Universiteit van Amsterdam
System & Network Engineering

Comparing TCP performance of tunneled and non-tunneled traffic using OpenVPN

Authors:
Berry Hoekstra
bhoekstra@os3.nl

Damir Musulin
dmusulin@os3.nl

Supervisor:
Jan Just Keijser
Nikhef

August 24, 2011
Version 1.1 — Revision 191

This page is left blank intentionally.

Abstract

When operating OpenVPN at Gigabit speed a bottleneck is seen in the throughput of data. This research is concentrated on finding the cause(s) of the bottlenecks that is shown when OpenVPN is operated at Gigabit speed. To pinpoint possible bottlenecks, a lab setup was created to perform test procedures. Different cryptographic algorithms were used in combination with different parameters. Test results show that a combination of factors determine the network throughput of data between nodes through OpenVPN. It seems that OpenVPN suffers from a fundamental problem. OpenVPN delivers data in small data packets to be encrypted using OpenSSL EVP function calls. The overhead of encrypting small data packets instead of one big data packet seems to matter in the performance of encrypting data. This study shows the impact in network performance of using different parameters of OpenVPN, OpenSSL EVP function calls, the TUN/TAP driver, operating system and network infrastructure.

Contents

1 Introduction

The emergence of high speed Internet connectivity enables one to connect multiple endpoints over the Internet without limitations such as slow network connectivity or high latency. This opens up the possibility for corporations to introduce secure high speed tunnels between office branches. Proprietary VPN solutions based on the IPsec protocol exist that are able to do this [1] [2] , but often introduce high costs, a required level of knowledge and administrative overhead. However, a widely used open source solution is available. OpenVPN [3] provides an open source community edition that enables secure connectivity between two or more hosts. While OpenVPN is capable of saturating a 100 Mbps connection, performance problems still exist on faster links. This might become a problem as Internet connections of 1 Gigabit per second (Gbps) become generally available [4]. Available bandwidth capabilities can not be fully utilized when using OpenVPN as a VPN solution. The lack of documentation on this specific topic encourages this study.

1.1 Research

In this work we study the causes of the decrease in network performance when using OpenVPN on a 1 Gigabit connection. This approach may provide a solution to achieve higher network throughput and point out possible bottlenecks. The results may provide detailed insights in the impact in network performance when using OpenVPN with different parameters.

In this research, we will answer the following research question:

> *What are the causes of the network performance loss when using OpenVPN at Gigabit speed?*

The following sub-questions will help to answer the main research question.

- What is the effect of using different encryption and authentication methods or parameters in OpenVPN?
- Is the same performance hit found on other OpenSSL-based tunnel solutions?
- Is the same performance hit found on other operating systems (e.g. FreeBSD)?
- What are the possibilities to mitigate slow OpenVPN network performance?

2 Virtual Private Networks (VPN)

2.1 VPN characteristics

A Virtual Private Network (VPN) enables a secure point-to-point connection between two or more points over an otherwise insecure network. The VPN connection is established by creating a tunnel between the connected nodes. The tunneling technique enables one to securely transfer data between the endpoints over the network. The payload data is encrypted and encapsulated into the layer below it. This is done by adding additional header information to the packets that are destined for the remote endpoint [5].

To secure a VPN connection, the following security measures are taken:

Encryption is the process of using an encryption algorithm to transform plaintext data into information that is unreadable for anyone but the one with the decryption key.

Authentication is when both sides provide information that can prove their identity to both ends. This is mostly done by using a Message Authentication Code (MAC) [6].

Integrity of data is done by digitally signing the data. A checksum of the data is generated that is a unique value of the contents of a packet. Integrity checking can be done by using an encryption algorithm such as MD5 or SHA-1.

Non-repudiation is a method in cryptography, which provides the sender of the data with a proof that the data is delivered to the other endpoint, while ensuring the senders identity to the receiver. Such a method can avoid the denial of a transaction at a later time.

In general, VPNs are used to securely access a private network from a remote location. When creating a VPN, the two networks at both ends are connected using a communication channel, which is secured using encryption. Integrity and authentication can be enforced by using hashing algorithms such as MD5 or SHA-1 and key exchange mechanisms such as a certificate or a pre-shared key. The connection is characterized by the fact that the connected points in the VPN are virtually connected in a private network. Hence, the technique is called Virtual Private Network.

2.1.1 VPN technologies

Generally, the four types of VPN technologies that are most popular are listed below.

Secure Socket Layer/Transport Layer Security (SSL/TLS) is a standard to encrypt communication channels between a client and a server at the Transport Layer (Layer 4, TCP/UDP). When implemented in a VPN tunnel, all communication on the tunnel is encrypted end-to-end, while authentication and integrity is supplied by SSL/TLS.

Internet Protocol Security (IPsec) is a security extension of the Internet Protocol developed by the Internet Engineering Task Force (IETF). IPsec enables encrypted end-to-end tunneling (tunnel mode) or packet encryption (transport mode). It authenticates and encrypts each IP packet depending on the implementation. In contrast to SSL-based VPN, encryption is done at the Network Layer (Layer 3, IP), instead of the Application Layer (Layer 7, e.g. SSL/TLS).

Point-to-Point Tunneling Protocol (PPTP) is a VPN technique running at the Data Link Layer (Layer 2) and is mostly used in Microsoft's Windows operating systems. It was introduced by the PPTP Forum, which consists of Microsoft, 3Com, US Robotics and other companies. Point-to-Point Protocol (PPP) packets are used to encapsulate data into IP datagrams for transmission [7]. Although there are multiple implementations of PPTP, the security model is based on tunneling the PPP packets. The most widely used implementation is Microsoft's version, which implements a security model based on PPP-based protocols for authentication and encryption. Packet filtering is used to enhance network security.

Layer 2 Tunneling Protocol (L2TP) is a combination of the best features in PPTP and the Layer 2 Forwarding Protocol (L2F) developed by Cisco [8]. Encryption and authentication is often implemented in combination with IPsec (L2TP/IPsec) [9], while sessions are maintained by the PPP protocol [10].

2.1.2 VPN packet transport

In the previous section, we covered some of the most used VPN implementations. Although different methods are used to secure the data, the way the encrypted data is transferred is similar. The basic principle for securely transferring data across is by using a tunneled connection between endpoints. In Figure 1 a visualization of the packet flow inside a VPN tunnel is shown.

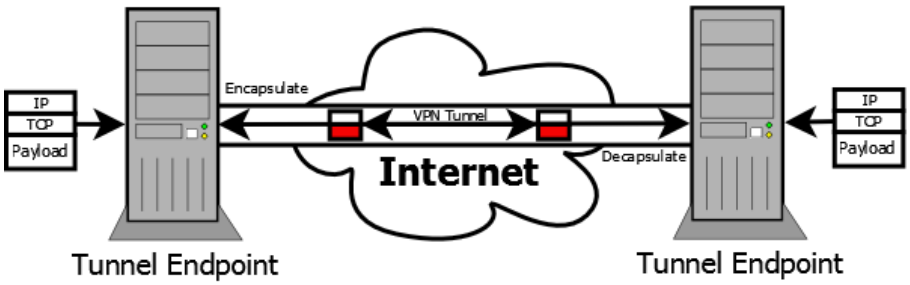

Figure 1: Packet flow inside a VPN tunnel.

The general idea is that the data is encrypted and encapsulated within another packet on a lower layer. The encapsulated data travels over the tunnel and is decapsulated at the other end, where the decapsulated packet is given to the kernel for further processing. In the Figure above, the encrypted packets are marked red, the encapsulation headers of the encapsulated packets in the tunnel are marked white.

2.2 OpenVPN

In this research we focus on OpenVPN [3], which is an open source software solution that implements VPN (Virtual Private Network) methods based on SSLv3/TLSv1 [11]. Unlike the SSL/TLS VPN techniques discussed above in Section 2.1.1 , OpenVPN applies encryption, authentication and integrity using the OpenSSL library [12] in conjunction with a virtual network interface, the TUN/TAP device [13] which is used for injecting network packets into the operating system.

OpenVPN is developed by James Yonan and is able to create encrypted tunnels over the Internet or other network. Unlike many other VPN implementations, OpenVPN does not use the IPsec protocol to secure the data transfers. Moreover, the application distinguishes itself with other VPN solutions with the fact it is a user space implementation of a VPN.

2.2.1 Application structure

OpenVPN consists of a single binary that can be used for both client and server modes of the application. It is currently available for the following platforms:

- Linux
- Windows 2000/XP/Vista/7
- {Open,Free,Net}BSD

- Solaris
- QNX
- Mac OS X
- Mobile OSs

User space The fact that OpenVPN is a user space VPN provides the advantage that the application is more easily developed and ported to other operating systems, as the kernel code does not need to be altered with every release. Running in user space also provides more security, as user space applications are more restricted in system calls and provides memory protection, as the application is isolated in memory [14].

Packet transport Figure 2 shows the flow of a packet destined for the other side of the tunnel.

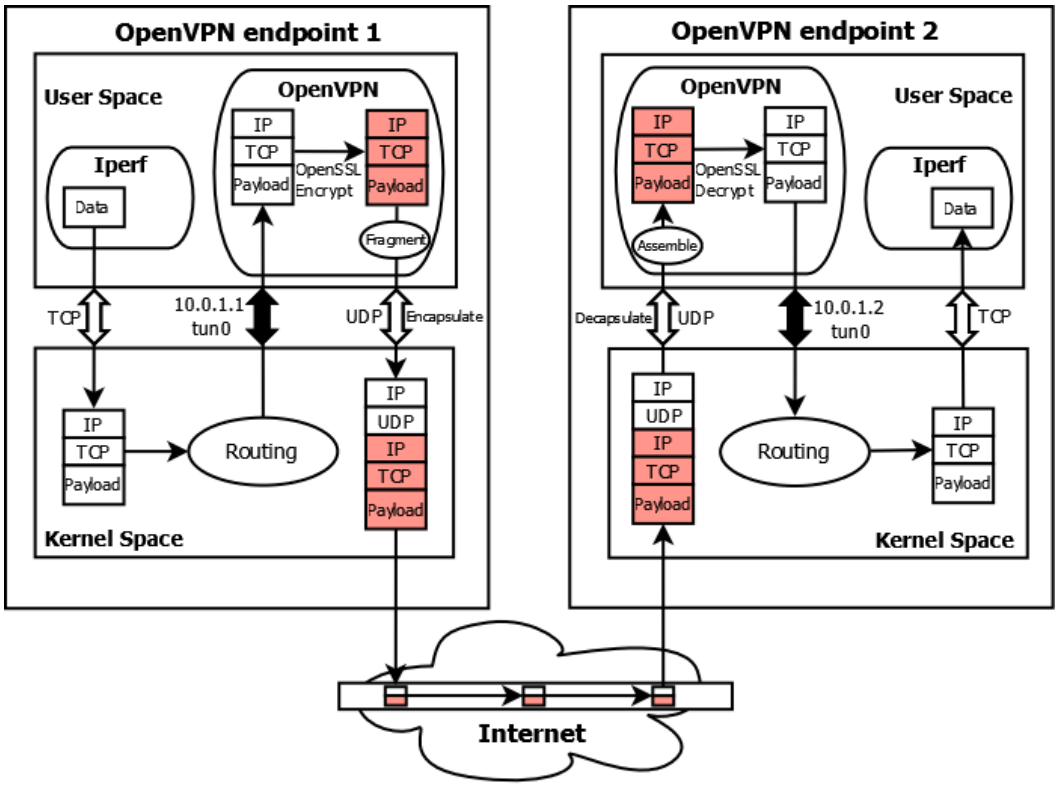

Figure 2: OpenVPN packet flow.

In this case, Iperf pushes a packet to the kernel via a TCP socket. Here, the kernel picks up the supplied destination address and routes it to the

over the virtual TUN interface to user space, according to the routing rules, and adds a source IP address. Here, OpenVPN reads the data that is sent by the kernel, encrypts and signs it by making use of the OpenSSL library. Then, the encrypted data is encapsulated inside a UDP packet and put on the outgoing buffer of the tunnel driver. Back in kernel space it travels over the physical network interface onto the network. The other side reverses the same process to be able to receive the data.

TUN/TAP driver To be able to create a tunnel, OpenVPN makes use of the tunnel driver that is provided as a module for the Linux kernel since 1999 [13]. The tunnel driver enables the creation of a virtual network interface, the so-called TUN/TAP device. The virtual network interface can be configured as a Layer 2 device (TAP) or as a Layer 3 device (TUN), which enables user space applications to create network bridges or perform routing, without the administrative burden of adjusting kernel configurations and parameters. In practice the difference means that with a TAP device a program is able to send out Ethernet frames, these Ethernet frames are capable to be send to computers in the same network segment, but can not be send to computers in other network segments, because segment then need to be routed, routing is done at layer 3 of the OSI model, thus a TUN device is needed to create packets that can be routed to other network segments. In the research we focus on TUN devices because VPN applications are mostly used to create a point-to-point connection between different networks segments.

When creating a point-to-point VPN with OpenVPN, a virtual TUN interface is created when initializing the VPN connection. The TUN interface is configured on both sides using the given IP addresses to create the point-to-point connection. Then, the required route to the other point of the tunnel is added to the routing table. On Linux, this is all achieved by calling the `ifconfig` and `ip route` tools from within OpenVPN.

Once the tunnel is set up, all packets destined for the other point of the tunnel will be forwarded by OpenVPN to the TUN device that will inject the packets into kernel space, which in turn sends the packets over the tunnel across to the other side. By making use of the tunnel driver to send packets to a other point of the tunnel, OpenVPN does not need to take care of the transmission of the data to the other point of the tunnel.

Switching between the kernel space and user space causes context switching. This is the process of "parking" the state of a process in the CPUs register for the CPU scheduler to be able to execute another process [15] . After this process is completed, an interrupt is generated to inform the action is completed. These cause for overhead, as the CPU is occupied with storing the states during context switching, and has to process the interrupts every time.

OpenSSL OpenVPN can be viewed as a layer on top of the Linux TUN/TAP driver and the OpenSSL library. All encryption and hashing algorithms

are performed by OpenSSL. Data that is not encrypted and put on a tunnel is traveling the tunnel in clear text. To solve this, OpenVPN payloads and control channels are encrypted and authenticated using a HMAC digest by using the OpenSSL EVP function calls.

Security modes As discussed in Section 2.2, the OpenSSL library can be used to apply encryption, authentication and integrity. A VPN tunnel can be instantiated in two different modes. Both modes have different (dis)advantages. The following table gives an overview of both security modes.

OpenVPN mode	Pre-shared keys	SSL
Cryptography mode	Symmetric	Asymmetric/Symmetric
Implementation	Easier	Harder
Speed	Faster	Slower
CPU usage	Lower	Higher
Key exchange	Yes	No
Encryption keys renewal	No	Yes
Peers authentication	No	Yes
Perfect Forward Secrecy	No	Yes

Table 1: OpenVPN security mode comparison [16]

Hardware acceleration The cryptographic functionality provided to OpenVPN by the OpenSSL library are CPU intensive operations. By offloading these CPU intensive operations to a piece of hardware that is designed to perform cryptographic functions, the CPU is relieved of these tasks. Because the hardware is specialized in performing cryptographic algorithms, it performs them much faster than a software based solution. Hardware acceleration is done using special engines that can be plugged into OpenSSL. Available engines can be viewed using the `openssl engine` command. In most cases, acceleration is done using special cryptographic cards, which can be addressed by OpenSSL from version 0.9.6 [17]. However, a recent interesting development in this field has been done by Intel. The AES-NI [18] instruction set is used to improve encryption and decryption performance using the AES encryption standard. This is a special instruction set proposed by Intel and AMD in 2008, which enables hardware acceleration of the AES standard. It plugs into OpenSSL as a separate engine module. A patch [19] has to be applied in order to use the new instruction set. In turn, VIA developed the PadLock engine [20] for improving AES performance. In this research, hardware acceleration will not be used, as there is no hardware available for this at the time of writing.

3 Related research

In this section, existing studies are reviewed and existing tools are discussed.

3.1 Literature review

There are some studies done that are related to this research. Some studies provided information and insights that were useful in this study. They are discussed in this section.

A study by the SANS Institute [21] discusses the concepts of SSL VPNs in general, but mainly focuses on OpenVPN [22]. The paper gives a short introduction to cryptography in general and explains the features, workings and internals of OpenVPN. A more practical and detailed guide to OpenVPN is the OpenVPN Cookbook written by Jan Just Keijser [23]. The Cookbook provides practical information on the usage of OpenVPN and also briefly describes performance on 1 Gbps connections. On page 238, the author suggests there is a performance bottleneck in the TUN/TAP driver in Linux.

The United States based National Institute of Standards and Technology (NIST) provides a "Guide to SSL VPNs" [5], which is a paper about the fundamentals of SSL VPNs and also describes the general workings of SSL VPNs. It provides detailed insight in all aspects of implementing a SSL VPN.

When comparing network performance while applying encryption and decryption operations to the traffic, the performance of the encryption algorithms must be taken into account. A performance evaluation of symmetric encryption algorithms is shown in a journal called Communications of the IBIMA [24]. The research includes the performance of the Blowfish and AES encryption algorithms and looks into the performance for different packet and key sizes. In another research, authors Nadeem, A. and Javed, M.Y. conclude that the Blowfish algorithm is the best performing algorithm based on their test setup [25]

Research by Kshirasagar Naik and David S. L. Wei states that there are more resources used for the transmission of the data than for the computation [26]. This will also be taken into account in this research.

Performance measurements of OpenVPN itself was also performed in a research that looked into OpenVPN performance on a consumer router running custom firmware [27]. The paper provided insight into possible research methodologies used in this research.

3.2 Existing tools

The following tools have been used in the research.

3.2.1 Tunneling

Simpletun is a program that allows one to create a tunnel between two points over a given network. The tunnel is created with the help of the Linux tunnel driver. The created tunnel is a TCP based plaintext tunnel [28].

Vtun is a program that allows one to create a tunnel between two point over a given network. Vtun differs from Simpletun in the amount of options that Vtun provides. For example, it is possible to create a UDP or TCP based tunnel and apply compression of the data that travels over the tunnel [29].

OpenVPN is used as the main VPN research tool in this project. Network performance issues occur on 1 Gbps connections as described in Section 1.

tinc is another VPN daemon that also uses the OpenSSL library to provide encryption and authentication [30].

3.2.2 Network measurement

Iperf can be used to measure TCP and UDP bandwidth performance [31]. It was developed by NLAND/DAST. A client-server setup is used to generate a traffic stream between two nodes. After a test is completed, the server sends back a report to the client. An interesting feature is the ability to save results to CSV format, which enables easy plot creation of the network throughput. An example research of Iperf [32] shows how useful Iperf can be in the testing of network throughput.

nuttcp is a benchmark tool to network performance. nuttcp is based on nttcp, which is an enhancement of ttcp [33]. Compared to Iperf, the reporting capabilities are not as advanced, as Iperf uses more CPU cycles to be able to present the right throughput. This is also mentioned in research done by Wu et al. [34].

netperf is able to measure network performance on many different types of networking [35]. It uses a similar client-server setup as Iperf. After some initial testing, we found netperf to be not as reliable compared to Iperf, as the measured throughput did not compare to manual throughput tests.

From the tools described above, Iperf was used as a traffic generator. The reason for this is that Iperf provides all the necessary functions needed for the measurements, while also providing reliable reporting capabilities.

4 Methodology

In this section the methodology that is used for the research is discussed. As stated in Section 1 , OpenVPN, or one of its components, introduces a possible bottleneck that causes the inability to saturate a 1 Gbps connection. It is not clear where the actual bottleneck occurs.

4.1 Lab setup

Two lab setup were created to be able to perform measurements using OpenVPN.

4.1.1 Endpoint to endpoint

The initial setup consists of two machines acting as the VPN endpoints. As shown in Figure 3 , the machines are connected back-to-back using 1 Gigabit Ethernet to eliminate possible overhead that might be caused when using a switch to connect both machines. This lab setup consists of two machines that are identical, apart from the hard drives. This difference does not influence test results, as the hard drive performance is similar and all operations that are related with the measurements are done in memory.

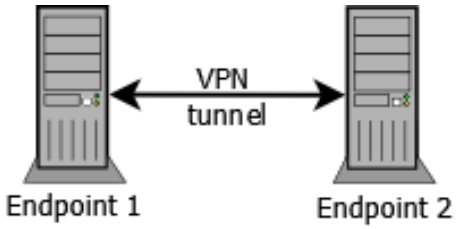

Figure 3: Lab setup 1: VPN with two endpoints.

4.1.2 Client to client

The second setup, as shown in Figure 4 , extends the initial one discussed above. Two additional machines with identical hardware specifications are added to the setup. These machine will act as regular client machines on a LAN, which use the VPN endpoints as a gateway to be able to reach each other. Routing is applied on all machines using default Linux kernel routing. The clients that are connected to the machines acting as VPN endpoints are used to simulate an actual real-live scenario where two office branches are interconnected using a VPN tunnel, delivered by OpenVPN. Although Figure 4 only shows two clients, multiple clients can be attached. The lab setup only has one client attached to each VPN endpoint for simulation purposes.

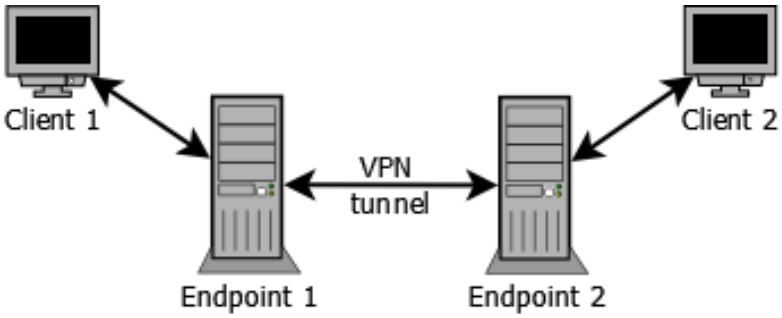

Figure 4: Lab setup 2: VPN with two endpoints and two clients.

The complete hardware specifications of both lab setups can be viewed in the Appendix A.

4.2 Test procedures

To get consistent test results, test procedures were created. The defined test procedures were applied on the different lab setups using different configuration parameters. This method of consistent testing results in comparable measurements.

4.2.1 Test plan

By performing measurements and compare them accordingly, we are able to observe possible bottlenecks in the functioning of OpenVPN and its components.

Raw speed of the network can be determined by performing an Iperf throughput test over the link without establishing a VPN connection.

Parameters A set of parameters were defined that can have an influence on the network performance of OpenVPN. The measurements differ from each other by the different values that are filled in for the test parameters.

Once the results of the different test procedures are known, it is possible to determine what options have a effect on the packet flow. Besides knowing what options have a effect on the packet flow of OpenVPN, it will enable us to analyze the effects of using certain options. This will specify a possible cause of the cause of in the decrease of network performance.

OpenSSL OpenVPN uses OpenSSL to encrypt and sign the packets that travel through the VPN tunnel. For the research it is interesting to study the influence of different versions of OpenSSL. Furthermore it would be interesting to see what the maximum performance is of OpenSSL encryption on a given system. The maximum encryption rate of OpenSSL can be compared with the maximum throughput of data that travels through a tunnel that is encrypted with OpenVPN.

Infrastructure Measurements will be performed on a second infrastructure as shown in Section 4.1 to eliminate possible routing issues with the Linux kernel.

Operating system As OpenVPN runs in user space, it can be easily ported to other operating systems. It is not known what the performance influence of a operation system is. To see the impact of using a different operating system, measurements will also be performed on FreeBSD.

4.2.2 Measuring network performance

As stated in Section 1.1 the research is focused on finding the cause of the inability of OpenVPN to saturate a 1 Gbps network link. The delay on the network is measured and different parameters are applied on both lab setups to determine the influence in network performance. The network throughput will indicate what the influence of a given parameter is.

Delay on the network causes a decrease in traffic throughput. The Round Trip Time (RTT) of the network links shows the latency on the link. The RTT is measured on both lab setups for a duration of one hour. This should be sufficient to be able to calculate the average RTT of the links. The measurements are done on both lab setups, over both non-VPN and VPN connections. The RTT values are gathered by using the standard `ping` command. Using the results of the ping command, the RTT values are extracted by using the following command:

Extracting the RTT data from the ping data.

```
$ cat pingdata.txt | grep icmp_seq | \
cut -d'=' -f4 | cut -d' ' -f1 > rttdata.txt
```

These values will be used to calculate the average RTT is using the "`stats`" utility, which is part of the `iproute2` package. The tool can be used to view information on the average RTT, the variation of the RTT, and the distribution of the RTT overtime. This information can be viewed by doing the following:

Extracting characteristics using stats.

```
$ ./stats rttdata.txt
mu =        0.249299
sigma =     0.025419
rho =       0.440400
```

This shows that the average delay on the link is 0.249 milliseconds. When this measurement is done on a VPN link, the following results apply:

Extracting characteristics using stats.

```
$ ./stats rttdata-vpn.txt
mu =        0.464440
sigma =     0.051739
rho =       0.293843
```

We immediately observe an increase in latency of 86,30%, from 0.249 to 0.464 milliseconds. This means that TCP traffic over the UDP-based VPN tunnel will decrease as a higher round trip time results in lower throughput of TCP traffic. However, this is mitigated by the dynamic TCP window scaling of the Linux kernel. UDP is not affected, as no packet loss is measured.

Traffic generation In Section 3.2.2 , we stated that we chose Iperf to generate the network traffic. Iperf is able to generate network traffic between two nodes. One node will act as the server, while a second node generates traffic to the server. Iperf can operate in UDP and TCP mode. OpenVPN operates in UDP mode by default to prevent TCP stacking problems [36] , as the TCP protocol is a more widely used protocol than UDP. As our lab setup uses OpenVPN in UDP mode, we chose to run Iperf in TCP mode. Each measurement will be done a total of three times for a duration of five minutes. This provides us with more reliable test results as the average throughput of each test can be calculated. Iperf v2.0.5 was built from the source tarball using default configuration options. On the server side, Iperf is started in server mode:

```
$ iperf -s
```

On the client side, Iperf has to connect to the server side, over the VPN tunnel at IP address 10.0.1.1 for a duration of five minutes (300 seconds). This is done using the following command:

```
$ iperf -c 10.0.1.1 -t 300
```

Using the above commands a traffic generation test is started. Iperf will generate data that will be sent to the other side of the VPN tunnel. After five minutes, the measured throughput will be reported on both client and server side. The report of the client side was used as the measured throughput, as it seemed to most accurate. The average of the measured results of all three tests were used as the measured throughput.

4.2.3 Source code analysis

An analysis of the source code may provide detailed insights into the workings of OpenVPN and the used OpenSSL function calls. We choose to do this analysis, as it may enable us to pinpoint the exact causes of the performance issues. The source code analysis is done in Section 6.

4.3 Control script

All the parameters were supplied by a control script, which is located in the Appendix B. The script is executed from a separate node located in the same network. Different commands are executed on the machines in the lab setup to be able to automate the tests. Using public key authentication, together with SSH, commands are executed on all machines to setup VPN connections and Iperf throughput tests. After a timeout, the script will perform the same tests again, but with different parameters. The following sections describe the details of these parameters.

4.4 OpenVPN parameters

The different parameters that are available in conjunction with OpenVPN are discussed in this section. The measurements are done using these parameters to observe the influences in network performance. Configuration files are used with OpenVPN to create consistent tests using the same settings. A distinction is made between server (VPN1) and client (VPN2).

The configuration file of the server is shown below:

VPN1 OpenVPN configuration file.

```
$ cat /etc/openvpn/config/default-static-server.cfg
dev tun
proto udp
local 10.0.0.1
remote 10.0.0.2
port 11000

secret /etc/openvpn/secret.key
ifconfig 10.0.1.1 10.0.1.2
```

The configuration file of the client that will connect to the server is shown below:

VPN2 OpenVPN configuration file.

```
$ cat /etc/openvpn/config/default-static-client.cfg
dev tun
proto udp
local 10.0.0.2
remote 10.0.0.1
port 11000

secret /etc/openvpn/secret.key
ifconfig 10.0.1.2 10.0.1.1
```

This method enables us to create a basic OpenVPN tunnel. Additional parameters can be passed to the application using the command line. An example of this is shown below:

```
$ openvpn --daemon --config /etc/openvpn/config/default-static-server.cfg \
--cipher aes-256-cbc --auth sha1 --fragment 0 --mssfix 0 --tun-mtu 60000
```

This command will start OpenVPN using the parameters in the configuration file, and uses the AES-256-CBC cipher for encryption, the SHA-1 algorithm for HMAC signing (default), disables all OpenVPN packet fragmentation options and configures the MTU of the TUN device to be 60000 bytes. In return, the client issues the following command to connect to the server:

```
$ openvpn --daemon --config /etc/openvpn/config/default-static-client.cfg \
--cipher aes-256-cbc --auth sha1 --fragment 0 --mssfix 0 --tun-mtu 60000
```

4.4.1 Encryption algorithms

To secure the data that travels over the VPN connection, OpenVPN applies encryption before encapsulating the data inside packets on a lower layer. The encryption is applied by using the open source OpenSSL library. Although OpenSSL supports a lot of different encryption and signing algorithms, OpenVPN currently only supports the following encryption modes [37]:

- Cipher Block Chaining (CBC)
- Cipher FeedBack (CFB)
- Output FeedBack (OFB)

Out of the encryption algorithms and security modes supported by the OpenSSL library, the ones that are most used with OpenVPN are the Blowfish and AES ciphers in CBC mode, which is an encryption method for encrypting blocks of plaintext. If needed, padding is applied to form a complete block of data.

Blowfish is a symmetric block cipher consisting of 64 byte blocks. It uses keys lengths varying from 32 bits to 448 bits used in 16 rounds for encryption. This cipher is used in CBC mode by default by OpenVPN.

Advanced Encryption Standard (AES) is a standard for symmetric-key encryption, which provides three different block ciphers. Each block cipher is 128 bits in size, and has different key sizes of 128, 192 and 256 bits that are used in 10, 12 or 14 rounds respectively. AES has been through extensive research by security experts, and is considered secure enough by the National Security Agency (NSA) to protect classified information. When the AES-256 cipher is used, the `--cipher aes-256-cbc` parameter has to be passed to OpenVPN.

The reason only these algorithms are used most is mainly because the performance is excellent compared to the alternative standards. Another reason is that OpenVPN uses the Blowfish cipher by default, while AES is the encryption standard approved by NIST.

Static key To simplify the measurements, we chose the static key method together with the Blowfish, AES-128 and AES-256 encryption algorithms in CBC mode. Static key encryption is used for the ease of setting up an encrypted tunnel. The alternative method would be using private keys together with certificates. There is no difference in performance between the two methods, only in the ease of configuration as there is no key distribution needed when using a static key (public key encryption/asymmetric encryption).

The OpenVPN static key is generated by using the following `openvpn` command:

```
$ openvpn -genkey -secret secret.key
```

From OpenVPN version 1.5-beta13, this command will generate a static key that consists of 512 hexadecimal characters, which results in a 2048-bit static key. Parts of the key are used in OpenVPN for encryption and authentication. As there is only one static key to maintain between two connected points, administration is relatively easy. However, the key must be present on both sides of connected tunnel. To prevent the key from being sniffed/leaked during the transfer to the other endpoint, a secure transfer method such as the `scp` secure copy command should be used. In Figure 5 , a visual view of the static key is presented, together with the parts that are used.

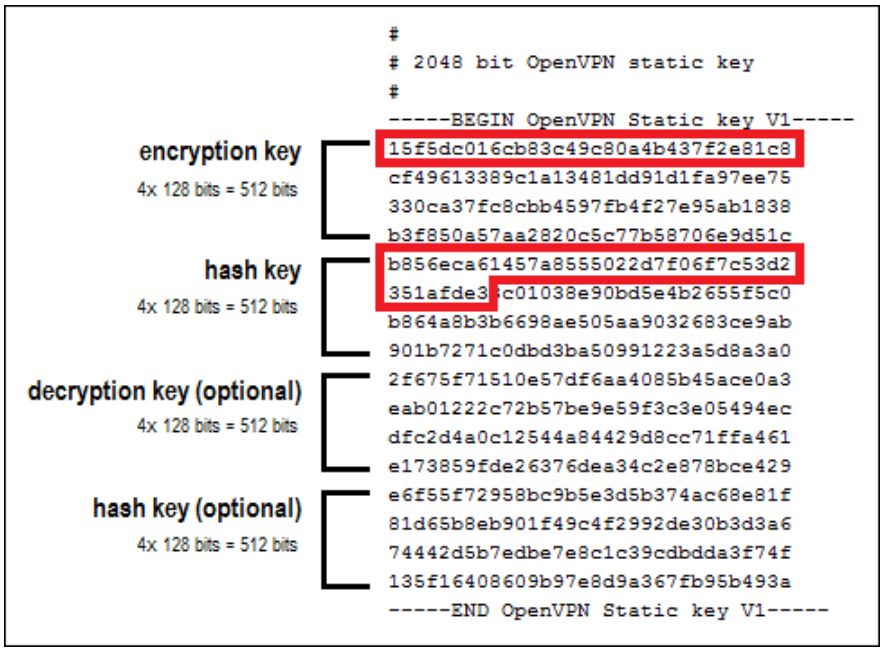

Figure 5: OpenVPN static key layout.

By default, OpenVPN uses the Blowfish algorithm. As shown in Figure 5 , the first part of the key that is marked red is used for encrypting the data (128 bits). The second part, also marked red, is the part used for the hash algorithm (160 bits). By default, both connected endpoints use the same parts of the static key. However, it is possible to use different keys for encryption and decryption on both sides, so more bits of the static key can be used.

The key file that was generated using the `openvpn` command above is used by using the `--secret` command with OpenVPN on both endpoints. The key can be tested by using the following command:

```
$ openvpn --test-crypto --secret secret.key
```

If the test is successful, OpenVPN is able to execute the OpenSSL function calls.

4.4.2 Authentication and signing algorithms

A Hash-based Message Authentication Code (HMAC) is used to provide packet "authentication". In essence, an HMAC is a specific cryptographic hashing algorithm that is able to calculate a Message Authentication Code (MAC) using a secret key. In this case the secret key is the OpenVPN static key described in Section 4.4.1. OpenVPN uses the OpenSSL function calls to sign all packets using the hashing algorithm to enable packet integrity. The use of a HMAC provides additional security feature for the VPN traffic. In the case where tampering of the VPN packets occurs, for example by a Man in the Middle attack [38], the HMAC signature will not be as expected, and OpenVPN will not trust the packet in question and a retransmission of the mismatched packets is needed.

SHA-1 By default OpenVPN uses SHA-1 algorithm (Secure Hash Algorithm) to sign the packets travelling across the VPN tunnel. The SHA-1 algorithm is a cryptographic hash function developed by the NSA. Although there are weaknesses in the algorithm [39] , it is considered secure enough for data transmission. The algorithm can be used on 512 bit data blocks to calculate a HMAC-MD5 HMAC of 160 bits.

MD5 Besides SHA-1, another algorithm that available is MD5 (Message Digest 5), which is a 128-bit cryptographic hash function that is less CPU intensive than SHA-1, but also vulnerable for collision attacks [40]. MD5 is commonly used to check the integrity of files, but can also be used on 512 bit data blocks to calculate a HMAC-MD5 HMAC of 128 bits. As discussed in subsection 2.1 , data signing is used to provide both integrity and authentication. To measure the difference in network performance, the MD5 algorithm is also used to perform HMAC authentication between the VPN endpoints.

4.4.3 Fragmentation options

The OpenVPN fragmentation options enable the encapsulation of TCP packets within a UDP datagram without the need for fragmentation.

Maximum Transmission Unit The MTU is the amount of bytes a protocol can send in one protocol data unit (PDU) [41]. In the case of Ethernet, increasing the size of the MTU enables more bytes to be sent to the target destination in every Ethernet frame. This will reduce the amount of frames that have to be sent. By increasing the Ethernet MTU from the default 1500 to 9000 bytes throughout an entire network path, fragmentation of the frames will not occur. A MTU of 9000 bytes is commonly known as Jumbo Frames [42]. In our measurements, we assume that most network interfaces don't use a MTU of 9000, which means that there is no available path MTU of 9000. The MTU value of the physical network interface will not be altered when doing throughput measurements.

As OpenVPN uses a virtual network interface, we can configure a different MTU value. This enables us to increase the amount of bytes sent through the virtual interface at once. As OpenVPN uses OpenSSL calls to encrypt the data coming through the virtual interface, we are able to increase the amount of bytes that is encrypted at once, which will theoretically increase performance. Measurements will be done using increasing MTU values of 1500, 6000, 9000, 12000, 24000, 36000, 48000 and 60000 bytes. This will show us the best value to use with certain cryptographic algorithms. This approach might also point out a possible bottleneck in the TUN/TAP driver or a limitation in the OS context switching between kernel space and user space.

Packet fragmentation The fragmentation option in OpenVPN (`--fragment`) configures OpenVPN in such a way that it never tries to send a UDP datagram that is larger than the given amount of bytes. The fragment option would be useful in cases with packets that require fragmentation, such as non-TCP protocol, or when path MTU discovery is broken. With a broken MTU it is not possible to set the correct MTU for a path, which can result in packet errors due to the fact that packets that are sent are to big for the other side. If packets larger than the specified amount come through, internal fragmentation is applied, which fragments the packets for the network interface before sending the data over the network. As a result, 4 bytes are added to the UDP datagram to be able to reassemble the datagram on the receiving side.

Maximum Segment Size (MSS) is the amount of bytes that can be sent or received unfragmented [43]. An option is provided to be able to configure OpenVPN to use a certain MSS value for the payload data, which is useful to solve MTU fragmentation problems. Using the `--mssfix` option, OpenVPN will calculate the overhead that is generated during the encapsulation process, and configures the TCP protocol to not use a MSS larger than the given value in bytes. This value is used during the TCP negotiations to prevent the fragmentation. This option is mostly used in conjunction with the `--fragment` option, as the `--mssfix` option prevents TCP from needing fragmentation and let `--fragment` fragments non-TCP packets internally if needed.

Even when there is no encryption applied to the tunnel, overhead is still present as the packets are encapsulated. Context switches between the user space and kernel space cause a slight hit in performance, which in turn will decrease network performance as less CPU cycles are available.

4.5 OpenSSL performance measurements

To test the hardware capabilities of our test setup, we measure the encryption speed of OpenSSL. All three encryption algorithms will be tested on a large file. The encryption test will be performed a total of three times to be able to calculate the average time of the algorithm. Different buffer sizes of 1024, 8192 and 60000 bytes respectively will be used to be able to measure differences in encryption speed when supplying OpenSSL with different sizes of plaintext blocks to encrypt. This can be done by using the `-bufsize` switch. The `time` command shows the amount of time a test takes. A file of 4.689.108.992 bytes is supplied to OpenSSL using the `-in` switch, together with a password used for encryption. To start the test, the following OpenSSL command is executed:

```
$ time openssl bf-cbc -in ~/large.iso -bufsize 60000 \
-pass pass:testing123 > /dev/null
```

The following formula is used together with the average encryption time to measure the theoretical throughput:

$$Throughput(Mbps) = \frac{Filesize(Bytes)/Encryptiontime(Seconds)}{(1024 \times 1024)}$$

4.6 Improving network throughput

Packet fragmentation options Using the `--fragment` and `--mssfix` options discussed in Section 4.4.3, network throughput might be optimized. For example, on high speeds, fragmentation of packets will cause overhead, as more 4 bytes are added to each fragment. Another factor is that more CPU cycles are needed for fragmentation and interrupt processing.

MTU options Increasing the MTU of the TUN/TAP device causes that more bytes can be sent to OpenVPN at once. When more bytes are delivered for encryption and signing, OpenVPN can optimally process the packets and send them over the line. No time is spent waiting for buffers to fill up.

Cryptographic options Another factor that might improve network throughput is by using different cryptographic algorithms used for the encryption and authentication of the packets. In contrast to the Blowfish algorithm, the AES algorithms are much more CPU intensive, and causes the CPU to be more occupied with cryptographic tasks than with the processing of the network packets.

Processor affinity OpenVPN is a CPU intensive program. Research by Intel [44] shows that the performance of CPU intensive program can be increased when assigning the process to a core that is close to memory or the network bus. As we are interested in gaining the maximum network performance using OpenVPN, processor affinity is researched in conjunction with OpenVPN. To test the processor affinity, the `taskset` command is used to lock a running process to a specified core. The example command below is used to assign a process with process ID 30431 to core 0:

```
$ taskset -pc 0 30431
```

The `taskset` command is part of the Linux scheduler utilities.

5 Measurements

In this Section, we will discuss the results of the tests that are described in the test plan in Section 4.2.1.

5.1 Executing test procedures

As described in Section 4.4, measurements are done by generating traffic using Iperf. Each measurement differs with the MTU of the virtual TUN network device. The MTU parameter is used with increasing values. This is done to see what the performance impact of the MTU size is on the throughput of data for the given static parameters. The fragmentation and MSS options with each test enabled and disabled and different encryption algorithm are used. The MD5 algorithm will be tested against the default SHA-1 algorithm used for the HMAC authentication to measure the difference in network performance. The tests will be done on both lab setups and another OS in the form of FreeBSD 8.2 will be used to eliminate OS restrictions. Note that a "default" measurement is done with encryption and SHA-1 signing enabled.

5.2 OpenSSL performance measurements

OpenSSL encryption speed is measured as discussed in Section 4.5. The graph in Figure 6 shows the differences in encryption speed offering different blocks of plaintext, the OpenSSL speed test shown are executed on CentOS 5.6.

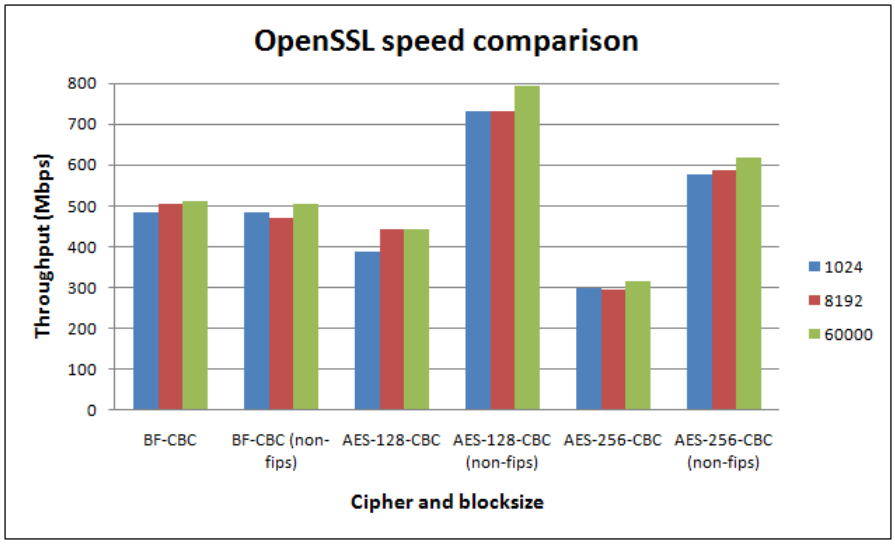

Figure 6: OpenSSL speed comparison.

5.3 Network performance measurements

In Section 4.2.2, the network latency was measured on both the normal links and VPN enabled links. Measurements over the VPN connection already shows us a slight increase in latency. An increase of 86,30%, from 0.249 to 0.464 milliseconds is measured when using the `ping` command over the VPN link.

The graphs generated out of the measurement results are separated per encryption cipher. This is done to avoid clutter when combining too many results in the graphs.

5.3.1 Generating traffic

As discussed in Section 4.2.2, traffic streams were generated between the two nodes using Iperf. This enabled us to determine network throughput of the network connection.

5.3.2 Raw throughput

To be able to rule out problems on the physical link, a raw throughput test was performed using Iperf. Measurements done on the first lab setup showed a raw network throughput of 942 Mbps with a configured MTU value of 1500 bytes. When using a MTU of 9000 bytes, the throughput increased to 990 Mbps. This shows that using a higher MTU results in a higher throughput. However, the physical network device will be configured with 1500 MTU, as this is the standard MTU value on the Internet at the moment.

5.3.3 Transparent VPN tunnel

A transparent tunnel is a VPN tunnel where no encryption and signing is applied to the traffic. When configuring a transparent VPN tunnel, OpenVPN uses a shortcut as there is no need to request an OpenSSL action. The graph in Figure 7 shows the results of the throughput measurements done on the raw link compared against the results of the transparent VPN tunnel with and without the OpenVPN fragmentation options enabled. The graphs shows that near-line-speed can be reached on the transparent tunnel when configuring the TUN device with a MTU size of 6000 bytes and higher. The fragmentation options limit the throughput to around 50% of the raw line speed.

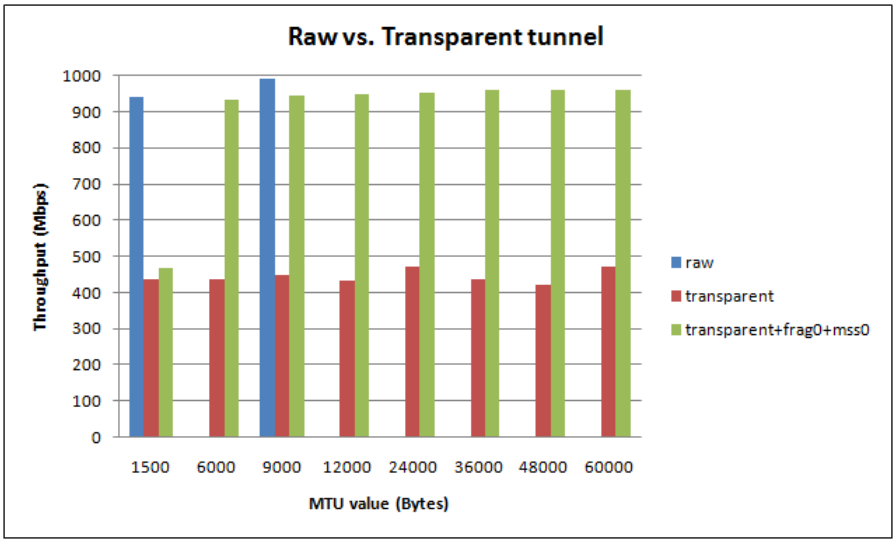

Figure 7: Network throughput measurement of raw versus transparent tunnel.

5.3.4 Blowfish-CBC

The throughput measured during the OpenSSL speed-test was around 500 Mbps. The throughput measured on the network is shown in Figure 8. The graph shows us that with the fragmentation options disabled, the network throughput does not greatly exceed 500 Mbps, with the exception of the test with a TUN MTU value of 48000 bytes. HMAC signing using the SHA-1 algorithm shows a performance hit of around 29% on the 48000 MTU test.

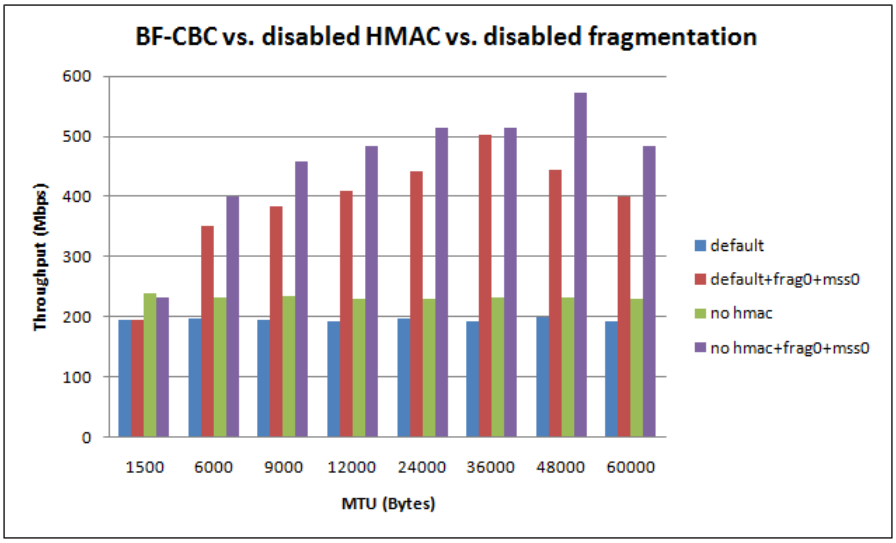

Figure 8: Network throughput measurement of BF-CBC tunnel.

The figure also shows us that using a larger MTU value of the TUN device increases throughput. The optimal throughput is reached on MTUs from 12000 bytes and larger. We clearly see that there is no difference in the OpenSSL speed and network throughput when using OpenVPN. This observation can be led back to the processor limitations present in the Blowfish CBC cipher. Multiple studies show that Blowfish outperforms the AES algorithm [24] [45]. However, when testing the encryption speed on our test setups using OpenSSL, we see that AES performs much better. The encryption of the Blowfish-CBC algorithm is limited to the clock speed of the CPU.

5.3.5 AES-128-CBC

The AES-128-CBC cipher reaches a theoretical OpenSSL throughput of 389 to 442 Mbps. Figure 9 shows that the measured throughput does not reach far beyond 300 Mbps when only encryption is used and disabled fragmentation options. This is a performance hit of around 44%.

Again, the optimal speed is reached on a MTU of 12000 bytes. When we look at the measurement that reached the highest throughput, with a TUN MTU value of 12000 bytes, the performance hit caused by the SHA-1 HMAC signing is around 14%. Again, the gain in throughput is seen when disabling the fragmentation options. This is observed on all TUN MTU sizes.

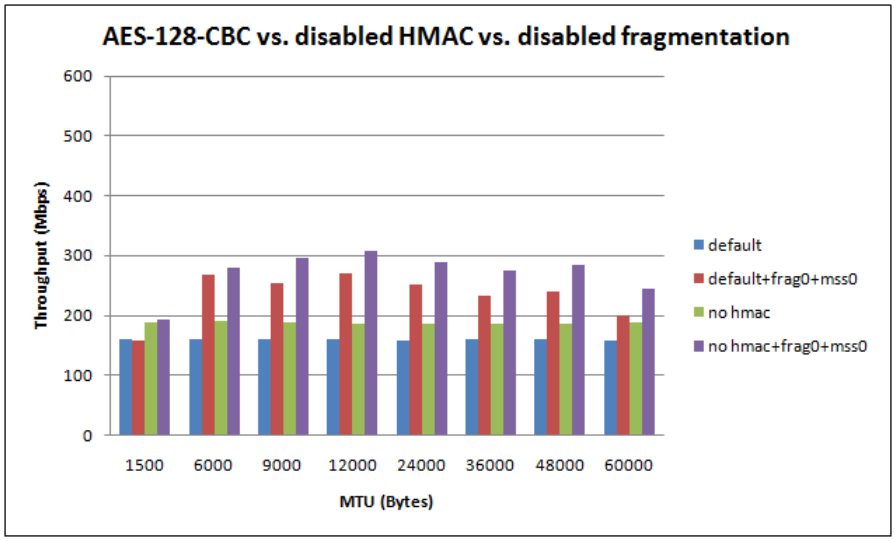

Figure 9: Network throughput measurement of AES-128-CBC tunnel.

5.3.6 AES-256-CBC

The OpenSSL test resulted in a theoretical throughput of 296 to 317 Mbps using the AES-256-CBC cipher. When we look at the network measurement results in Figure 10, we see that this throughput is not reached. The optimal throughput is measured on a TUN MTU of 12000 bytes, but this is only 230 Mbps, against the 317 Mbps of the maximum measured OpenSSL speed. This is a performance loss of around 38%.

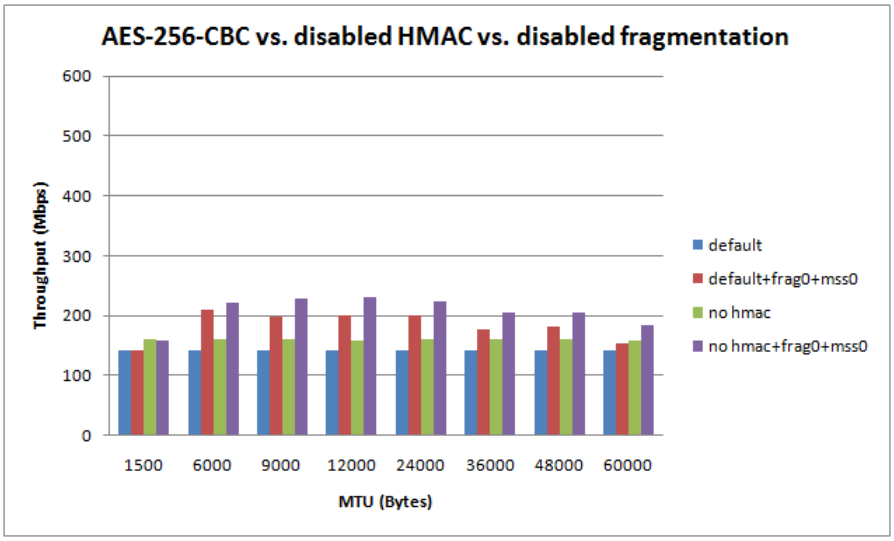

Figure 10: Network throughput measurement of AES-256-CBC tunnel.

5.4 Comparing to the HMAC-MD5 algorithm

To measure the impact of using different hashing algorithms for the HMAC signing, we have ran tests on a VPN tunnel where only HMAC signing was configured. Throughput results of a transparent tunnel versus using SHA-1 or MD5 for HMAC signing were compared. Figure 11 shows this comparison. The graph shows the performance hit of both algorithms against using a transparent tunnel.

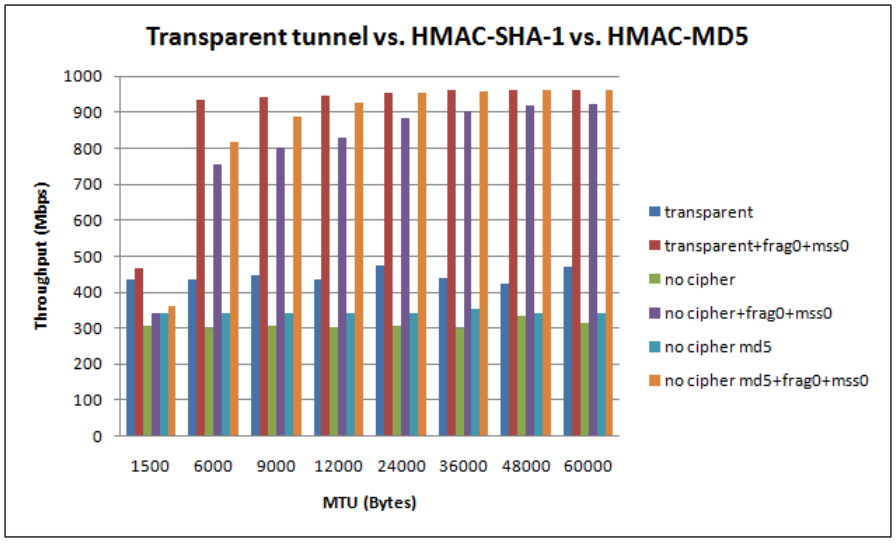

Figure 11: Network throughput measurement different HMACs.

When the OpenVPN fragmentation options are enabled, throughput does not go beyond 500 Mbps, but instead reaches an average throughput of 444 Mbps. However, we see that tests done on the transparent tunnel reach a maximum throughput of near-line-speed when disabling the fragmentation options.

When only applying the SHA-1 algorithm on a VPN tunnel, a performance hit with a maximum of 26,6% is seen on a TUN MTU of 1500 bytes. The performance hit gradually decreases when the amount of bytes the TUN MTU is configured with increases. For example, when configuring the TUN MTU with 60000 bytes the measured performance hit is only 3.95%.

Comparing this to the MD5 algorithm, which is less CPU intensive, a maximum hit of 22,97% on a TUN MTU of 1500 bytes to 0% on 60000 bytes is seen. The following table shows a comparison of the differences:

SHA-1 hit	**MD5 hit**	**Difference**
-26,6%	-22,97%	+3,63%
-19,06%	-12,28%	+6,78%
-15,12%	-5,9%	+9,22%
-12,46%	-2,22%	+10,24%
-7,37%	-0,1%	+7,27%
-5,83%	-0,1%	+5,73%
-4,48%	0%	+4,48%
-3,95%	0%	+3,95%

Table 2: HMAC algorithm performance hit comparison.

The difference of using SHA-1 versus the MD5 algorithm is negligible. From a TUN MTU with 12000 bytes configured, applying MD5 signing does not affect throughput much. Similar behaviour is seen for SHA-1 from a TUN MTU with 24000 bytes configured. MD5 even reaches same throughput levels as a transparent tunnel, which is the connection limit in this case.

5.5 Comparing to other SSL VPN solutions

Vtun and a modified version of the Simpletun code were used to be able to compare OpenVPN with other VPN solutions that use OpenSSL. Both applications are discussed in Section 3.2. The modifications to Simpletun are discussed in Section 6.

The idea behind the comparison is that a simple program like Simpletun with encryption added will show a performance overhead by encrypting the data that is going through the tunnel.

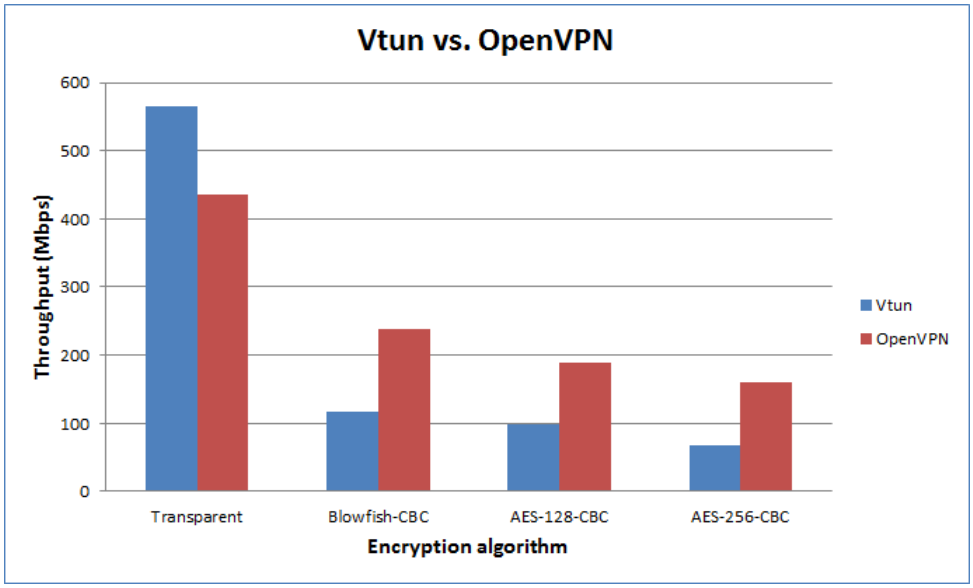

Figure 12: OpenVPN vs Vtun

The result for the modified Simpletun are omitted. This was done as it was not possible to get reliable measurements because of segmentation faults occurring at at random intervals. We expect the more error handling needs to be implemented in the Simpletun code, which was not feasible in the limited time frame for this project.

Figure 12 shows that the throughput with OpenVPN is slower on a transparent tunnel compared with Vtun. However, when encryption is applied, OpenVPN outperforms Vtun. From this result an unclear picture emerges about the relative performance of both products.

5.6 Comparing with FreeBSD

As OpenVPN is available for multiple operating systems, a comparison with FreeBSD may provide more insight. It presents an opportunity to measure if there is a performance difference between CentOS 5.6 and FreeBSD 8.2.

Figure 13 shows the results of the measurements done.

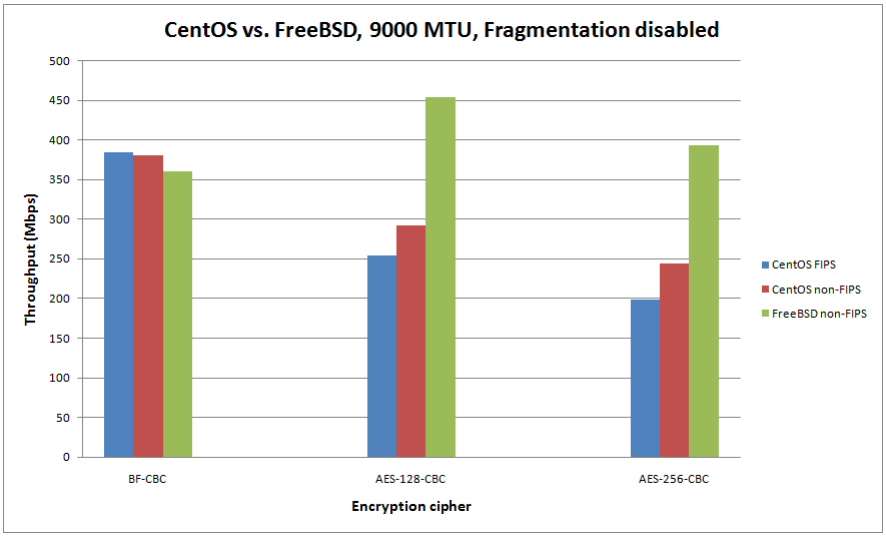

Figure 13: Network throughput measurement of CentOS vs. FreeBSD.

We notice similar results on both operating systems using the Blowfish-CBC cipher. This can again be led back to the relation with the CPU frequency. However, when using AES-CBC ciphers, we notice a 50% drop in performance on the CentOS installation. Analysis shows this behavior is caused by the default OpenSSL installation on this machine, which is FIPS compliant [46]. For example, when performing tests using the AES-256-CBC cipher, with a version of OpenSSL that is not FIPS enabled, a 60% increase in network throughput is measured when using the same source on FreeBSD 8.2.

5.6.1 FIPS vs. non-FIPS

FIPS is the Federal Information Processing Standard 140-2, which is a data processing standard by the US government. To test the impact of the FIPS version of OpenSSL against the non-FIPS version, OpenSSL had to be rebuild on the original test operating system (CentOS). OpenVPN had to be compiled again using the new OpenSSL libraries. A comparison is shown in Figure 14 below.

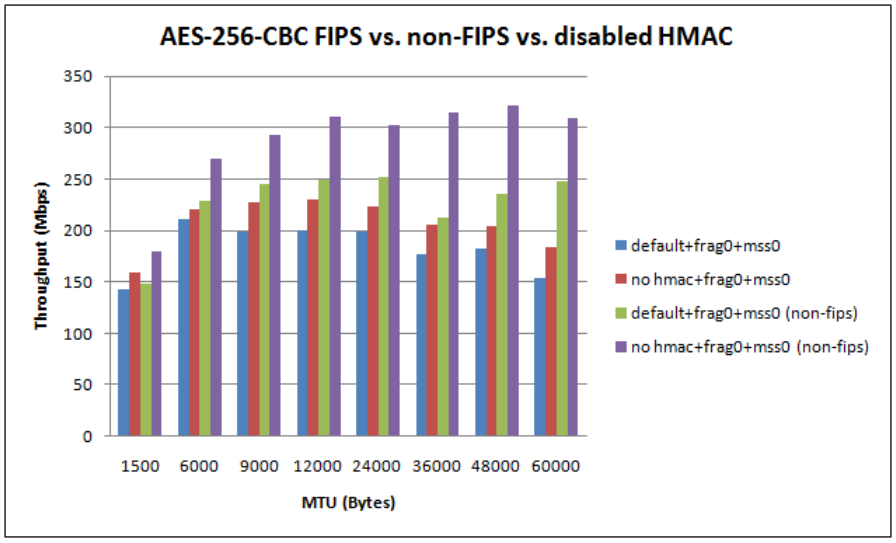

Figure 14: Network throughput measurement of AES-256-CBC tunnel (FIPS vs. non-FIPS).

We observe a large increase in throughput on a TUN MTU of 60000 bytes where only encryption is used. This test shows an increase of 126 Mbps, which is 68,48% higher than the FIPS version. This shows the difference between this FIPS version is substantial. We have learned that this loss in performance is not seen in the FIPS version 1.0.0 of OpenSSL. This points to a bug in the implementation of the FIPS module in the default version of OpenSSL shipped with CentOS, which is version `0.9.8e-fips-rhel5`

5.7 Comparing with a routed setup

The second lab setup, described in Section 4.1, is a routed setup.

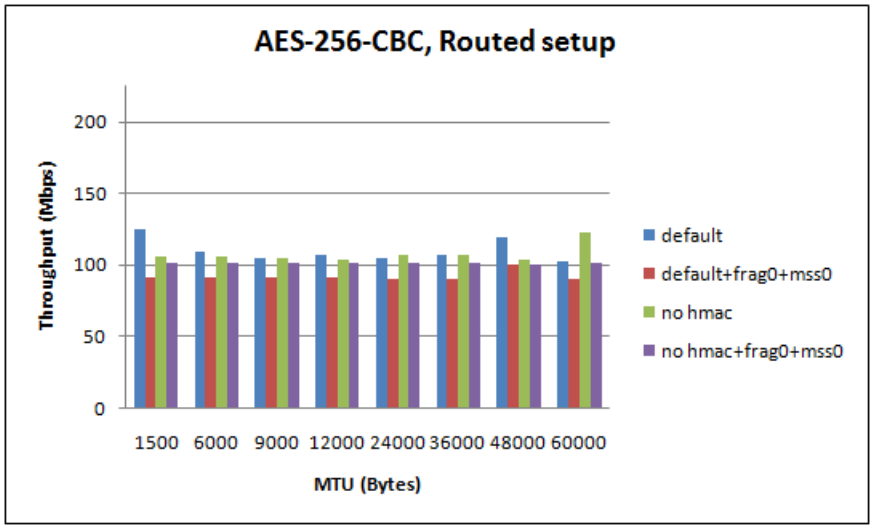

Figure 15: OpenVPN in conjunction with clients.

Figure 15 shows the results of the routed setup with different TUN MTU sizes. The graph shows that increasing the MTU of the tunnel does not lead to a better performance in throughput of data. The reason for the constant numbers is due to the fact that the the physical interface that receives the data packets from the clients has a MTU of 1500, thus increasing the MTU of the tunnel device of an endpoint will not lead to a better performance of throughput.

5.8 Processor affinity

Research by Intel [44] shows an increase in TCP network performance in Symmetric Multi Processor systems. To test the influence on OpenVPN throughput, we measured the throughput by assigning the most CPU intensive process to the core that is the closest to the system bus, so that transport to memory and network card is optimal.

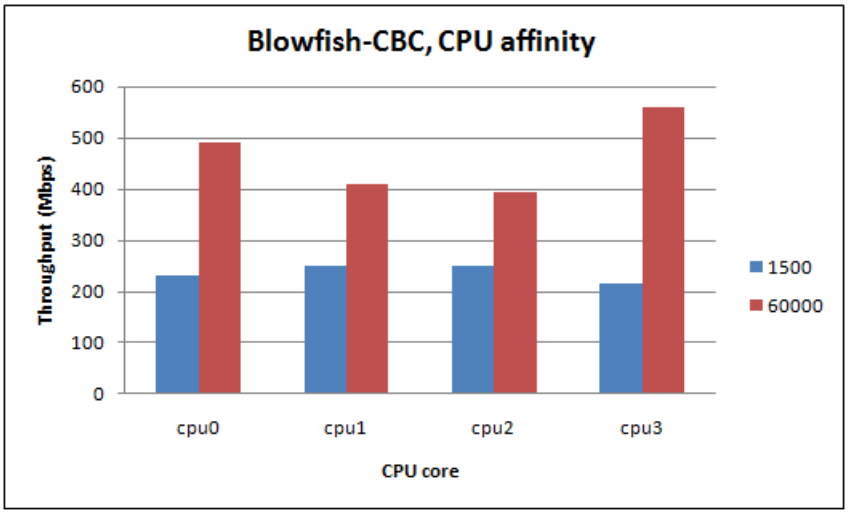

Figure 16: Throughput of OpenVPN on different cores.

Figure 16 shows the throughput of the OpenVPN process running on different cores of a processor. After the OpenVPN process is started, the process ID is allocated to a specific CPU core to measure the effectiveness in throughput performance. This is done as discussed in Section 4.6.

When configuring the TUN interface with a large MTU, a big block of data is given to OpenVPN and in turn to OpenSSL. A big performance differences is seen among the cores. Core 0 and core 3 of the used CPU show a performance that is significantly better when compared with core 1 and core 2. A maximum increase of around 40% is seen. Core 3 shows an increase of 10% when no affinity is used.

6 Source code analysis

To gain a better understanding of OpenVPN a source code analysis has been done.

6.1 OpenVPN encrypted packet flow

As described in Section 2.1.2 packets that flow through the tunnel to the other endpoint of a tunnel have a specific packet flow through an endpoint system. In this section the packet flow is of packets that are put on a VPN tunnel are shown in combination with source code from OpenVPN.

As described in Section 2.2.1, OpenVPN uses the TUN/TAP driver to setup a point-to-point connection between two endpoints. The piece of code that is shown below is the starting point of reading the data from the TUN virtual network interface. First, the function `read_incoming_tun` is called. Then, the code checks if a signal such as a SIGHUP is received. If this is not the case, the packet is processed by the `process_incoming_tun` function.

OpenVPN process io function. (forward.c)

```
1494 void
1495 process_io (struct context *c)
1496 {
1497   const unsigned int status = c->c2.event_set_status;
...
 /* Incoming data on TUN device */
1525   else if (status & TUN_READ)
1526     {
1527       read_incoming_tun (c);
1528       if (!IS_SIG (c))
1529     process_incoming_tun (c);
1530     }
...
```

The code below shows the entry point of a data packet that flow into OpenVPN. The function called `read_tun_buffered` takes three parameters:

1. The file descriptor of the TUN/TAP interface.
2. The data buffer that contains the data for the destination endpoint.
3. The frame length.

If `IS_SIG` is false the packet can be processed by the `process_incoming_tun` function.

OpenVPN read incoming tun function. (forward.c)

```
910 void
911 read_incoming_tun (struct context *c)
912 {
913   /*
914    * Setup for read() call on TUN/TAP device.
915    */
916   /*ASSERT (!c->c2.to_link.len);*/
917
918   perf_push (PERF_READ_IN_TUN);
919
920   c->c2.buf = c->c2.buffers->read_tun_buf;
921 #ifdef TUN_PASS_BUFFER
922   read_tun_buffered (c->c1.tuntap, &c->c2.buf, MAX_RW_SIZE_TUN (&c->c2.frame));
923 #else
924   ASSERT (buf_init (&c->c2.buf, FRAME_HEADROOM (&c->c2.frame)));
925   ASSERT (buf_safe (&c->c2.buf, MAX_RW_SIZE_TUN (&c->c2.frame)));
926   c->c2.buf.len = read_tun (c->c1.tuntap, BPTR (&c->c2.buf), \
MAX_RW_SIZE_TUN  (&c->c2.frame));
927 #endif
```

The `process_incoming_tun` that is shown below is shown in the bare essentials that we need to explain. The code shows that the buffer length of `c` is checked, if it is bigger than zero it is than `c` is passed along to the `encrypt_sign` function, else the buffer is reset.

OpenVPN process incoming tun function. (forward.c)

```
958 void
959 process_incoming_tun (struct context *c)
960 {
...
965   if (c->c2.buf.len > 0)
966     c->c2.tun_read_bytes += c->c2.buf.len;
...
976   if (c->c2.buf.len > 0)
977     {
...
993       encrypt_sign (c, true);
994     }
995   else
996     {
997       buf_reset (&c->c2.to_link);
998     }
```

```
...
1001 }
```

Two things that can be seen in the code below is that the function `openvpn_encrypt` is called. After the `openvpn_encrypt` call, the encrypted packet receives the address where it needs to go to. First, we will explain the `openvpn_encrypt` function, then the `link_socket_get_outgoing_addr` function.

The `openvpn_encrypt` functions are shown below:

OpenVPN encrypt signing function. (forward.c)

```
421 void
422 encrypt_sign (struct context *c, bool comp_frag)
423 {
...
449 #ifdef USE_CRYPTO
...
461   /*
462    * Encrypt the packet and write an optional
463    * HMAC signature.
464    */
465   openvpn_encrypt (&c->c2.buf, b->encrypt_buf, &c->c2.crypto_options, &c->c2.frame);
466 #endif
467   /*
468    * Get the address we will be sending the packet to.
469    */
470   link_socket_get_outgoing_addr (&c->c2.buf, get_link_socket_info (c),
471                  &c->c2.to_link_addr);
472 #ifdef USE_CRYPTO
```

OpenVPN openvpn encrypt function. (crypto.c)

```
73 void
74 openvpn_encrypt (struct buffer *buf, struct buffer work,
75          const struct crypto_options *opt,
76          const struct frame* frame)
77 {
...
81   if (buf->len > 0 && opt->key_ctx_bi)
82     {
83       struct key_ctx *ctx = &opt->key_ctx_bi->encrypt;
84
85       /* Do Encrypt from buf -> work */
86       if (ctx->cipher)
```

```
87     {
88       uint8_t iv_buf[EVP_MAX_IV_LENGTH];
89       const int iv_size = EVP_CIPHER_CTX_iv_length (ctx->cipher);
90       const unsigned int mode = EVP_CIPHER_CTX_mode (ctx->cipher);
91       int outlen;
...
127       /* initialize work buffer with FRAME_HEADROOM bytes of prepend capacity */
128       ASSERT (buf_init (&work, FRAME_HEADROOM (frame)));
...
137       /* cipher_ctx was already initialized with key & keylen */
138       ASSERT (EVP_CipherInit_ov (ctx->cipher, NULL, NULL, iv_buf, DO_ENCRYPT));
...
154       /* Encrypt packet ID, payload */
155       ASSERT (EVP_CipherUpdate_ov (ctx->cipher, BPTR (&work), &outlen, \
             BPTR (buf), BLEN (buf)));
156       work.len += outlen;
157
158       /* Flush the encryption buffer */
159       ASSERT (EVP_CipherFinal (ctx->cipher, BPTR (&work) + outlen, &outlen));
160       work.len += outlen;
161       ASSERT (outlen == iv_size);
162
163       /* prepend the IV to the ciphertext */
164       if (opt->flags & CO_USE_IV)
165         {
166           uint8_t *output = buf_prepend (&work, iv_size);
167           ASSERT (output);
168           memcpy (output, iv_buf, iv_size);
169         }
...
182       work = *buf;
```

Out of the code above, we see what happens with the interaction between OpenVPN and OpenSSL. The EVP (EnVeloPe) functions that are shown are part of the high level API interface of OpenSSL [47]. On line 138, the `ctx` (cipher context) structure is initialized. As the comment on line 137 states, the key and key length are already set for the cipher context structure. The initialization of the key and the key length is done at line line 89 and line 90. Once the `ctx` structure is initialized, the encryption can start. This can be seen on line 155.

The data that is located in the buffer called `buf` is passed to OpenSSL to encrypt. The desired encryption is set in the `ctx` structure. The encrypted data is put into the work buffer that is initialized at line 128. After the `EVP_CipherUpdate_ov` is done, it can happen that there is data left that is too small to fit in a given encryption block. This is data that is placed in a partial block. To solve the encryption with a partial block, the `EVP_CipherFinal` function is called. This

function adds additional padding to the partial block to be able to encrypt it as an entire block of plaintext data.

On line 182 the most interesting part happens. The plaintext buffer called `buf` is pointed to the memory address of the encrypted buffer called `work`. This means that functions that call the buffer pointer will get the memory address of the encrypted buffer work. The code at line 182 effectively swaps the plaintext buffer with a encrypted buffer.

After the buffer is encrypted by the `openvpn_encrypt` function, the `link_socket_get_outgoing_addr` function is called by the `encrypt_sign` function. The `link_socket_get_outgoing_addr` is used as is defined in the name, to get the outgoing address for the encrypted buffer. After this part, there is no code left to be executed. This seems to be strange as the packet is encrypted and the outgoing address is also known, so all that needs to be done is to send the packet to the other endpoint of the tunnel.

OpenVPN link socket get outgoing addr function. (socket.h)

```
642 static inline void
643 link_socket_get_outgoing_addr (struct buffer *buf,
644                   const struct link_socket_info *info,
645                   struct link_socket_actual **act)
646 {
647   if (buf->len > 0)
648     {
649       struct link_socket_addr *lsa = info->lsa;
650       if (link_socket_actual_defined (&lsa->actual))
651     *act = &lsa->actual;
652       else
653     {
654       link_socket_bad_outgoing_addr ();
655       buf->len = 0;
656       *act = NULL;
657     }
658     }
659 }
```

The code that is shown below shows a while loop that continues until interrupted by a signal, which are caught by `P2P_CHECK_SIG();`. As long as the while loop runs, the function `process_io (c);` is executed. This function processes the input and output of the TUN/TAP device and the network device. This function was already discussed in Section 6.1 for the TUN device. However, now we need to send the encrypted data with its address on a socket for the device that is connected with the network.

OpenVPN tunnel point-to-point function (openvpn.c)

```
45 static void
46 tunnel_point_to_point (struct context *c)
47 {
48   context_clear_2 (c);
49
50   /* set point-to-point mode */
51   c->mode = CM_P2P;
52
53   /* initialize tunnel instance */
54   init_instance_handle_signals (c, c->es, CC_HARD_USR1_TO_HUP);
55   if (IS_SIG (c))
56     return;
57
58   /* main event loop */
59   while (true)
60     {
...
78       /* process the I/O which triggered select */
79       process_io (c);
80       P2P_CHECK_SIG();
...
83     }
84
85   uninit_management_callback ();
86
87   /* tear down tunnel instance (unless --persist-tun) */
88   close_instance (c);
89 }
```

The next example below shows the first part of the `process_io` function where the encrypted buffer with the remote address included is sent to the other side of the endpoint by using the `process_outgoing_link` function.

OpenVPN process io function. (forward.c)

```
1494 void
1495 process_io (struct context *c)
1496 {
1497   const unsigned int status = c->c2.event_set_status;
1498
1499 #ifdef ENABLE_MANAGEMENT
1500   if (status & (MANAGEMENT_READ|MANAGEMENT_WRITE))
1501     {
1502       ASSERT (management);
1503       management_io (management);
```

```
1504     }
1505 #endif
1506
1507   /* TCP/UDP port ready to accept write */
1508   if (status & SOCKET_WRITE)
1509     {
1510       process_outgoing_link (c);
1511     }
```

Next, the first the packet is checked for total length. If the total length is larger than zero and bigger than the expanded size, the code is allowed to continue. After the packet is checked for the correct sizes, the link address is checked. If this is defined as true, then we are able to send the code to the other side of the tunnel. In the code at line 1070 another function called `link_socket_write` is called. This function is responsible for sending the packets with the correct protocol to the other side.

OpenVPN process outgoing link function. (forward.c)

```
1056 void
1057 process_outgoing_link (struct context *c)
1058 {
1059   struct gc_arena gc = gc_new ();
1060
1061   perf_push (PERF_PROC_OUT_LINK);
1062
1063   if (c->c2.to_link.len > 0 && c->c2.to_link.len <= EXPANDED_SIZE (&c->c2.frame))
1064     {
1065       /*
1066        * Setup for call to send/sendto which will send
1067        * packet to remote over the TCP/UDP port.
1068        */
1069       int size = 0;
1070       ASSERT (link_socket_actual_defined (c->c2.to_link_addr));
 ...
1119         /* Send packet */
1120         size = link_socket_write (c->c2.link_socket,
1121                                   &c->c2.to_link,
1122                                   to_addr);
 ...
1174   buf_reset (&c->c2.to_link);
```

The function `link_socket_write` below, only determines if the protocol to send the packet needs to be UDP or TCP. Once this is determined, the `link_socket_write` function hand of the sending of data to the determined function.

OpenVPN link socket write function. (socket.h)

```
843 /* write a TCP or UDP packet to link */
844 static inline int
845 link_socket_write (struct link_socket *sock,
846            struct buffer *buf,
847            struct link_socket_actual *to)
848 {
849   if (sock->info.proto == PROTO_UDPv4)
850     {
851       return link_socket_write_udp (sock, buf, to);
852     }
853   else if (sock->info.proto == PROTO_TCPv4_SERVER || sock->info.proto \
       == PROTO_TCPv4_CLIENT)
854     {
855       return link_socket_write_tcp (sock, buf, to);
856     }
857   else
858     {
859       ASSERT (0);
860       return -1; /* NOTREACHED */
861     }
862 }
```

As OpenVPN tunnels are based on UDP encapsulation methods to prevent the TCP stacking problem [36] , we will take a look at the `link_socket_write_udp` function. The `link_socket_write` has two options to send out data on a UDP socket. If the system is running of a Windows system, if will use the `link_socket_write_win32`. If it is a POSIX [48] capable system like Linux it will use the `link_socket_write_udp_posix` function. Because the research was done on a POSIX capable system (Linux) we will look at the `link_socket_write_udp_posix` function.

OpenVPN link socket write udp function. (socket.h)

```
831 static inline int
832 link_socket_write_udp (struct link_socket *sock,
833                struct buffer *buf,
834                struct link_socket_actual *to)
835 {
836 #ifdef WIN32
837   return link_socket_write_win32 (sock, buf, to);
838 #else
839   return link_socket_write_udp_posix (sock, buf, to);
840 #endif
841 }
```

The code snippet below shows the UDP `sendto` function [49] is used to send out the UDP packet with the encrypted data to the other endpoint of the tunnel.

OpenVPN link socket write udp posix function. (socket.h)

```
802 static inline int
803 link_socket_write_udp_posix (struct link_socket *sock,
804                  struct buffer *buf,
805                  struct link_socket_actual *to)
806 {
 ...
816     return sendto (sock->sd, BPTR (buf), BLEN (buf), 0,
817             (struct sockaddr *) &to->dest.sa,
818             (socklen_t) sizeof (to->dest.sa));
819 }
```

6.2 OpenVPN plaintext packet flow

Besides the encrypted flow as discussed in in the previous section, OpenVPN is also capable to setup a plaintext tunnel. If this is the case, OpenVPN makes use of a shortcut through the code and not call OpenSSL to encrypt packets.

The code shown below is part of the `encrypt_sign` function that is discussed in Section 6.1. The `encrypt_sign` function is also called if no encryption is used. The code shown consists of the function called `buffer_turnover` where the buffer is not encrypted but just turned over to the buffer that is written out. This is a shortcut to increase performance by not calling OpenSSL to provide null encryption.

OpenVPN encrypt sign(forward.c)

```
421 void
422 encrypt_sign (struct context *c, bool comp_frag)
423 {
424   struct context_buffers *b = c->c2.buffers;
425   const uint8_t *orig_buf = c->c2.buf.data;
426
 ...
488   /* if null encryption, copy result to read_tun_buf */
489   buffer_turnover (orig_buf, &c->c2.to_link, &c->c2.buf, &b->read_tun_buf);
490 }
```

The function `buffer_turnover` turns over the source stub of a packet to the destination stub so that the packet is ready to be sent over the VPN tunnel.

OpenVPN buffer turnover (forward.c)

```
399 /*
400  * Buffer reallocation, for use with null encryption.
401  */
402 static inline void
403 buffer_turnover  \
(const uint8_t *orig_buf, struct buffer *dest_stub, \
struct buffer *src_stub, struct buffer *storage)
404 {
405   if (orig_buf == src_stub->data && src_stub->data != storage->data)
406     {
407       buf_assign (storage, src_stub);
408       *dest_stub = *storage;
409     }
410   else
411     {
412       *dest_stub = *src_stub;
413     }
414 }
```

6.3 Simpletun UDP

As described is Section 3.2, Simpletun is a simple tunnel implementation to create a plaintext TCP tunnel. No signing or encryption is performed on packets traveling over the tunnel. The problem with the original Simpletun program is that it uses TCP to initiate a tunnel between two point over a given network. This can create a problem when a TCP connection is used over a TCP connection. This is known as the TCP stacking problem [36]. To solve the problem, Simpletun was modified to be capable of creating a UDP-based tunnel. As Simpletun (UDP) is a very basic tunnel one can set up between two endpoints, it is interesting to see the performance differences of Simpletun UDP versus other tunnel programs like Vtun and OpenVPN.

What can be seen in the code below are two functions. The first function reads data from the TAP device and writes it to the network device. The second function reads data from the network device and writes is to the TAP device. This is how Simpletun works for both the TCP and the UDP version that was written for this research. The difference between the UDP and TCP version is made in the setup of the connection between the two version.

Simpletun UDP while loop

```
327   while(1) {
328     int ret;
329     fd_set rd_set;
```

```
330
331     FD_ZERO(&rd_set);
332     FD_SET(tap_fd, &rd_set); FD_SET(net_fd, &rd_set);
333
334     ret = select(maxfd + 1, &rd_set, NULL, NULL, NULL);
335
336     if (ret < 0 && errno == EINTR){
337       continue;
338     }
339
340     if (ret < 0) {
341       perror("select()");
342       exit(1);
343     }
344
345     if(FD_ISSET(tap_fd, &rd_set)) {
346       /* data from tun/tap: just read it and write it to the network */
347
348       nread = cread(tap_fd, buffer, BUFSIZE);
349
350       /* write length + packet */
351       plength = htons(nread);
352       nwrite = cwrite(net_fd, (char *)&plength, sizeof(plength));
353       nwrite = cwrite(net_fd, buffer, nread);
354
355
356     }
357
358     if(FD_ISSET(net_fd, &rd_set)) {
359       /* data from the network: read it, and write it to the tun/tap interface.
360        * We need to read the length first, and then the packet */
361
362       /* Read length */
363       nread = read_n(net_fd, (char *)&plength, sizeof(plength));
364
365
366       /* read packet */
367       nread = read_n(net_fd, buffer, ntohs(plength));
368
369
370       /* now buffer[] contains a full packet
             or frame, write it into the tun/tap interface */
371       nwrite = cwrite(tap_fd, buffer, nread);
372
373     }
374   }
```

This part of the code is relatively simple. The `SOCK_DGRAM` states that DGRAM style packets need to be used. The option that states `0` is the option to specify that the socket needs to use UDP style sockets. This can be set to zero because the operating system checks the `SOCK_DGRAM` option to use UDP style sockets.

Because UDP does not keep a session, the source and destination address are statically set, so communication destination port and source port do not need to be negotiated. This has been done for simplicity.

Simpletun UDP socket setup

```
262   if ( (sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
263     perror("socket()");
264     exit(1);
265   }
```

Finally, the code below shows the setup of the details required for the source address and the destination address. This includes the ports that are filled in when a packet is sent over the tunnel interface or the physical network card.

Simpletun UDP socket variable setup

```
289     /* avoid EADDRINUSE error on bind() */
290     if(setsockopt(sock_fd, SOL_SOCKET,  \
          SO_REUSEADDR, (char *)&optval, sizeof(optval)) < 0) {
291       perror("setsockopt()");
292       exit(1);
293     }
294
295     memset(&local, 0, sizeof(local));
296     local.sin_family = AF_INET;
297     local.sin_addr.s_addr = htonl(INADDR_ANY);
298     local.sin_port = htons(port);
299     if (bind(sock_fd, (struct sockaddr*) &local, sizeof(local)) < 0) {
300       perror("bind()");
301       exit(1);
302     }
303
304         net_fd = sock_fd;
305
306          memset(&remote, 0, sizeof(remote));
307          remote.sin_family = AF_INET;
308          inet_aton(CLT_IP, &remote.sin_addr);
309      remote.sin_port = htons(port);
```

```
310      if (connect(sock_fd, (struct sockaddr*) &remote, sizeof(remote)) < 0) {
311         perror("connect()");
312         exit(1);
313       }
314
315
316 do_debug("\n(%s , %d) said : ",inet_ntoa(remote.sin_addr),
317                                        ntohs(remote.sin_port));
318
319
320     do_debug("SERVER: Client connected from %s \n",
321         inet_ntoa(remote.sin_addr));
322   }
```

7 Conclusion

In this section we state our conclusions of the results that were discussed in the previous sections.

When starting this project, the encryption algorithms were suspected to be the initial cause of the loss in network performance. However, by calculating the theoretical network throughput by means of OpenSSL file-based encryption, we were able to calculate the theoretical throughput of the machine. To be able to compare these to the practical network throughput, many Iperf throughput tests were performed. Results show that OpenVPN is unable to reach the same throughput as expected from the OpenSSL speed tests. The percentage of overhead we measured is around 75%.

To rule out the possibility of inefficient encryption routines within OpenVPN, different tests were performed using three different encryption ciphers, together with HMAC signing enabled or disabled. As the internal fragmentation options of OpenVPN are not needed on stable links, we disabled them on each test. This way, the Linux kernel provides the packet fragmentation. Using these different parameters, the impact of each could be determined.

A large increase in network throughput can be achieved by increasing the TUN MTU value together with disabling the internal fragmentation options of OpenVPN. The virtual TUN device is able to supply more bytes to OpenVPN, which in return passes more plaintext data to encrypt by OpenSSL. When configuring the TUN MTU with more than 9000 bytes, the internal OpenVPN fragmentation options cause a huge bottleneck. Our advise is to disable them if possible. The maximum gain in performance we measured on the lab setup was around 150% for the Blowfish-128-CBC cipher and around 30% to 80% for the AES ciphers.

OpenVPN forms a bottleneck by providing a user space VPN implementation. Analysis of the source code of OpenVPN provided insight in the internal workings. Calls to the EVP interface of OpenSSL are not the main cause the problem. However, a small overhead is caused by using the OpenSSL EVP calls to encrypt blocks of plaintext originating from the virtual TUN device. The encryption calls are most effective when supplying larger blocks to encrypt, which can be achieved by configuring the TUN MTU with a higher MTU value. The larger the block size, the less EVP calls have to be performed. An educated guess we make is that on even higher network speeds, the CPU will be occupied even more with processing the context switching and kernel interrupts caused by the physical network interface. The Future work section describes possible future points to study on this matter. Due to specific hardware being unavailable to us, not all tests could be performed. However, these are described as future research so in the case such hardware becomes available, the impact can be measured.

Many different parameters can be used on the first lab setup. In particular the

OpenVPN fragmentation options and TUN MTU size. However, on the second lab setup it is very hard to improve performance as it is not desired to tweak all separate client configurations. We found that the performance of OpenVPN relies on the version of the underlying OpenSSL library. Especially the default OpenSSL version in CentOS 5.6 causes a performance decrease because of an unoptimized FIPS implementation. Fine tuning can be performed by assigning the OpenVPN process to the CPU core that is the closest to the system bus. On our lab setup, we have seen throughput increasing with a maximum of 10%. A maximum difference in throughput of 40% was measured when comparing different cores.

In a limited time frame of three weeks of measurements we gained insight in the impact in network performance with each parameter by studying the OpenVPN network throughput using different parameters, such as encryption ciphers, signing algorithms, TUN MTU values, fragmentation options and OpenSSL versions. Even though we were unable to *exactly* pinpoint the exact cause of the performance loss, we came to interesting results showing differences in performance loss using different encryption and hashing algorithms. The proposed future work supplies for enough interesting studies

8 Future work

Due to time limitations, not every aspect was feasible to study extensively. They are discussed in this Section.

Hardware acceleration can be used to speedup encryption performance. As discussed in the paper, hardware acceleration is supported by OpenSSL.

GPUs (General Processing Unit) are opted as hardware acceleration for encryption. There are studies that implement cryptographic algorithms with a GPU [50]. There is a fundamental problem of using GPUs with cryptographic algorithms, GPUs are best used for calculating tasks that can be done in parallel, as they have many cores (shader units) available. A GPU might not be useful to boost performance for few users, as a part of the cryptographic ciphers can not be executed in parallel. However, it might boost performance for commercial VPN suppliers processing high volumes of users. A side project that stress-tests OpenVPN with GPU hardware acceleration can provide interesting results.

Related research [51] stated that using the AES-NI patch together with OpenSSL speeds up AES operations, even if the CPU does not have the instruction set build in. As the CPUs in our test machines do not have this instruction set, we were interested in the results. After patching OpenSSL using the AES-NI patch, and comparing performance, we found that there is no gain in performance. However, future research will be able to show the impact of using such extensions on AES-NI capable hardware.

Another more common method is to use cryptographic acceleration cards as a method of hardware acceleration.

Kernel mode As OpenVPN is a VPN solution that runs in user space, many operations are needed to switch between kernel space and user space to process packets. An additional side study we encountered is to study the influence of using OpenVPN in Kernel Mode Linux [52]. Kernel Mode Linux is an implementation of Linux which enables user space applications to run in kernel space. The advantage of this is that the overhead of context switching can be avoided. According to the author of OpenVPN, James Yonan, this will not speed up performance [53] , but it might on 1 Gbps connections. A trade-off between stability and performance will have to be made, but it might be interesting to measure the difference in performance here.

TAP-Win32 driver In case of the Windows operating system a special TAP-Win32 driver needs to be installed, which is also developed and supplied by OpenVPN. However, development of this driver is not very popular. Future research is needed to measure which possible bottlenecks are introduced by the Windows drivers.

10 Gbps performance measurements The maximum speed of network interfaces is increasing when using optical technologies. On these speeds, TCP tuning is already required to reach near-line-speed. It will be interesting to see what the performance of OpenVPN is on 10 Gbps links. Initial research has already been done on this [51] and shows a 60% drop in performance on a transparent tunnel without special tuning done.

Profiler is a tool that is able to perform low-level performance measurements. Low-level Linux performance counters provide insight in the bottlenecks of certain applications, (kernel) modules and data flows. It even is able to observe slowdowns in pieces of code. Due to the limited time frame of this project, we only looked into this method of performance measurements briefly, as the resulting output of the measurements have a steep learning curve to understand.

IPsec Comparing measurements done on IPsec to OpenVPN performance measurements would provide insight in the differences in implementation (kernel space versus user space).

A Server hardware

In the lab setups described in Section 4.1 , four machines were used that all have equal hardware specifications.

Brand Dell

Model PowerEdge R210

CPU Intel(R) Xeon(R) CPU L3426 @ 1.87GHz

Memory 8GB

NIC Embedded Broadcom 5716 (x2)

- **Brand** Broadcom Corporation
- **Model** NetXtreme II BCM5716
- **Revision** Revision 20
- **Standard** Gigabit Ethernet

Linux kernel 2.6.18-238.el5 #1 SMP Thu Jun 23 15:51:15 EST 2011 x86_64 x86_64 x86_64 GNU/Linux

B Measurement script

All measurements described in Section 5 were done by using a BASH script. The basic principle of this script is that it runs from an external host controlling all machines in the lab setup. The script enables automation of the tests, and writes the Iperf measurement results to a file on one of the machines in the lab setup. Altering of the script was done to be able to operate on the second lab setup, which consisted of four machines instead of two. Both scripts are in the following sections below.

B.1 Lab setup 1

Measurement script for lab setup 1.

```
#!/bin/bash

# Kill OpenVPN and Iperf on both VPN machines.
ssh root@paris.studlab.os3.nl 'kill -9 `pidof openvpn` &> /dev/null'
ssh root@paris.studlab.os3.nl 'kill -9 `pidof iperf` &> /dev/null'
ssh root@vpn.studlab.os3.nl 'kill -9 `pidof openvpn` &> /dev/null'
ssh root@vpn.studlab.os3.nl 'kill -9 `pidof iperf` &> /dev/null'
```

```
# Start the VPN server connection.
ssh root@paris.studlab.os3.nl 'openvpn --daemon \
--config /etc/openvpn/config/default-static-server.cfg'
# Sleep for 2 seconds for the OpenVPN server to start up.
sleep 2
# Start the VPN client connection.
ssh root@vpn.studlab.os3.nl 'openvpn --daemon \
--config /etc/openvpn/config/default-static-client.cfg'
# Sleep for 5 seconds for the OpenVPN client to
# start up and tunnel establishment.
sleep 5
# Start Iperf in server mode.
ssh root@paris.studlab.os3.nl 'nohup iperf -s \
> foo.out 2> foo.err < /dev/null &'
# Start Iperf in client mode, connect it to the Iperf
# server over the VPN tunnel, test for 5 minutes (3 times)
ssh root@vpn.studlab.os3.nl 'nohup iperf -c 10.0.1.1 -t 300 \
>> ~/results.txt 2> foo.err < /dev/null &'
# Sleep for 6 minutes to let Iperf finish and the network
# to get clean.
sleep 360
ssh root@vpn.studlab.os3.nl 'nohup iperf -c 10.0.1.1 -t 300 \
>> ~/results.txt 2> foo.err < /dev/null &'
sleep 360
ssh root@vpn.studlab.os3.nl 'nohup iperf -c 10.0.1.1 -t 300 \
>> ~/results.txt 2> foo.err < /dev/null &'
sleep 360
```

B.2 Lab setup 2

Measurement script for lab setup 2.

```
#!/bin/bash

# Kill OpenVPN and Iperf on both VPN machines.
ssh root@paris.studlab.os3.nl 'kill -9 `pidof openvpn` &> /dev/null'
ssh root@paris.studlab.os3.nl 'kill -9 `pidof iperf` &> /dev/null'
ssh root@vpn.studlab.os3.nl 'kill -9 `pidof openvpn` &> /dev/null'
ssh root@vpn.studlab.os3.nl 'kill -9 `pidof iperf` &> /dev/null'

# Set static route to brussels via vpn1.
ssh root@bern.studlab.os3.nl 'ip route add 145.100.104.43/32 \
via 145.100.104.56'
# Set static route to bern via vpn2.
```

```
ssh root@brussels.studlab.os3.nl 'ip route add 145.100.104.42/32 \
via 145.100.104.36'

# Start the VPN server connection.
ssh root@paris.studlab.os3.nl 'openvpn --daemon --config \
/etc/openvpn/config/no-cipher-no-hmac-static-server.cfg'
# Sleep for 2 seconds for the OpenVPN server to start up.
sleep 2
# Start the VPN client connection.
ssh root@vpn.studlab.os3.nl 'openvpn --daemon --config \
/etc/openvpn/config/no-cipher-no-hmac-static-client.cfg'
# Sleep for 5 seconds for the OpenVPN client to start up
# and tunnel establishment.
sleep 5
# Set static route to brussels via the tun0 virtual interface.
ssh root@paris.studlab.os3.nl 'ip route add 145.100.104.43/32 \
via 10.0.1.1'
# Set static route to bern via the tun0 virtual interface.
ssh root@vpn.studlab.os3.nl 'ip route add 145.100.104.42/32 \
via 10.0.1.2'

# Start Iperf in server mode.
ssh root@brussels.studlab.os3.nl 'nohup iperf -s > foo.out 2> foo.err < /dev/null &'
# Start Iperf in client mode, connect it to the Iperf server over
# the VPN tunnel, test for 5 minutes (3 times).
ssh root@bern.studlab.os3.nl 'nohup iperf -c 145.100.104.43 -t 300 \
>> /root/test-results/bf-transparant.txt 2> foo.err < /dev/null &'
sleep 360
ssh root@bern.studlab.os3.nl 'nohup iperf -c 145.100.104.43 -t 300 \
>> /root/test-results/bf-transparant.txt 2> foo.err < /dev/null &'
sleep 360
ssh root@bern.studlab.os3.nl 'nohup iperf -c 145.100.104.43 -t 300 \
>> /root/test-results/bf-transparant.txt 2> foo.err < /dev/null &'
sleep 360
```

C Simpletun UDP code

The code for the Simpletun UDP modifications is located at: `https://www.os3.nl/_media/2010-2011/students/simpletun_udp.txt`

References

[1] Check Point Software Technologies Ltd. Vpn-1 power multi-core. Website; http://www.checkpoint.com/products/vpn-1_power_multicore/index.html. [Online; consulted on June 21, 2011].

[2] Array Networks. Array networks' industry leading ssl vpn performance validated by real-world testing. Website; http://www.arraynetworks.net/cms-array-press-releases-array-networks-industry-leading-ssl-vpn-performance-validated-html. [Online; consulted on June 21, 2011].

[3] OpenVPN.net. Openvpn.net website. https://www.openvpn.net, May 2011. [Online; Consulted on May 30, 2011].

[4] Google. Google fiber. Website; http://googleblog.blogspot.com/2010/02/think-big-with-gig-our-experimental.html, February 2010. [Online; Consulted on June 7, 2011].

[5] Angela Orebaugh Sheila Frankel, Paul Hoffman and Richard Park. National institute of standards and technology - guide to ssl vpns. Website; http://csrc.nist.gov/publications/nistpubs/800-113/SP800-113.pdf, July 2008. [Paper; Consulted on June 14, 2011].

[6] Wikipedia: Hmac. Website; http://en.wikipedia.org/wiki/HMAC. [Online; consulted on June 23, 2011].

[7] K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, and G. Zorn. Point-to-Point Tunneling Protocol (PPTP). RFC 2637 (Informational), July 1999.

[8] W. Townsley, A. Valencia, A. Rubens, G. Pall, G. Zorn, and B. Palter. Layer Two Tunneling Protocol "L2TP". RFC 2661 (Proposed Standard), August 1999.

[9] B. Patel, B. Aboba, W. Dixon, G. Zorn, and S. Booth. Securing L2TP using IPsec. RFC 3193 (Proposed Standard), November 2001.

[10] W. Simpson. The Point-to-Point Protocol (PPP). RFC 1661 (Standard), July 1994. Updated by RFC 2153.

[11] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

[12] The OpenSSL Project. Openssl: The open source toolkit for ssl/tls. Website; http://www.openssl.org. [Online; Consulted on June 20, 2011].

[13] Maxim Krasnyansky and Bishop Clark. Vtun - tun/tap devices. Website; http://vtun.sourceforge.net/tun/. [Online; consulted on June 23, 2011].

[14] Wikipedia: User space. Website; http://en.wikipedia.org/wiki/User_space. [Online; consulted on June 10, 2011].

[15] Context switching. Website; http://osr507doc.sco.com/en/PERFORM/context_switching_cpu.html. [Online; consulted on June 28, 2011].

[16] Open Maniak. Openvpn - the easy tutorial - introduction. Website; http://openmaniak.com/openvpn.php. [Online; consulted on June 21, 2011].

[17] Ssl acceleration. Website; http://www.kegel.com/ssl/hw.html. [Online; consulted on June 11, 2011].

[18] Intel. Intel®advanced encryption standard instructions (aes-ni). Website; http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/, May 2011. [Online; Consulted on May 30, 2011].

[19] Openssl aes-ni patch for 0.9.8 branch. Website; http://rt.openssl.org/Ticket/Display.html?id=2067&user=guest&pass=guest. [Online; consulted on June 15, 2011].

[20] VIA. Via padlock security engine. Website, http://www.via.com.tw/en/initiatives/padlock/hardware.jsp. [Online; consulted on June 20 , 2011].

[21] SANS.org. Computer security training, network research and resources. Website; http://www.sans.org. [Online; Consulted on June 14, 2011].

[22] Charlie Hosner. Openvpn and the ssl vpn revolution. Website; http://www.sans.org/reading_room/whitepapers/vpns/openvpn-ssl-vpn-revolution_1459, 2004. [Paper; Consulted on June 14, 2011].

[23] Jan Just Keijser. Openvpn 2 cookbook. PACKT Publishing, February 2011.

[24] H. M. Abdul Kader D. S. Abdul. Elminaam and M. M. Hadhoud. Performance evaluation of symmetric encryption algorithms. Website; http://www.ibimapublishing.com/journals/CIBIMA/volume8/v8n8.pdf, 2009. [Journal; Consulted on June 20, 2011].

[25] Kshirasagar Naik and David S. L. Wei. Software implementation strategies for power-conscious systems, June 2001. [Online; http://portal.acm.org/citation.cfm?id=383768].

[26] A A. Nadeem and M.Y. Javed. A performance comparison of data encryption algorithms, August 2005. [Online; http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1598556].

[27] Michael Hall. Performance analysis of openvpn on a consumer grade router. http://www1.cse.wustl.edu/~jain/cse567-08/ftp/ovpn/index.html, November 2008.

[28] Davide Brini. Tun/tap interface tutorial. Website; http://backreference.org/2010/03/26/tuntap-interface-tutorial/. [Online; consulted on June 22, 2011].

[29] Maxim Krasnyansky and Bishop Clark. Vtun - virtual tunnels over tcp/ip networks. Website; http://vtun.sourceforge.net. [Online; consulted on June 22, 2011].

[30] tinc - welcome to tinc! Website; http://www.tinc-vpn.org/. [Online; consulted on June 26, 2011].

[31] Iperf website. Website; http://sourceforge.net/projects/iperf/. [Online; consulted on Februari 3th, 2011].

[32] Ajay Tirumala, Les Cottrell, and Tom Dunigan. Measuring end-to-end bandwidth with iperf using web100, 2003.

[33] nuttcp website. Website; http://nuttcp.org. [Online; consulted on June 3, 2011].

[34] Yixin Wu, Suman Kumar, and Seung-Jong Park. Measurement and performance issues of transport protocols over 10gbps high-speed optical networks. *Computer Networks*, 54(3):475 – 488, 2010.

[35] Netperf website. Website; http://www.netperf.org/netperf/. [Online; consulted on June 3, 2011].

[36] Olaf Titz. Why tcp over tcp is a bad idea. Website; http://sites.inka.de/~W1011/devel/tcp-tcp.html. [Online; consulted on June 9, 2011].

[37] Openvpn.net 2.1 manual. Website; http://openvpn.net/index.php/open-source/documentation/manuals/69-openvpn-21.html. [Online; consulted on June 22, 2011].

[38] Valtteri Niemi N. Asokan and Kaisa Nyberg. Man-in-the-middle in tunnelled authentication protocols. Website; http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.5836&rep=rep1&type=pdf. [Paper; consulted on June 26, 2011].

[39] Andrew C Yao Xiaoyun Wang and Frances Yao. Cryptanalysis on sha-1. Website; http://csrc.nist.gov/groups/ST/hash/documents/Wang_SHA1-New-Result.pdf. [Paper; consulted on June 26, 2011].

[40] Ondrej Mikle. Practical attacks on digital signatures using md5 message digest. Website; http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.7624&rep=rep1&type=pdf. [Paper; consulted on June 25, 2011].

[41] Wikipedia: Maximum transmission unit. Website; http://en.wikipedia.org/wiki/Maximum_transmission_unit. [Online; consulted on June 22, 2011].

[42] Wikipedia: Jumbo frame. Website; http://en.wikipedia.org/wiki/Jumbo_frame. [Online; consulted on June 25, 2011].

[43] Wikipedia: Maximum segment size. Website; http://en.wikipedia.org/wiki/Maximum_segment_size. [Online; consulted on June 22, 2011].

[44] Improved linux* smp scaling: User-directed processor affinity. Website; http://software.intel.com/en-us/articles/improved-linux-smp-scaling-user-directed-processor-affinity/. [Online; consulted on June 16, 2011].

[45] Abdel-Karim Al Tamimi. Performance analysis of data encryption algorithms. http://www1.cse.wustl.edu/~jain/cse567-06/ftp/encryption_perf/index.html.

[46] Openssl fips object module. Website; http://www.openssl.org/docs/fips/SecurityPolicy-1.2.pdf. [Online; consulted on June 18, 2011].

[47] OpenSSL. Evp(3). Website, http://www.openssl.org/docs/crypto/evp.html. [Online; consulted on June 20 , 2011].

[48] The IEEE and The Open Group. Posix.1-2008. Website, http://pubs.opengroup.org/onlinepubs/9699919799/. [Online; consulted on June 19 , 2011].

[49] David Mertz. Programming linux sockets, part 2: Using udp. Website, http://developerspoint.files.wordpress.com/2008/07/programming-linux-sockets-part-2-using-udp.pdf. [Online; consulted on June 20 , 2011].

[50] Svetlin Manavski. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. Website; http://www.manavski.com/downloads/PID505889.pdf. [Paper; consulted on June 25, 2011].

[51] Jan Just Keijser. Optimizing performance on gigabit networks. https://community.openvpn.net/openvpn/wiki/Gigabit_Networks_Linux, May 2011. [Online; Consulted on May 30, 2011].

[52] Kernel mode linux : Execute user processes in kernel mode. Website; http://web.yl.is.s.u-tokyo.ac.jp/~tosh/kml/. [Online; consulted on June 26, 2011].

[53] Openvpn.net mailing list post. Website; http://openvpn.net/archive/openvpn-users/2004-11/msg00548.html. [Online; consulted on June 25, 2011].