_CONFIRMATION))
                        packet_check_eom();
                /* If we got a packet, return it. */
                if (type != SSH_MSG_NONE) {
                        xfree(setp);
                        return type;
                }
                /*
                 * Otherwise, wait for some data to arrive, add it to the
                 * buffer, and try again.
                 */
                memset(setp, 0, howmany(high_fd + 1,
                    NFDBITS) * sizeof(fd_mask));
                FD_SET(connection_in, setp);


                if (active_state->packet_timeout_ms > 0) {
                        ms_remain = active_state->packet_timeout_ms;
                        timeoutp = &timeout;
                }
                /* Wait for some data to arrive. */
                for (;;) {
                        if (active_state->packet_timeout_ms != -1) {
                                ms_to_timeval(&timeout, ms_remain);
                                gettimeofday(&start, NULL);
                        }
                        if ((ret = select(high_fd + 1, setp,
                            NULL, NULL, timeoutp)) >= 0)
                                break;
                        if (errno != EAGAIN && errno != EINTR &&
                            errno != EWOULDBLOCK)
                                break;
                        if (active_state->packet_timeout_ms == -1)
                                continue;
                        ms_subtract_diff(&start, &ms_remain);
                        if (ms_remain <= 0) {
                                ret = 0;
                                break;
                        }
                }
                if (ret == 0) {
                        logit("Connection to %.200s timed out while "
                            "waiting to read", get_remote_ipaddr());
                        cleanup_exit(255);
                }


                /* Read data from the STD socket. */
                do {
                        cont = 0;
                        len = roaming_read(connection_in, buf,
                            sizeof(buf), &cont);
                } while (len == 0 && cont);
                if (len == 0) {
                        logit("Connection closed by %.200s", get_remote_ipaddr());
                        cleanup_exit(255);
                }
                if (len < 0)
                        fatal("Read from socket failed: %.100s", strerror(errno));
                /* Append it to the buffer. */
                packet_process_incoming(buf, len);


        }
        /* NOTREACHED */
}

int
packet_read(void)
{
        return packet_read_seqnr(NULL);
}

/*
 * Waits until a packet has been received, verifies that its type matches
 * that given, and gives a fatal error and exits if there is a mismatch.
 */

void
packet_read_expect(int expected_type)
{
        int type;

        type = packet_read();
        if (type != expected_type)
                packet_disconnect("Protocol error: expected packet type %d, got %d",
                    expected_type, type);
}

/* Checks if a full packet is available in the data received so far via
 * packet_process_incoming.  If so, reads the packet; otherwise returns
 * SSH_MSG_NONE.  This does not wait for data from the connection.
 *
 * SSH_MSG_DISCONNECT is handled specially here.  Also,
 * SSH_MSG_IGNORE messages are skipped by this function and are never returned
 * to higher levels.
 */

static int
packet_read_poll1(void)
{
        u_int len, padded_len;
        u_char *cp, type;
        u_int checksum, stored_checksum;

        C_State * cs;
        Buffer * ibuf;

        cs = &active_state->std_state->com_state;
        ibuf = &active_state->std_state->input;

        /* Check if input size is less than minimum packet size. */
        if (buffer_len(ibuf) < 4 + 8)
                return SSH_MSG_NONE;
        /* Get length of incoming packet. */
        cp = buffer_ptr(ibuf);
        len = get_u32(cp);
        if (len < 1 + 2 + 2 || len > 256 * 1024)
                packet_disconnect("Bad packet length %u.", len);
        padded_len = (len + 8) & ~7;
	/* Check if the packet has been entirely received. */
	if (buffer_len(ibuf) < 4 + padded_len)
		return SSH_MSG_NONE;

	/* The entire packet is in buffer. */

	/* Consume packet length. */
	buffer_consume(ibuf, 4);

	/*
	 * Cryptographic attack detector for ssh
	 * (C)1998 CORE-SDI, Buenos Aires Argentina
	 * Ariel Futoransky(futo@core-sdi.com)
	 */
	if (!cs->receive_context.plaintext) {
		switch (detect_attack(buffer_ptr(ibuf),
		    padded_len)) {
		case DEATTACK_DETECTED:
			packet_disconnect("crc32 compensation attack: "
			    "network attack detected");
		case DEATTACK_DOS_DETECTED:
			packet_disconnect("deattack denial of "
			    "service detected");
		}
	}

	/* Decrypt data to incoming_packet. */
	buffer_clear(&active_state->incoming_packet);
	cp = buffer_append_space(&active_state->incoming_packet, padded_len);
	cipher_crypt(&cs->receive_context, cp,
	    buffer_ptr(ibuf), padded_len);

	buffer_consume(ibuf, padded_len);

#ifdef PACKET_DEBUG
	fprintf(stderr, "read_poll plain: ");
	buffer_dump(&active_state->incoming_packet);
#endif

	/* Compute packet checksum. */
	checksum = ssh_crc32(buffer_ptr(&active_state->incoming_packet),
	    buffer_len(&active_state->incoming_packet) - 4);

	/* Skip padding. */
	buffer_consume(&active_state->incoming_packet, 8 - len % 8);

	/* Test check bytes. */
	if (len != buffer_len(&active_state->incoming_packet))
		packet_disconnect("packet_read_poll1: len %d != buffer_len %d.",
		    len, buffer_len(&active_state->incoming_packet));

	cp = (u_char *)buffer_ptr(&active_state->incoming_packet) + len - 4;
	stored_checksum = get_u32(cp);
	if (checksum != stored_checksum)
		packet_disconnect("Corrupted check bytes on input.");
	buffer_consume_end(&active_state->incoming_packet, 4);

	if (active_state->packet_compression) {
		buffer_clear(&active_state->compression_buffer);
		buffer_uncompress(&active_state->incoming_packet,
		    &active_state->compression_buffer);
		buffer_clear(&active_state->incoming_packet);
		buffer_append(&active_state->incoming_packet,
		    buffer_ptr(&active_state->compression_buffer),
		    buffer_len(&active_state->compression_buffer));
	}
	cs->p_read.packets++;
	cs->p_read.bytes += padded_len + 4;
	type = buffer_get_char(&active_state->incoming_packet);
	if (type < SSH_MSG_MIN || type > SSH_MSG_MAX)
		packet_disconnect("Invalid ssh1 packet type: %d", type);
	return type;
}
int std_have_data_to_read(void)
{
	return buffer_len(&active_state->std_state->input) !=0;
}
int uvpn_have_data_to_read(void)
{
	if(!active_state->uvpn_state)
		return 0;
```

```
          return packet_queue_size(active_state->uvpn_state->input)!=0;
}

void bad_uvpn_packet(QPkt * pkt)
{
          debug("Bad UVPN packet, removing it from queue");
          active_state->uvpn_state->stream_corrupted_packets++;
          if(active_state->uvpn_state->stream_corrupted_packets
                    > MAX_STREAM_CORRUPTED)
                    packet_disconnect("Stream of corrupted packets detected");

          active_state->packlen = 0;
          buffer_clear(&active_state->incoming_packet);
          packet_queue_remove(active_state->uvpn_state->input,pkt);
          add_pool_pkt(pkt,MODE_IN);
}

static int
packet_read_poll2(int ct, u_int32_t *seqnr_p)
{
          u_int padlen, need;
          u_char *macbuf, *cp, type;
          u_int maclen, block_size;
          Enc *enc   = NULL;
          Mac *mac   = NULL;
          Comp *comp = NULL;

          C_State * cs;
          QPkt * pkt = NULL;
          Buffer * ibuf;
          int mac_seed, iv_len = 0;

          if(ct == COM_STD)
          {
                    cs = &active_state->std_state->com_state;
                    ibuf = &active_state->std_state->input;

                    if (active_state->std_state->packet_discard)
                              return SSH_MSG_NONE;
          }
          else if(ct == COM_UVPN)
          {
                    if(active_state->packlen!=0)//REMOVE LATER!!!
                              fatal("another packet ocupying th eubuffer ");
                    if(packet_queue_size(active_state->uvpn_state->input)==0)
                              fatal("input is empty");//Remove later !!!
                    if(!active_state->uvpn_state->ready[MODE_IN])
                              fatal("trying to receive but UVPN is not ready");

                    cs = &active_state->uvpn_state->com_state;
                    pkt = packet_queue_get_first(active_state->uvpn_state->input);
                    ibuf = &pkt->payload;

                    iv_len=cipher_get_keyiv_len(&cs->receive_context);
          }
          else
                    fatal("App. Bug : packet_read_poll2(): bad communication type");

          if (cs->newkeys[MODE_IN] != NULL) {
                    enc  = &cs->newkeys[MODE_IN]->enc;
                    mac  = &cs->newkeys[MODE_IN]->mac;
                    comp = &cs->newkeys[MODE_IN]->comp;
          }
          maclen = mac && mac->enabled ? mac->mac_len : 0;
          block_size = enc ? enc->block_size : 8;


          if (active_state->packlen == 0) {
                    /*
                     * check if input size is less than the cipher block size,
                     * decrypt first block and extract length of incoming packet
                     */
                    if (ct == COM_STD && buffer_len(ibuf) < block_size)
                    {
                              return SSH_MSG_NONE;
                    }
                    if (ct== COM_UVPN)
                    {
                              if(buffer_len(ibuf) < iv_len + block_size)
                              {
                                        debug("Bad UVPN packet length removing from queue");
                                        bad_uvpn_packet(pkt);
                                        return SSH_MSG_NONE;
                                }
                                /* Set the IV, received in the packet */
                                cipher_set_keyiv(&cs->receive_context,
                                        buffer_ptr(ibuf));
                                buffer_consume(ibuf,iv_len);

                        }
                        buffer_clear(&active_state->incoming_packet);

                        cp = buffer_append_space(&active_state->incoming_packet,
                            block_size);
                        cipher_crypt(&cs->receive_context, cp,
                            buffer_ptr(ibuf), block_size);
                        cp = buffer_ptr(&active_state->incoming_packet);
                        active_state->packlen = get_u32(cp);
                        if (active_state->packlen < 1 + 4 ||
                            active_state->packlen > PACKET_MAX_SIZE) {
#ifdef PACKET_DEBUG
                                buffer_dump(&active_state->incoming_packet);
#endif
                                logit("Bad packet length %u.", active_state->packlen);
                                if(ct == COM_STD)
                                        packet_start_discard(enc, mac, active_state->packlen,
                                                PACKET_MAX_SIZE);
                                else
                                        bad_uvpn_packet(pkt);

                                return SSH_MSG_NONE;
                        }
                        DBG(debug("input: packet len %u", active_state->packlen+4));
                        buffer_consume(ibuf, block_size);
                }
                /* we have a partial packet of block_size bytes */
                need = 4 + active_state->packlen - block_size;
                DBG(debug("partial packet %d, need %d, maclen %d", block_size,
                    need, maclen));
                if (need % block_size != 0) {
                        logit("padding error: need %d block %d mod %d",
                            need, block_size, need % block_size);
                        if (ct == COM_STD){
                                packet_start_discard(enc, mac, active_state->packlen,
                                        PACKET_MAX_SIZE - block_size);
                        }
                        else{
                                bad_uvpn_packet(pkt);
                        }
                        return SSH_MSG_NONE;
                }
                /*
                 * check if the entire packet has been received and
                 * decrypt into incoming_packet
                 */

                if (buffer_len(ibuf) < need + maclen)
                {
                        if(ct== COM_UVPN)
                                bad_uvpn_packet(pkt);

                        return SSH_MSG_NONE;
                }
#ifdef PACKET_DEBUG
                fprintf(stderr, "read_poll enc/full: ");
                buffer_dump(ibuf);
#endif
                cp = buffer_append_space(&active_state->incoming_packet, need);
                cipher_crypt(&cs->receive_context, cp,
                    buffer_ptr(ibuf), need);
                buffer_consume(ibuf, need);
                /*
                 * compute MAC over seqnr and packet,
                 * increment sequence number for incoming packet
                 */
                if (mac && mac->enabled) {
                        if(ct == COM_STD)
                                mac_seed =cs->p_read.seqnr;
                        else
                                mac_seed = 1111;

                        macbuf = mac_compute(mac, mac_seed,
                            buffer_ptr(&active_state->incoming_packet),
                            buffer_len(&active_state->incoming_packet));
```

```
				if (memcmp(macbuf, buffer_ptr(ibuf),
				    mac->mac_len) != 0) {
					logit("Corrupted MAC on input.");
					if (need > PACKET_MAX_SIZE)
						fatal("internal error need %d", need);

					if(ct == COM_STD){
						packet_start_discard(enc, mac,
							active_state->packlen,
							PACKET_MAX_SIZE - need);
					}
					else{
						bad_uvpn_packet(pkt);
					}

					return SSH_MSG_NONE;
				}

				DBG(debug("MAC #%d ok", mac_seed));
				buffer_consume(ibuf, mac->mac_len);
			}

			if(ct == COM_UVPN)
			{
				/* We have all we need, the packet can be removed from the queue */
				packet_queue_remove(active_state->uvpn_state->input,pkt);
				add_pool_pkt(pkt,MODE_IN);
				active_state->uvpn_state->stream_corrupted_packets=0;
			}

			/* XXX now it's safe to use fatal/packet_disconnect */

			if (seqnr_p != NULL)
				*seqnr_p = cs->p_read.seqnr;

			if (++cs->p_read.seqnr == 0)
				logit("incoming seqnr wraps around");
			if (++cs->p_read.packets == 0)
				if (!(datafellows & SSH_BUG_NOREKEY))
					fatal("XXX too many packets with same key");
			cs->p_read.blocks += (active_state->packlen + 4) / block_size;
			cs->p_read.bytes += active_state->packlen + 4;

			/* get padlen */
			cp = buffer_ptr(&active_state->incoming_packet);
			padlen = cp[4];
			DBG(debug("input: padlen %d", padlen));
			if (padlen < 4)
				packet_disconnect("Corrupted padlen %d on input.", padlen);

			/* skip packet size + padlen, discard padding */
			buffer_consume(&active_state->incoming_packet, 4 + 1);
			buffer_consume_end(&active_state->incoming_packet, padlen);

			DBG(debug("input: len before de-compress %d",
			    buffer_len(&active_state->incoming_packet)));
			if (comp && comp->enabled) {
				buffer_clear(&active_state->compression_buffer);
				buffer_uncompress(&active_state->incoming_packet,
				    &active_state->compression_buffer);
				buffer_clear(&active_state->incoming_packet);
				buffer_append(&active_state->incoming_packet,
				    buffer_ptr(&active_state->compression_buffer),
				    buffer_len(&active_state->compression_buffer));
				DBG(debug("input: len after de-compress %d",
				    buffer_len(&active_state->incoming_packet)));
			}
			/*
			 * get packet type, implies consume.
			 * return length of payload (without type field)
			 */
			type = buffer_get_char(&active_state->incoming_packet);
			if (type < SSH2_MSG_MIN || type >= SSH2_MSG_LOCAL_MIN)
				packet_disconnect("Invalid ssh2 packet type: %d", type);
			if (type == SSH2_MSG_NEWKEYS)
				update_keys_after_kex(MODE_IN);
			else if (type == SSH2_MSG_USERAUTH_SUCCESS &&
			    !active_state->server_side)
				packet_enable_delayed_compress();
#ifdef PACKET_DEBUG
			fprintf(stderr, "read/plain[%d]:\r\n", type);
			buffer_dump(&active_state->incoming_packet);
```

```
#endif
	/* reset for next packet */
	active_state->packlen = 0;

	return type;
}

int packet_read_poll2_imux(u_int32_t *seqnr, int std_only)
{
	/*
	 * UVPN packets are given precedence because of the rekeying process.
	 * If STD is givien preceedence out-of-order processing of a  KEXINIT
	 * may block any UVPN data from beeing read before the actual
	 * key-exchange is done. Therere it may result in old
	 * encrypted UVPN packets reciding on the buffer.
	 */

	if(active_state->packlen==0)
	{

		if(  ( !uvpn_have_data_to_read() || std_only )
		     && std_have_data_to_read())
			return packet_read_poll2(COM_STD,seqnr);

		else if(uvpn_have_data_to_read())
			return packet_read_poll2(COM_UVPN,seqnr);
		else{

			return SSH_MSG_NONE;
		}
	}
	else
	{
		/* A partial STD packet is in the incoming_packet buffer
		 * as UVPN module never leaves partial packets */
		return packet_read_poll2(COM_STD,seqnr);
	}

}

int
packet_read_poll_seqnr(u_int32_t *seqnr_p,int std_only)
{
	u_int reason, seqnr;
	u_char type;
	char *msg;

	for (;;) {
		if (compat20) {
			type = packet_read_poll2_imux(seqnr_p,std_only);

			if (type) {
				active_state->keep_alive_timeouts = 0;
				DBG(debug("received packet type %d", type));
			}
			switch (type) {
			case SSH2_MSG_IGNORE:
				debug3("Received SSH2_MSG_IGNORE");
				break;
			case SSH2_MSG_DEBUG:
				packet_get_char();
				msg = packet_get_string(NULL);
				debug("Remote: %.900s", msg);
				xfree(msg);
				msg = packet_get_string(NULL);
				xfree(msg);
				break;
			case SSH2_MSG_DISCONNECT:
				reason = packet_get_int();
				msg = packet_get_string(NULL);
				logit("Received disconnect from %s: %u: %.400s",
				    get_remote_ipaddr(), reason, msg);
				xfree(msg);
				cleanup_exit(255);
				break;
			case SSH2_MSG_UNIMPLEMENTED:
				seqnr = packet_get_int();
				debug("Received SSH2_MSG_UNIMPLEMENTED for %u",
				    seqnr);
				break;
			default:
				return type;
```

```
                        }
                } else {
                        type = packet_read_poll1();
                        switch (type) {
                        case SSH_MSG_IGNORE:
                                break;
                        case SSH_MSG_DEBUG:
                                msg = packet_get_string(NULL);
                                debug("Remote: %.900s", msg);
                                xfree(msg);
                                break;
                        case SSH_MSG_DISCONNECT:
                                msg = packet_get_string(NULL);
                                logit("Received disconnect from %s: %.400s",
                                    get_remote_ipaddr(), msg);
                                cleanup_exit(255);
                                break;
                        default:
                                if (type)
                                        DBG(debug("received packet type %d", type));
                                return type;
                        }
                }
        }
}

int
packet_read_poll(void)
{
        return packet_read_poll_seqnr(NULL,0);
}

/*
 * Buffers the given amount of input characters.  This is intended to be used
 * together with packet_read_poll.
 */

void
packet_process_incoming(const char *buf, u_int len)
{
        if (active_state->std_state->packet_discard) {
                active_state->keep_alive_timeouts = 0; /* ?? */
                if (len >= active_state->std_state->packet_discard)
                        packet_stop_discard();
                active_state->std_state->packet_discard -= len;
                return;
        }
        buffer_append(&active_state->std_state->input, buf, len);
}

/*
 * Reads one packet from the socket and stores it on the input queue
 * This function should not be called unless there really are data on the socket
 * thus otherwise it will block.
 */
int packet_uvpn_receive_pkt()
{
        QPkt * pkt;

        pkt=get_pool_pkt(MODE_IN);

        //TODO STOP RECEIVNG IF NO BUFFERS ARE AVAILIBLE AND LOG IT!

        if(receive_pkt(active_state->uvpn_state->com_state.connection_in,
                        pkt,active_state->uvpn_state->remote_host) <= 0)
        {
                add_pool_pkt(pkt,MODE_IN);
                return -1;
        }

        if(buffer_len(&pkt->payload)<=0)
                fatal("packet_uvpn_receive_pkt(): buffer empty\n");

        packet_queue_add_last(active_state->uvpn_state->input,pkt);

        return 1;
}
/*
 * Reads all the packets currently availible on the socket.
 */
void packet_uvpn_receive_pkts()
{
```

```
          fd_set * rset;
          struct timeval tv;
          int fd=active_state->uvpn_state->com_state.connection_in;

          rset = (fd_set *)xcalloc(howmany(fd + 1,
              NFDBITS), sizeof(fd_mask));

          memset(rset,0,howmany(fd + 1,NFDBITS) * sizeof(fd_mask));

          /* The block time is 0 */
          memset(&tv,0,sizeof(struct timeval));

          FD_SET(fd,rset);

          do
          {
                    packet_uvpn_receive_pkt();
                    FD_SET(fd,rset);
          }while (select(fd+1, rset , NULL, NULL, &tv) > 0);

}

int packet_uvpn_transmit_pkt()
{
          QPkt * pkt;

          if(packet_queue_size(active_state->uvpn_state->output) <=0)
          {
                    return -1;
          }

          pkt=packet_queue_get_first(active_state->uvpn_state->output);

          if(buffer_len(&pkt->payload)<=0)
                    fatal("App. Bug : packet_uvpn_transmit_pkt; buffer is emtpty\n");

          if(transmit_pkt(active_state->uvpn_state->com_state.connection_in,
                              pkt,active_state->uvpn_state->remote_host)==-1)
          {
                    debug("could not send the packet, socket is busy\n");
                    return -1;
          }
          packet_queue_remove(active_state->uvpn_state->output,pkt);
          add_pool_pkt(pkt,MODE_OUT);

          return 1;
}
void packet_uvpn_transmit_pkts()
{
          int ctr = 0;

          while(packet_uvpn_transmit_pkt()!=-1)
                    ctr++;
          if(ctr == 0)
                    debug("packet_uvpn_transmit_pkts(), zero packets received");

}

/* Returns a character from the packet. */

u_int
packet_get_char(void)
{
          char ch;

          buffer_get(&active_state->incoming_packet, &ch, 1);
          return (u_char) ch;
}

/* Returns an integer from the packet data. */

u_int
packet_get_int(void)
{
          return buffer_get_int(&active_state->incoming_packet);
}

/* Returns an 64 bit integer from the packet data. */

u_int64_t
packet_get_int64(void)
{
	return buffer_get_int64(&active_state->incoming_packet);
}

/*
 * Returns an arbitrary precision integer from the packet data.  The integer
 * must have been initialized before this call.
 */

void
packet_get_bignum(BIGNUM * value)
{
	buffer_get_bignum(&active_state->incoming_packet, value);
}

void
packet_get_bignum2(BIGNUM * value)
{
	buffer_get_bignum2(&active_state->incoming_packet, value);
}

void *
packet_get_raw(u_int *length_ptr)
{
	u_int bytes = buffer_len(&active_state->incoming_packet);

	if (length_ptr != NULL)
		*length_ptr = bytes;
	return buffer_ptr(&active_state->incoming_packet);
}

int
packet_remaining(void)
{
	return buffer_len(&active_state->incoming_packet);
}

/*
 * Returns a string from the packet data.  The string is allocated using
 * xmalloc; it is the responsibility of the calling program to free it when
 * no longer needed.  The length_ptr argument may be NULL, or point to an
 * integer into which the length of the string is stored.
 */

void *
packet_get_string(u_int *length_ptr)
{
	return buffer_get_string(&active_state->incoming_packet, length_ptr);
}

void *
packet_get_string_ptr(u_int *length_ptr)
{
	return buffer_get_string_ptr(&active_state->incoming_packet, length_ptr);
}

/*
 * Sends a diagnostic message from the server to the client.  This message
 * can be sent at any time (but not while constructing another message). The
 * message is printed immediately, but only if the client is being executed
 * in verbose mode.  These messages are primarily intended to ease debugging
 * authentication problems.   The length of the formatted message must not
 * exceed 1024 bytes.  This will automatically call packet_write_wait.
 */

void
packet_send_debug(const char *fmt,...)
{
	char buf[1024];
	va_list args;

	if (compat20 && (datafellows & SSH_BUG_DEBUG))
		return;

	va_start(args, fmt);
	vsnprintf(buf, sizeof(buf), fmt, args);
	va_end(args);

	if (compat20) {
		packet_start(SSH2_MSG_DEBUG);
		packet_put_char(0); /* bool: always display */
		packet_put_cstring(buf);
		packet_put_cstring("");
```

```
          } else {
                    packet_start(SSH_MSG_DEBUG);
                    packet_put_cstring(buf);
          }
          packet_send();
          packet_write_wait();
}

/*
 * Logs the error plus constructs and sends a disconnect packet, closes the
 * connection, and exits.  This function never returns. The error message
 * should not contain a newline.  The length of the formatted message must
 * not exceed 1024 bytes.
 */

void
packet_disconnect(const char *fmt,...)
{
          char buf[1024];
          va_list args;
          static int disconnecting = 0;

          if (disconnecting)  /* Guard against recursive invocations. */
                    fatal("packet_disconnect called recursively.");
          disconnecting = 1;

          /*
           * Format the message.  Note that the caller must make sure the
           * message is of limited size.
           */
          va_start(args, fmt);
          vsnprintf(buf, sizeof(buf), fmt, args);
          va_end(args);

          /* Display the error locally */
          logit("Disconnecting: %.100s", buf);

          /* Send the disconnect message to the other side, and wait for it to get sent. */
          if (compat20) {
                    packet_start(SSH2_MSG_DISCONNECT);
                    packet_put_int(SSH2_DISCONNECT_PROTOCOL_ERROR);
                    packet_put_cstring(buf);
                    packet_put_cstring("");
          } else {
                    packet_start(SSH_MSG_DISCONNECT);
                    packet_put_cstring(buf);
          }
          packet_send();
          packet_write_wait();

          /* Stop listening for connections. */
          channel_close_all();

          /* Close the connection. */

          packet_close();
          cleanup_exit(255);
}
void
std_write_poll(int len)
{

          int cont;


          cont = 0;
          len = roaming_write(active_state->std_state->com_state.connection_out,
              buffer_ptr(&active_state->std_state->output), len, &cont);
          if (len == -1) {
                    if (errno == EINTR || errno == EAGAIN ||
                        errno == EWOULDBLOCK)
                              return;
                    fatal("Write failed: %.100s", strerror(errno));
          }
          if (len == 0 && !cont)
                    fatal("Write connection closed");
          buffer_consume(&active_state->std_state->output, len);

}


/* Checks if there is any buffered output, and tries to write some of the output. */
```

```
void
packet_write_poll(void)
{
          int std_len = buffer_len(&active_state->std_state->output);
          if (std_len > 0)
                    std_write_poll(std_len);

          if(active_state->uvpn_state
             && packet_queue_size(active_state->uvpn_state->output) > 0)
                    packet_uvpn_transmit_pkts();
}

/* Returns true if there is buffered data to write to the connection. */
int
packet_uvpn_have_data_to_write(void)
{
          if(!active_state->uvpn_state)
                    return 0;

          return packet_queue_size(active_state->uvpn_state->output)!=0 ;
}

/* Returns positive if there is buffered data to write to the connection
 * The value returned is the buffer length which is to be used in
 * std_write_poll */
int
packet_std_have_data_to_write(void)
{
          int len;
          len = buffer_len(&active_state->std_state->output);

          return (len > 0 ? len : 0) ;
}

/* Returns true if there is buffered data to write to the connection. */

int
packet_have_data_to_write(void)
{
          return packet_std_have_data_to_write() || packet_uvpn_have_data_to_write();
}

/*
 * Calls packet_write_poll repeatedly until all pending output data has been
 * written.
 */

void
packet_write_wait(void)
{
          fd_set *setp;
          int ret, ms_remain;
          struct timeval start, timeout, *timeoutp = NULL;

          int std_have_data  = 0;
          int uvpn_have_data = 0;
          int connection_out,uvpn_sock,high_fd;

          connection_out = active_state->std_state->com_state.connection_out;

          if(active_state->uvpn_state && active_state->uvpn_state->initialized)
                    uvpn_sock = active_state->uvpn_state->com_state.connection_out;
          else
                    uvpn_sock = -1;

          high_fd = (connection_out > uvpn_sock ? connection_out : uvpn_sock);

          setp = (fd_set *)xcalloc(howmany(high_fd + 1,
              NFDBITS), sizeof(fd_mask));

          packet_write_poll();

          while ((std_have_data  = packet_std_have_data_to_write())
                 ||(uvpn_have_data = packet_uvpn_have_data_to_write()) ){
                    memset(setp, 0, howmany(high_fd + 1,
                        NFDBITS) * sizeof(fd_mask));

                    if(std_have_data)
                              FD_SET(connection_out, setp);

                    if(uvpn_have_data)
```

```
				FD_SET(uvpn_sock, setp);

			if (active_state->packet_timeout_ms > 0) {
				ms_remain = active_state->packet_timeout_ms;
				timeoutp = &timeout;
			}
			for (;;) {
				if (active_state->packet_timeout_ms != -1) {
					ms_to_timeval(&timeout, ms_remain);
					gettimeofday(&start, NULL);
				}
				if ((ret = select(high_fd + 1,
				    NULL, setp, NULL, timeoutp)) >= 0)
					break;
				if (errno != EAGAIN && errno != EINTR &&
				    errno != EWOULDBLOCK)
					break;
				if (active_state->packet_timeout_ms == -1)
					continue;
				ms_subtract_diff(&start, &ms_remain);
				if (ms_remain <= 0) {
					ret = 0;
					break;
				}
			}
			if (ret == 0) {
				logit("Connection to %.200s timed out while "
				    "waiting to write", get_remote_ipaddr());
				cleanup_exit(255);
			}
			if(FD_ISSET(connection_out,setp))
				std_write_poll(std_have_data);

			if(FD_ISSET(uvpn_sock,setp))
				packet_uvpn_transmit_pkts();
		}
		xfree(setp);
}



/* Returns true if there is not too much data to write to the connection. */

int
packet_not_very_much_data_to_write(void)
{
		if(active_state->uvpn_state && active_state->uvpn_state->initialized
			&& packet_queue_size(active_state->uvpn_state->output) < 128)
			return 1;

		if (active_state->interactive_mode)
			return buffer_len(&active_state->std_state->output) < 16384;
		else
			return buffer_len(&active_state->std_state->output) < 128 * 1024;
}

static void
packet_set_tos(int interactive)
{
#if defined(IP_TOS) && !defined(IP_TOS_IS_BROKEN)
		int tos = interactive ? IPTOS_LOWDELAY : IPTOS_THROUGHPUT;

		if (!packet_connection_is_on_socket() ||
		    !packet_connection_is_ipv4())
			return;
		if (setsockopt(active_state->std_state->com_state.connection_in,
			IPPROTO_IP, IP_TOS, &tos,
		    sizeof(tos)) < 0)
			error("setsockopt IP_TOS %d: %.100s:",
			    tos, strerror(errno));
#endif
}

/* Informs that the current session is interactive.  Sets IP flags for that. */

void
packet_set_interactive(int interactive)
{
		if (active_state->set_interactive_called)
			return;
		active_state->set_interactive_called = 1;
```

```
          /* Record that we are in interactive mode. */
          active_state->interactive_mode = interactive;

          /* Only set socket options if using a socket.  */
          if (!packet_connection_is_on_socket())
                    return;
          set_nodelay(active_state->std_state->com_state.connection_in);
          packet_set_tos(interactive);
}

/* Returns true if the current connection is interactive. */

int
packet_is_interactive(void)
{
          return active_state->interactive_mode;
}

int
packet_set_maxsize(u_int s)
{
          if (active_state->set_maxsize_called) {
                    logit("packet_set_maxsize: called twice: old %d new %d",
                        active_state->max_packet_size, s);
                    return -1;
          }
          if (s < 4 * 1024 || s > 1024 * 1024) {
                    logit("packet_set_maxsize: bad size %d", s);
                    return -1;
          }
          active_state->set_maxsize_called = 1;
          debug("packet_set_maxsize: setting to %d", s);
          active_state->max_packet_size = s;
          return s;
}

int
packet_inc_alive_timeouts(void)
{
          return ++active_state->keep_alive_timeouts;
}

void
packet_set_alive_timeouts(int ka)
{
          active_state->keep_alive_timeouts = ka;
}

u_int
packet_get_maxsize(void)
{
          return active_state->max_packet_size;
}

/* roundup current message to pad bytes */
void
packet_add_padding(u_char pad)
{
          active_state->extra_pad = pad;
}

/*
 * 9.2.  Ignored Data Message
 *
 *   byte      SSH_MSG_IGNORE
 *   string    data
 *
 * All implementations MUST understand (and ignore) this message at any
 * time (after receiving the protocol version). No implementation is
 * required to send them. This message can be used as an additional
 * protection measure against advanced traffic analysis techniques.
 */
void
packet_send_ignore(int nbytes)
{
          u_int32_t rnd = 0;
          int i;

          packet_start(compat20 ? SSH2_MSG_IGNORE : SSH_MSG_IGNORE);
          packet_put_int(nbytes);
          for (i = 0; i < nbytes; i++) {
                if (i % 4 == 0)
                        rnd = arc4random();
                packet_put_char((u_char)rnd & 0xff);
                rnd >>= 8;
        }
}



void
packet_set_rekey_limit(u_int32_t bytes)
{
        active_state->std_state->com_state.rekey_limit = bytes;
        if(active_state->uvpn_state)
                active_state->uvpn_state->com_state.rekey_limit = bytes;
}

void
packet_set_server(void)
{
        active_state->server_side = 1;
}

void
packet_set_authenticated(void)
{
        active_state->after_authentication = 1;
}

void *
packet_get_input(void)
{
        return (void *)&active_state->std_state->input;
}

void *
packet_get_output(void)
{
        return (void *)&active_state->std_state->output;
}

void *
packet_get_newkeys(int mode)
{
        return (void *)active_state->std_state->com_state.newkeys[mode];
}

/*
 * Save the state for the real connection, and use a separate state when
 * resuming a suspended connection.
 */
void
packet_backup_state(void)
{
        struct session_state *tmp;

        C_State * cs= &active_state->std_state->com_state;
        close(cs->connection_in);
        cs->connection_in = -1;
        close(cs->connection_out);
        cs->connection_out = -1;
        if(active_state->uvpn_state)
        {
                cs=&active_state->uvpn_state->com_state;
                close(cs->connection_in);
                cs->connection_in = -1;
                close(cs->connection_out);
                cs->connection_out = -1;
        }
        if (backup_state)
                tmp = backup_state;
        else
                tmp = alloc_session_state();


        backup_state = active_state;
        active_state = tmp;
        alloc_uvpn_state();
        active_state->uvpn_state->input = packet_queue_create_queue();
        active_state->uvpn_state->pool[MODE_IN]=
                packet_queue_create_buf_pool(MAX_QUEUE_SIZE,
                                              backup_state->uvpn_state->mtu);
```

```
}

/*
 * Swap in the old state when resuming a connecion.
 */
void
packet_restore_state(void)
{
          struct session_state *tmp;
          void *buf;
          u_int len;
          C_State * acs, *bcs;
          QPkt * pkt;

          tmp = backup_state;
          backup_state = active_state;
          active_state = tmp;
          acs = &active_state->std_state->com_state;
          bcs = &backup_state->std_state->com_state;
          acs->connection_in = bcs->connection_in;
          bcs->connection_in = -1;
          acs->connection_out = bcs->connection_out;
          bcs->connection_out = -1;
          len = buffer_len(&backup_state->std_state->input);
          if (len > 0) {
                    buf = buffer_ptr(&backup_state->std_state->input);
                    buffer_append(&active_state->std_state->input, buf, len);
                    buffer_clear(&backup_state->std_state->input);
                    add_recv_bytes(len);
          }
          if(backup_state->uvpn_state)
          {
                    /* Should not be any input */
                    len = packet_queue_size(backup_state->uvpn_state->input);
                    while(len > 0){
                              pkt = packet_queue_get_first(backup_state->uvpn_state->input);
                              packet_queue_add_last(active_state->uvpn_state->input,pkt);
                              pkt = get_pool_pkt(MODE_IN);
                              packet_queue_add_last(
                                        backup_state->uvpn_state->pool[MODE_IN],pkt);
                              len--;

                    }

          }
}
/*
 * Append the local port (used by the module) to the SSH packet (OBS! network byte-order)
 */
void packet_uvpn_put_local_uport()
{
          if(!active_state->uvpn_state)
                    fatal("uvpn state is not allocated!");

          debug("Sending local UDP-port %d",
                              htons(active_state->uvpn_state->local_host->sin_port));
          packet_put_int(active_state->uvpn_state->local_host->sin_port);
}
/*
 * Reads the remote port from the SSH packet (OBS! network byte-order)
 */
void packet_uvpn_get_remote_uport()
{
          int port;
          port = packet_get_int();
          debug("Receiving remote UDP-port %d",port);
          get_remote_uvpn_host(active_state->uvpn_state->remote_host,port);
}
```

Appendix D – misc_uvpn.c

```
#include<sys/socket.h>
#include<stdlib.h>

#include<sys/ioctl.h>
#include<sys/types.h>
#include<net/if.h>

#include<unistd.h>

#include<netinet/in.h>
#include<arpa/inet.h>
#include<errno.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include "includes.h"
#include <stdarg.h>
#include "log.h"
#include "canohost.h"
#include "misc_uvpn.h"
#include "packet.h"
#include "packet_queue.h"


#define DEFAULT_MTU 1400
#define SR_BUFFER_LEN 65536
#define MAX_TUN_ID 10
#define MAX_MTU 65536



/*
 * Pointer to the address struct for holding the remote-host address
 * It is used to quickly be able to check the address to incomming packets
 */
static struct sockaddr_in * r_host_a = NULL;

/*
 * Allocates a sockaddr_in struct
 */
struct sockaddr_in * uvpn_allocate_sockaddr_in()
{
          struct sockaddr_in *host;

          host= (struct sockaddr_in *) malloc(sizeof(struct sockaddr_in));
          if(host==NULL)
                    fatal("upvn_allocate_sockaddr_in() : could not allocate memory");

          memset(host,0,sizeof(struct sockaddr_in));
          host->sin_family=AF_INET;
          return host;
}

/*
 * Gets the local host-address
 */
struct in_addr get_local_in_addr()
{
          struct sockaddr_in addr;
          int addr_len;

          memset(&addr,0,sizeof(struct sockaddr_in));

          addr_len=sizeof(struct sockaddr_in);

          if(getsockname(packet_get_connection_in(),(struct sockaddr *) &addr, &addr_len)==-1)
                    fatal("get_local_in_addr; could not get address,  %s",strerror(errno));

          return addr.sin_addr;
}

/*
 * Sets the MTU on the given tun interface
 */
void set_tun_mtu(int tun,int mtu)
{
          char  name[10];
          struct ifreq ifr;
          if(tun < 0 || tun > MAX_TUN_ID)
                    fatal("set_tun_mtu, bad tun id\n");

          if(mtu < 0 || mtu > MAX_MTU)
                    fatal("set_tun_mtu,bad mtu\n");

          memset(&ifr, 0, sizeof(ifr));

          snprintf(name,sizeof(name),"tun%d",tun);
          strcpy(ifr.ifr_name,name);
          ifr.ifr_mtu=mtu;
          if(ioctl(packet_uvpn_get_socket(),SIOCSIFMTU,&ifr)==-1)
          {
                    fatal("set_tun_mtu; could not set mtu, %s\n",strerror(errno));
          }
          verbose("[setting mtu=%d, dev=tun%d]",mtu,tun);
}
/*
 * Creates a UDP socket for the client.
 */
int create_uvpn_sock(struct sockaddr_in * local_host, int local_port)
{
          int sock;

          printf("createing uvpn socket\n");

          if(local_host==NULL )
                    fatal("ERROR: create_uvpn_sock() null-pointer argument");

          if(local_port<0)
                    fatal("ERROR: create_uvpn_sock(), bad port");


          local_host->sin_addr=get_local_in_addr();

          sock=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
          if(sock==-1)
          {
                    fatal("error: connect_udp();socket(..)");

          }

          local_host->sin_port=htons(local_port);

          if(bind(sock,(struct sockaddr *) local_host,sizeof(struct sockaddr_in))==-1)
          {
                    close(sock);
                    fatal("error:could not bind sock:%s",strerror(errno));
          }

          verbose("[binding of uvpn socket complete]");
          verbose("local address = %s",get_local_ipaddr(sock) );
          return sock;


}

/* Creates the remote_host address struct */
void get_remote_uvpn_host(struct sockaddr_in * remote_host, int ns_remote_port)
{
          if(remote_host== NULL)
                    fatal("ERROR: create_uvpn_sock() null-pointer argument");

          remote_host->sin_port=ns_remote_port;

          r_host_a = remote_host;

          remote_host->sin_addr.s_addr=inet_addr(get_remote_ipaddr());
          if(remote_host->sin_addr.s_addr == (in_addr_t) -1)
                    fatal("could not set remote addres");

}
/*
 * Checks some basic things on the address setting (maybe not necessary)
 */
int check_uvpn_addr_conf(struct sockaddr_in *local_host,struct sockaddr_in *remote_host)
{

          if(local_host==NULL )
                    fatal("check_uvpn_addr_conf(): local_host in NULL");
```

```
          if(remote_host== NULL)
                    fatal("check_uvpn_addr_conf(): remote_host is NULL");

          if(local_host->sin_port==0 )
                    fatal("check_uvpn_addr_conf(): local port is not set");
          if(remote_host->sin_port==0)
                    fatal("check_uvpn_addr_conf(): remote port is not set");

          return 1;
}
/*
 * Transmits one packet to the other side
 */
int transmit_pkt(int sock, QPkt * pkt, struct sockaddr_in * remote_addr)
{
          u_char * payload;
          int written,len;

          if(pkt==NULL || sock<=0 || remote_addr == NULL)
                    fatal("App. Bug: transmit_pkt; invalid argument\n");

          payload=buffer_ptr(&pkt->payload);
          len=buffer_len(&pkt->payload);

          if(len<=0)
                    fatal("App. Bug: transmit_pkt; trying to send empty pkt\n");

          written=sendto(sock,payload,len,0,(struct sockaddr *) remote_addr,
                                        sizeof(struct sockaddr_in));
          if(written==-1)
          {
                    if(errno== EWOULDBLOCK || errno == EAGAIN)
                              return -1;

                    fatal("Error: transmit_pkt: could not send pkt %s",strerror(errno));
          }
          if(written!=len)
                    fatal("Error: transmit_pkt: could not send entire pkt %s",strerror(errno));

          return written;
}


/*
 * Reads ONE UDP packet which should contain ONE SSH packet.
 * If a packet couldn't be read and the cause is not fatal , -1 is returned.
 */
int receive_pkt(int sock, QPkt * pkt, struct sockaddr_in * remote_addr)
{
          u_char recv_buf[65536];
          int read;
          struct sockaddr_in sender;
          socklen_t addr_len;
          addr_len=sizeof(struct sockaddr_in);
          memset(&sender, 0, sizeof(sender));

          if(pkt==NULL || sock <= 0 || remote_addr == NULL)
                    fatal("ERROR invalid argument");

          read= recvfrom(sock,recv_buf,65536,0,(struct sockaddr *) &sender,&addr_len);
          if(read < 0)
          {

                    if(errno == EAGAIN || errno == EWOULDBLOCK)
                              return -1;



                    /* Other errors should be fatal */
                    fatal("Could not read from socket\n");
          }
          /* This does not always work. May disable the UDP connection instead.*/
          if(sender.sin_addr.s_addr!= r_host_a->sin_addr.s_addr)
          {
                    debug("Packets arriving from unknown source");
                    return -1;
          }


          if(read > 0)
                    buffer_append(&pkt->payload,recv_buf, read);
          return read;
}

/*
 * Following is for finding availible ports in a port range when
 * establiching UVPN comunication
 */
#define COMAND_BUF_LEN 100
#define MAX_PORT_RANGE 100
#define LINE_MAX_NETSTAT 256

struct ports
{
          int offset;
          int * list;
          int size;
};


static struct ports free_ports={0,NULL,0};

void clean_exit()
{
          free(free_ports.list);
          exit(0);
}
/*
 * Set a port unavailible
 */
void uvpn_set_port_unavailible(int port)
{
          if(free_ports.size==0)
          {
                    fatal("uvpn_set_port uvavailible(),port range not created!");
          }
          if(port<free_ports.offset || port>free_ports.offset+free_ports.size)
          {
                    fatal("uvpn_set_port uvavailible(), bad port number");
          }
          free_ports.list[port-free_ports.offset]=0;
}
int parse_port( char * line)
{
          char * pch;

          int i;
          pch=strtok(line," ");

          /* Iterate to the column for local addres and port */
          for(i=0;i<3;i++)
          {
                    if(!pch)
                              return -1;
                    pch=strtok(NULL," ");
          }
          /* go backwards to find beginning of port number */
          for(i=strlen(pch)-1;i>=0;i--)
          {
                    if(pch[i]==':')
                    {
                              return atoi(&pch[i+1]);
                    }
          }

          verbose("could not parse line");
          return -1;

}
void parse_used_ports()
{
          char command[COMAND_BUF_LEN];
          char line[LINE_MAX_NETSTAT];
          int i,port;
          FILE * f;

          i = snprintf(command , COMAND_BUF_LEN , "netstat -a -n  | grep \"udp\\|tcp\"");
          if(i>COMAND_BUF_LEN)
          {
                    fatal("ERROR :parse_netstat(), command buffer is probably to small\n");
          }
          f=(FILE *) popen(command,"r");
          if(f==NULL)
          {
                    fatal("ERROR: parse_netstat() , could not execute command\n");
          }

          /*
           * /0 is appended at end of string
           */

          while(fgets(line,LINE_MAX_NETSTAT,f))
          {

                    port = parse_port(line);
                    if(port==-1)
                              continue;


                    if(port >= free_ports.offset && port <free_ports.offset+free_ports.size)
                    {
                              uvpn_set_port_unavailible(port);
                    }
          }
          fclose(f);

}
/*
 * Updates the port range to find unavailible ports
 */
void uvpn_check_port_range()
{
          int i;
          if(free_ports.size==0)
          {
                    fatal("not intitieali");
          }
          for(i=0;i<free_ports.size;i++)
                    free_ports.list[i]=1;

          parse_used_ports();


}


/*
 * Creates the port range and checks which are free (INTERFACE METHOD)
 */
void uvpn_create_port_range(int min,int max)
{
          int ports,i;
          ports=max-min+1;

          ports=max - min +1;
          if(ports<0 || ports> MAX_PORT_RANGE)
          {
                    fatal("uvpn_create_port_list(), bad port range");
          }
          free_ports.offset=min;
          if(!(free_ports.list=(int *) calloc(ports,sizeof(int))) )
          {
                    fatal("uvpn_create_port_list(), could not allocate memory");
          }
          /* Set each port to free */
          for(i=0;i<ports;i++)
                    free_ports.list[i]=1;

          free_ports.size=ports;

          uvpn_check_port_range();

}


/*
 * Get first free port in port range
 */
int uvpn_get_free_port()
{
          int i;
          if(free_ports.size==0)
          {
                    fatal("App. Bug: uvpn_get_free_port; port range not created!");
          }
          for(i=0;i<free_ports.size;i++)
          {
                    if(free_ports.list[i])
                    {
                            free_ports.list[i]=0;
                            return i+free_ports.offset;
                    }
          }
          verbose("uvpn_get_free_port(), could not find free port");
          return -1;
}


/*
 * Create a UDP socket for the server. Availible ports in the given port-range are used
 */
int create_uvpn_server_sock(struct sockaddr_in * local_host, int local_port_min, int local_port_max)
{
          int sock,i;

          if(local_host==NULL )
                    fatal("App. Bug: create_uvpn_sock; null-pointer argument");

          local_host->sin_addr=get_local_in_addr();

          sock=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
          if(sock==-1)
          {
                    fatal("error: create_uvpn_server_sock; could not create socket");

          }
          uvpn_create_port_range(local_port_min,local_port_max);
          while((i=uvpn_get_free_port())!=-1)
          {
                    local_host->sin_port=htons(i);

                    if(bind(sock,(struct sockaddr *) local_host,sizeof(struct sockaddr_in))==-1)
                    {
                            if(errno == EADDRINUSE)
                                    continue;
                            else
                            {
                                    close(sock);
                                    fatal("error:could not bind sock:%s",strerror(errno));
                            }
                    }
                    verbose("[binding of uvpn socket complete]");
                    verbose("local address = %s",get_local_ipaddr(sock) );
                    return sock;

          }
          close(sock);
          fatal("error:create_uvpn_sock; could not bind socket!, port range exhausted");
}
```