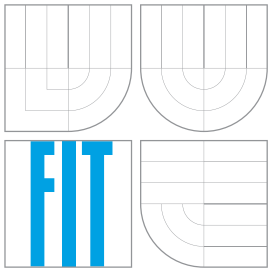
BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

LINUX VPN PERFORMANCE AND OPTIMIZATION

OPTIMALIZACE VÝKONU VPN V LINUXU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — Bc. FRIDOLÍN POKORNÝ
AUTOR PRÁCE

SUPERVISOR — Ing. TOMÁŠ KAŠPÁREK
VEDOUCÍ PRÁCE

BRNO 2016

Brno University of Technology - Faculty of Information Technology

Computer Centre | Academic year 2015/2016

Master Thesis Specification

For: **Pokorný Fridolín, Bc.**
Branch of study: Information Technology Security
Title: **Linux VPN Performance and Optimization**
Category: Operating Systems

Instructions for project work:

1. Study the principles of VPN in Linux, kernel and user-space support and systems used. Verify the operation scheme for at least OpenVPN and OpenConnect systems.
2. Develop a set of high and/or low level benchmarks and tests to reveal problematic parts in VPN processing pipeline.
3. Identify several parts that can be optimized. Device and implement proposed optimization either in user-space or in kernel.
4. Test implemented optimizations and discuss improvements reached together with possibilities for future work.

Basic references:

- OpenVPN, https://openvpn.net/, navštíveno 2015-10-27
- OpenConnect, http://www.infradead.org/openconnect/, navštíveno 2015-10-27
- Kolesnikov O., Hatch B.: Building Linux Virtual Private Networks, 2002, New Riders, ISBN: 978-1578702664

Requirements for the semestral defense:
Items 1 and 2.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Master Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Kašpárek Tomáš, Ing.**, CC FIT BUT
Beginning of work: November 1, 2015
Date of delivery: May 25, 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

Dušan Kolář
Associate Professor and Head of Department

Abstract

This thesis provides an analysis of the available software VPN solutions and its performance on the Linux system. This analysis is then used as a basis to determine performance bottlenecks, suggest performance improvements and further design and implement the most promising of them. The result of this thesis is a Linux kernel module which does TLS and DTLS transmission and reception in kernel space. The module utilizes key material established during a TLS or DTLS handshake in user space. Despite the fact that our developed module was designed for use by VPNs we identified several other use-cases which can take advantage of our module.

Abstrakt

Tato práce se zabývá analýzou stávajících a aktivních VPN řešení, jejich výkonu a slabých stránek. Výsledkem práce je jaderný modul pro Linux, který implementuje datový přenos pomocí protokolů TLS a DTLS na základě konfigurace ustanoveného spojení v chráněném režimu. Primárním cílem bylo odstranit datové kopie a změny kontextu z chráněného režimu do režimu jádra během datových přenosů ve VPN řešeních založených na protokolech TLS a DTLS. Práce analyzuje cenu těchto operací a na základě analýz lokalizuje další kroky nutné k využití implementovaného jaderného modulu ve VPN řešeních. Práce se dále zabývá analýzou dalších možných využití implementovaného jaderného modulu mimo VPN řešení.

Keywords

VPN, Linux, optimization, TLS, OpenConnect, DTLS, security

Klíčová slova

VPN, Linux, optimalizace, TLS, OpenConnect, DTLS, bezpečnost

Reference

POKORNÝ, Fridolín. *Linux VPN Performance and Optimization.* Brno, 2016. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Kašpárek Tomáš.

Linux VPN Performance and Optimization

Declaration

I declare that I worked on this thesis on my own and that I used only resources mentioned in the Bibliography section. The work was done under the guidance of Ing. Tomáš Kašpárek and Nikos Mavrogiannopoulos.

.....
Fridolín Pokorný
May 25, 2016

Acknowledgements

I would like to express my gratitude to my supervisor Ing. Tomáš Kašpárek for the useful comments and remarks. Furthermore I would like to thank Nikos Mavrogiannopoulos, PhD. for introducing me to the topic, guidance, patience and the support on the whole way. Last but not least I would like to thank Red Hat Czech for providing hardware necessary for development and testing.

Contents

Chapter 1

Introduction

A Virtual Private Network, also commonly known abbreviated as VPN, is a technology, which enables users to connect to local area networks (LAN) from a wide area network (WAN). After establishing a VPN connection, a user can both access and use resources provided by LAN as if they have been accessed from the local network. To respect LAN's restrictions and rules it is necessary to secure an established VPN connection.

Nowadays, this technology is widespread in corporations, which expose devices situated in LAN via VPN for employees. By enabling to establish remote access to the local network, employees can access remote machines, printers or servers without physical access or the need to be directly connected to LAN. This gives employees the power to easily work remotely.

This work is mainly focused on optimization of existing tools which directly or indirectly implement VPN solutions. The main focus is given to open source solutions because of open implementation and the available source codes.

The thesis is organised as follows: In the first section 1 one can find the basic principles of VPN and available open source VPN tools. There are introduced available open source projects of VPN solutions, their liveness, upstream status and their primary focus. There, it is also stated the importance of analysing weak parts and bottlenecks. In the following section 2, one can find an analysis of current implementation status, analysis of the packet path and transmission in kernel and user space, list of available ciphers within solutions and their security and speed impact. Benchmarks and environment created for benchmarks are analysed in the section 3. The section 4 is focusing on work done in order to bring optimized solution. In the following section, 5 is analysed an implemented solution based on results. Based on these results, future work that needs to be done in section 6 is discussed. The last section, 7 summarizes this thesis.

1.1 The Purpose of Optimizing a VPN

As stated in the introduction, VPNs are used widely in order to provide secured a connection to remote LANs and access servers, printers and other network devices situated in a LAN remotely. This concept is suitable for small offices or even for large corporations which want to provide employees with remote access.

Introducing a VPN to a network involves additional computation and memory consumption for the involved network devices, which need to deal with packet encryption, packet decryption and routing. The additional load can be partially reduced with well

designed optimizations. The purpose of this thesis is to analyse current open source VPN solutions, find weak points and based on discovered weak points, implement optimizations which could increase VPN performance.

There are currently available three main VPN projects, OpenConnect [7, 8], OpenVPN [11] and Libreswan [6]. These projects are introduced in the following sections.

A term `context switch` in the text refers to a context switch from user space to kernel space and vice versa, if not stated otherwise. There is also used a term `record`, which stands for one unit of the referenced protocol type (TLS or DTLS).

1.2 OpenConnect

OpenConnect is the youngest VPN project apart from OpenVPN and Libreswan. OpenConnect was originally designed as an open source implementation of CISCO's proprietary AnyConnect VPN, and, according to authors [39], it should be capable to interoperate with AnyConnect VPN. The implementation consists of two applications: OpenConnect client [7] and OpenConnect server (also known as ocserv) [8].

The OpenConnect project was originally started by David Woodhouse, who is currently upstream maintainer too. Later on, OpenConnect was enhanced to support Pulse Connect Secure protocol (originally known as Jupiter SSL VPN). Both OpenConnect server and OpenConnect client are released under the terms of Gnu Lesser Public License, version 2.1. Both support all major platforms, such as Linux (including Android), various BSD distributions (like FreeBSD, OpenBSD), Mac OS X, OpenSolaris, Solaris 10/11, Windows and Mac OS X [7, 8].

OpenConnect server and OpenConnect client support IPv4 and IPv6 networking. OpenConnect project and AnyConnect VPN respectively, deliver a new protocol based on exchanging XML messages for authentication. They are designed to use existing TLS [15] and DTLS [17] protocols to deal with security. These protocols handle transmission over both reliable and unreliable layers. DTLS and TLS were not reimplemented, OpenConnect uses GnuTLS library in order to serve DTLS and TLS records. GnuTLS [23] library is open source as well and is handled by the Gnu project [21].

OpenConnect is referred to be an "SSL based VPN". The abbreviation SSL can be substituted with more accurate TLS, but the term "SSL based VPN" is spread nowadays because of historical reasons.

Project OpenConnect is very live and the community behind this project is relatively active. More information can be found at project's home pages [7, 8].

1.3 OpenVPN

OpenVPN is currently the biggest open source VPN project. It was started by James Yonan and the very first usable release was published in 2001. OpenVPN is currently maintained by organization called OpenVPN Technologies, Inc. Nowadays, OpenVPN is the most used VPN application [11].

OpenVPN is currently available on all major platforms – Linux (including Android), Solaris, OpenBSD, FreeBSD, NetBSD, Mac OS X, and Windows XP or newer. It is released under the terms of Gnu GPL license, version 2. OpenVPN is registered trademark of OpenVPN Technologies, Inc.

OpenVPN is tightly bound to OpenSSL library [9] and uses its capabilities to secure a connection. It supports IPv4, even IPv6 and OpenVPN is capable to operate over reliable or even unreliable networks. There are no separate client and server applications – instead, OpenVPN provides one user space application, which can act like a client or a server based on configuration and command line arguments.

OpenVPN introduced its own protocol used to transfer encrypted messages. This protocol is not standardized and is currently only used within OpenVPN application, so called OpenVPN protocol. This protocol is similar to the TLS protocol. More information about OpenVPN can be found at the project's home page [11], deep analysis of OpenVPN cryptography can be found in documentation [12].

1.4 Libreswan

Libreswan is not a standalone pure VPN application. It is based on a standardized VPN protocol based on IPsec and IKE (Internet Key Exchange) format. Both protocols are maintained by Internet Engineering Task Force (IETF) [24].

Libreswan was introduced back in 1997, originally named FreeS/WAN. The original authors are John Gilmore and Hugh Daniel. The project was later renamed to Openswan in 2003 but from 2012 its name was forced to change to Libreswan due to legal issues. Libreswan is released under the terms of Gnu GPL license, version 2 [6].

Libreswan runs on Linux 2.4 or higher, FreeBSD and Apple OSX. Rather then introducing a standalone VPN user space client, Libreswan uses built-in IPsec stack (XFRM/NETKEY) or its own stack called KLIPS. As can be seen, Libreswan operates mostly in kernel space and uses other kernel parts in order to operate. Libreswan relies on IPsec technology, thus it tends to hide communication details, such as IP protocol version.

In order to secure the connection, Libreswan uses Mozilla's Network Security Services, also known as NSS. Even NSS is a user space library, it is utilized by Libreswan IKE daemon (pluto) for cryptographic operations.

Libreswan is currently an active project. It is maintained by community called The Libreswan Project. Even though Libreswan is based on IPsec and extends this technology to serve VPN connections, it is considered to be a VPN solution. Documentation, source codes and project's details can be found at the project's home page [6].

Chapter 2

Current VPN Technologies

In this section one can find a brief overview of OpenConnect, OpenVPN and Libreswan implementation status. We describe their architectures, dependencies and provide protocol analysis. An overview of record transmission and encryption handling are described as well, with respect to supported encryption algorithms.

2.1 User Space and Kernel Space

In the section 1, there was introduced three VPN solutions, which are currently available and used. To generalize these solutions, we can distinguish the solutions into two main categories – kernel space solutions and user space solutions.

The introduced user space solutions, OpenConnect and OpenVPN, are pure user space applications. They use kernel only for sending and receiving records. All operations, such as record assembling, record encoding, record disassembling, are implemented in user space. The kernel only provides an interface to handle and forward IP packets. Such implementation of the interface is called TUN/TAP device [36]. Both OpenConnect and OpenVPN use TUN/TAP device in order to send and receive packets.

Even though the VPN term is often referred to a user space solution, there are available implementations which handle VPN inside kernel. This could benefit when running VPN on heavy traffic. There can be omitted user space operations which need context switches every time a record is going to be handled by user space application. To support this, test comparison scenarios for user space versus kernel space implementations could be evaluated.

Implementing all user space applications and protocols to kernel space would lead to completely omit user space and kernel space separation. Implementing VPN in kernel space does not mean to force all user space implementations (not only VPN applications) to the kernel. This would lead to enlarging kernel and would lead to a huge and difficult to manage kernel. All user space benefits (memory management, security, process management and much more) would be lost. Implementing some parts of protocols to the kernel can benefit specialized machines and optimize their performance. These optimizations for specialized machines should be possible to turn on when required.

2.1.1 TUN/TAP Device Driver

The TUN/TAP driver is a kernel interface, which provides IP packet reception and transmission for user space programs. User space programs can use read and write operations on an appropriate file in order to inject IP packets into networking stack. Receiving and

sending is transparent to the user space applications and all of the details like type of media are hidden [36].

An application, which wants to use TUN/TAP driver, has to open `/dev/net/tun` and register a network device with the kernel by calling `ioctl(2)`. From version 3.8, Linux supports multiqueue TUN/TAP. This approach is suitable for multicore systems, where sending or receiving records can be parallelized. Unfortunately, this approach has minimal positive impact on single core systems. Moreover, it could have a negative impact, since multiple resources are allocated but none of them are used at the same time. This leads to degradation to serialization as if no multiqueue TUN/TAP would be used [36]. This optimization is used in OpenVPN in multiplexed mode [12].

The TUN/TAP driver can operate on both Ethernet frames and IP frames. TAP driver is used for Ethernet frames and TUN driver is used for IP frames [36].

A more detailed explanation of a packet path can be found in [31].

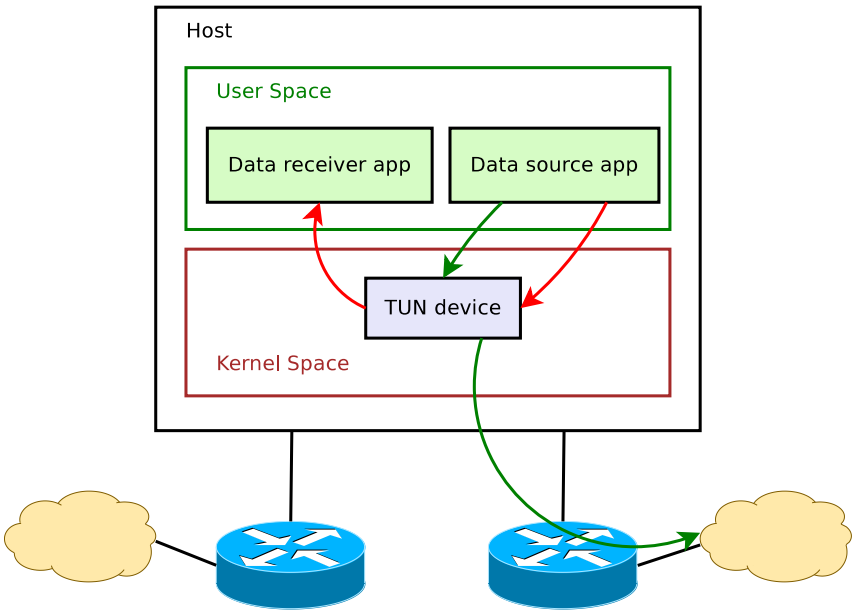


Figure 2.1: A diagram of a communication flow using TUN/TAP device driver

The functionality of TUN/TAP device driver is illustrated on figure 2.1. Data source application (*Data source app*) is sending data to TUN device using `write(2)` operation. The application knows only the destination IP of the receiver. After sending records to the TUN device, all decisions on record routing are done in the kernel. Kernel, based on configured routing tables and knows how to handle the record. Data receiver application (*Data receiver app*) can be situated on local or even on a remote host. Based on routing, records are send to local application (red path) or to a remote machine using appropriate network device (green path).

2.2 OpenConnect

OpenConnect's protocol uses the TLS [15], Datagram TLS protocol (DTLS) [17] and the HTTP protocol. It was designed to operate on DTLS (UDP), but can fallback to TLS (TCP) if necessary.

The OpenConnect VPN protocol supports three main types of client authentication:

- password
- certificate
- HTTP SPNEGO (Kerberos, GSSAPI)

All methods use XML sent over the HTTP protocol secured with TLS or DTLS based on network [39].

OpenConnect VPN protocol is designed to be a client – server protocol. The server can be configured to route desired subnets or act like a default gateway for all connections from a client. It usually listens on port 443 because of TLS used, but it can be configured to listen on any desired port. All control messages from server to client are in an XML format. The main advantage of OpenConnect VPN protocol is that it uses the standardized protocol TLS and DTLS for communication. Any communication done in a VPN tunnel seems to an observer to ordinary encrypted HTTPS traffic.

There can be used various ciphers to encrypt communication between client and server. Since OpenConnect uses GnuTLS, supported ciphers are subset of ciphers available in GnuTLS [23]. The current implementation of OpenConnect VPN server supports AES128 GCM, AES256 GCM, AES128 SHA and DES CBC3 SHA ciphers. Ciphers are managed upstream and during handshake, when establishing secured connection, negotiation of ciphers is completed. There is pre-configured cipher priority, but all ciphers could be prioritized or even disabled if necessary (via priority strings [22]).

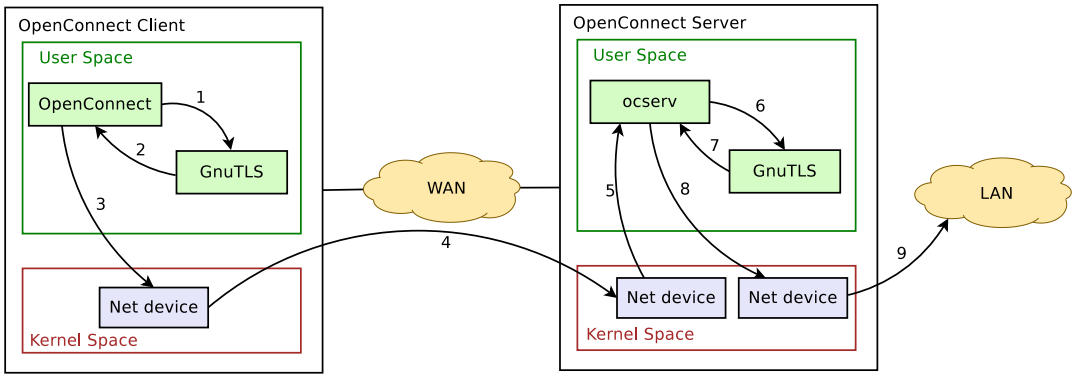


Figure 2.2: A diagram of a record transmission in OpenConnect

As mentioned above, when establishing a secured connection there is a TLS handshake done. After the handshake, data can be sent. On the figure 2.2 there is illustrated data flow after the handshake. The OpenConnect client wants to send data to OpenConnect Server (ocserv application). Record, which should be used in remote LAN, is encrypted in GnuTLS library using cryptographic functions; this is done in user space. An encrypted record is after that sent to an appropriate network device and travels through the network. The OpenConnect client reads record from network device using `read(2)` operation. Record has to be decrypted using negotiated algorithm using GnuTLS library. After decryption, the packet is sent from OpenConnect server to TUN device in order to packet record to desired host in ocserv's local network network.

2.2.1 Transport Layer Security – TLS

Transport Layer Security, also referred as TLS, is a mechanism used to authenticate and establish secured session between two communicating nodes [15]. TLS was introduced in 1999 by Internet Engineering Task Force (IETF) [24] as a successor of SSL, which was originlly developed by Netscape Communications. Nowadays, SSL and TLS terms are often exchanged due to historical reasons. TLS uses asymmetric cryptography in order to exchange keys for symmetric cryptography. After key exchange only symmetric cryptography is used due to speed.

The TLS protocol is mostly connected with web browsers which use TLS in HTTPS (Hypertext Protocol Secure).

TLS is suitable to be used over a reliable transport layer such as TCP. On the other hand, datagram TLS (DTLS) [17] is designed to be used on reliable as well as on unreliable transport layer, such as UDP.

An open source implementation of TLS in C programming language can be found in a Gnu project called GnuTLS [23]. Its original author is Nikos Mavrogiannopoulos. It supports various TLS and DTLS versions – TLS 1.2, TLS 1.1, TLS 1.0, DTLS 1.0 DTLS 1.2. and even SSL 3.0. There can be used various authentication standards (e.g. X.509) in order to verify certifications of communicating nodes. The implementation is split into three cooperating parts – *TLS protocol part* (TLS protocol itself), *certificate part* (certificate parsing and verification based on `libtasn1` library) and *cryptographic back-end* (based on `nettle` and `gmplib` libraries) [23].

2.3 OpenVPN

OpenVPN provides two main authentication modes on cryptographic layer [12]:

- Static key
- SSL/TLS mode

When static key mode is used, keys used in session are exchanged between clients before the tunnel is established. On the other hand in SSL/TLS mode an TLS session is established with bidirectional authentication. If both clients are authenticated, keys used in session are randomly generated [12].

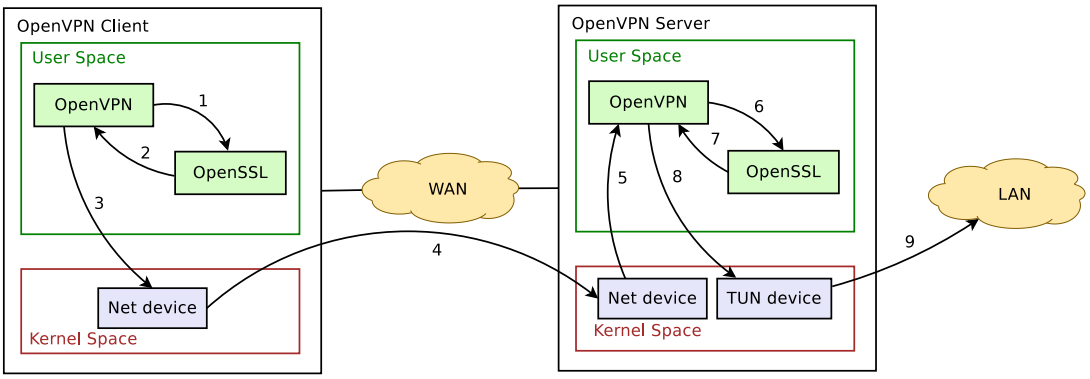


Figure 2.3: A diagram of a record transmission in OpenVPN

OpenVPN is designed to be a client – server protocol, but unlike OpenConnect, OpenVPN uses its own protocol for client – server communication. A different approach can be seen in the way how clients are accessed as well. OpenVPN server acts like a NAT and based on destination port it multiplexes communication to clients [12]. This way of implementing VPN is sparing resources but it consumes additional computational power on the server. Moreover, it is not easy to parallelize communication on server side and the current implementation does not support it. Running OpenVPN on multicore system does not benefit the communication in any way.

As can be seen on figure 2.3, record path is very similar to OpenConnect's path. When a client wants to send record to a server, record is encrypted using OpenSSL. Encrypted and encapsulated record is then sent to OpenVPN server. A record is then decrypted and routed via TUN device to appropriate host in server's LAN.

2.3.1 OpenSSL

OpenSSL [9] was the very first open source library that implemented a secure communication based on SSL. It contains implemented TLS and DLTS protocols (DTLS 1.2 is still in beta). It supports modern cryptographic algorithms and hash functions.

OpenSSL's license is not compatible with Gnu project licensing, so GnuTLS was introduced as a substitution of OpenSSL. There exist various forks of OpenSSL, which try to be compatible with OpenSSL, such as LibreSSL [5].

2.4 Libreswan

Libreswan VPN is based on IPsec [6]. It's main advantage over OpenConnect and OpenVPN is the optimization of doing all operations on encrypted records in kernel space. This optimization saves context switches and user space operations that can be moved to kernel.

On the other hand, Libreswan VPN does not provide a way how to count transmitted record between peers, there is no easy way how to limit speed for certain clients or do more advanced operations like client idle timeout (even some could be done by kernel, but most of them strongly depend on the kernel for providing such features). Libreswan creates a virtual tunnel between two hosts. This can be an disadvantage when using broken firewalls too. ESP packets, which are used to encrypt connection, can be restricted in desired network. In this situation it is not possible to use IPsec in general.

Currently, Libreswan is the fastest open – source way to use VPN on Unix-like systems supporting IPsec. More about IPsec and its performance analysis can be found in [44], The main advantage is in kernel space optimizations.

2.4.1 Crypto API in Linux Kernel

The early Linux kernel versions introduced a cryptographic framework. This cryptographic framework allowed to expand kernel features and gave birth to implementations of disk encryptions in kernel or IPsec implementation.

Crypto API in Linux is accessible to kernel modules and kernel itself. It implements all well – known block ciphers and hash functions [26]. IPsec uses Linux Crypto API to do encryption in kernel. One of the major advantages of Linux Crypto API is the ability to use transparently cryptographic accelerators [1, 40, 41],

Chapter 3

Benchmarks and Identifying Bottlenecks

This section discusses initial benchmarks done in testing environment, which was created in order to study the principles, compare and test available VPN solutions. The testing environment is explained in detail and illustrated in this section.

3.1 Testing Environment

Originally, tests were done on a simulated environment made on Red Hat's OpenStack platform [10]. This platform provides easy to set up and run prepared images of various Gnu/Linux distributions. All images run in a pool of operating system instances. Unfortunately, this testing environment was not suitable for VPN testing purposes, since the software network was very restricted and traffic limits were easily reached. Network in OpenStack is fully hidden from a user and could be highly scalable, so it relies on particular configuration of a deployed environment. Benchmarks reached the platform limits and we noticed no significant differences between cryptographic algorithms nor any traffic performance differences.

To avoid real traffic and overload differences in traffic (delays on routers, unrelated traffic overhead), there was used Docker [19]. Docker's goal is to provide easily deployable images, called containers. Docker uses Linux kernel feature called *name spaces* in order to run a container. This minimizes resources to run the operating system without adding software layer above the kernel to virtualise hardware for a guest operating system. To virtualise a network, there is a software bridge between guest and host operating system. More about Docker can be found in section 3.2.

There was an effort to eliminate all factors that could introduce noise when evaluating benchmarks of VPN solutions, on the other hand, creating a new environment which differs from a real world usage introduces new paths which have to be analyzed. Figures 3.1 and 3.3 demonstrate how a connection looks like when using a VPN connection in a real world situations. We can distinguish two main VPN configurations – a remote access VPN and a site to site VPN.

Figure 3.1 demonstrates a remote access VPN. There is a VPN server, which enables connected VPN clients, which are not situated on the local LAN, to access local LAN devices. If we analyze a path of the record, we have four possible directions how can a packet flow through the system – from LAN device through VPN server to VPN client,

from VPN client to VPN server, from VPN client through VPN server to a LAN device and from VPN server to VPN client.

The first path stated, from LAN device through VPN server to VPN client, is communication from the LAN device to the remote host, which is mediated by a VPN server. The VPN server has to encrypt packed (if configured to do encryption) and encapsulate the record from the LAN and send it to appropriate output interface. There can be multiple clients connected to the server.

The second path stated is from VPN client through VPN server to a LAN device. This connection is opaque of the first path and is used for exchanging data as well.

The third and fourth path stated, from the VPN server to VPN client and vice versa is mostly used for VPN protocol itself. These messages are control messages, such as establishing or termination session, dead peer detections and many others. Another use of this path could be if a VPN server is configured to provide some services only for devices connected to the LAN. This path is a special case for the first and the second path.

If we look under the hood of server and clients, there can be distinguished two main categories of VPNs – pure kernel space and pure user space connections, depending on where the record encryption and encapsulation is done (where VPN server and VPN client operates), see 2.1. There are solutions where a VPN server can be run in the kernel and VPN clients are run in user space, but we will focus on ready to made and fully supported solutions.

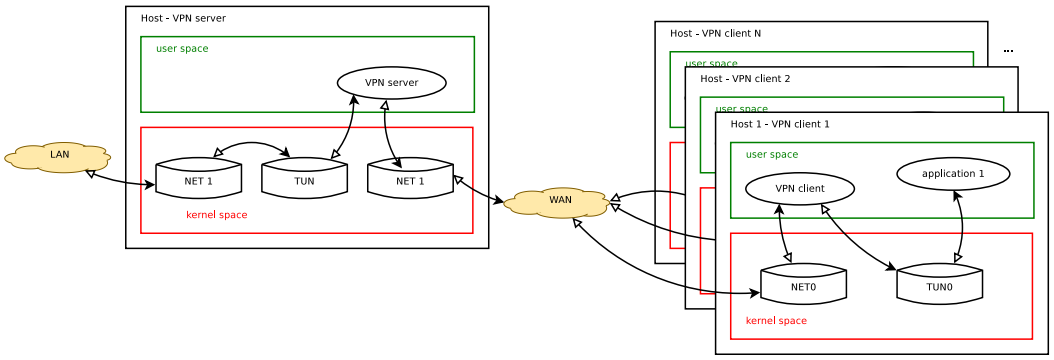


Figure 3.1: A real world record path overview for remote access VPN – user space VPNs

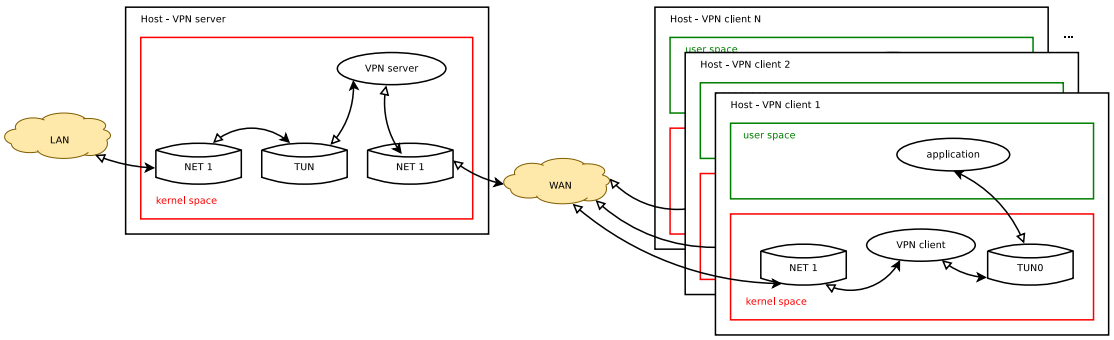


Figure 3.2: A real world record path overview for remote access VPN – IPsec

As stated before, we should eliminate parts of the system, which can introduce noise

in our benchmarks. In general, if there is a network in the system, it may add additional paths delays in router queues, delays on physical media or different routing paths which can occur. Thus it would benefit us to remove random network effects from our testing environment.

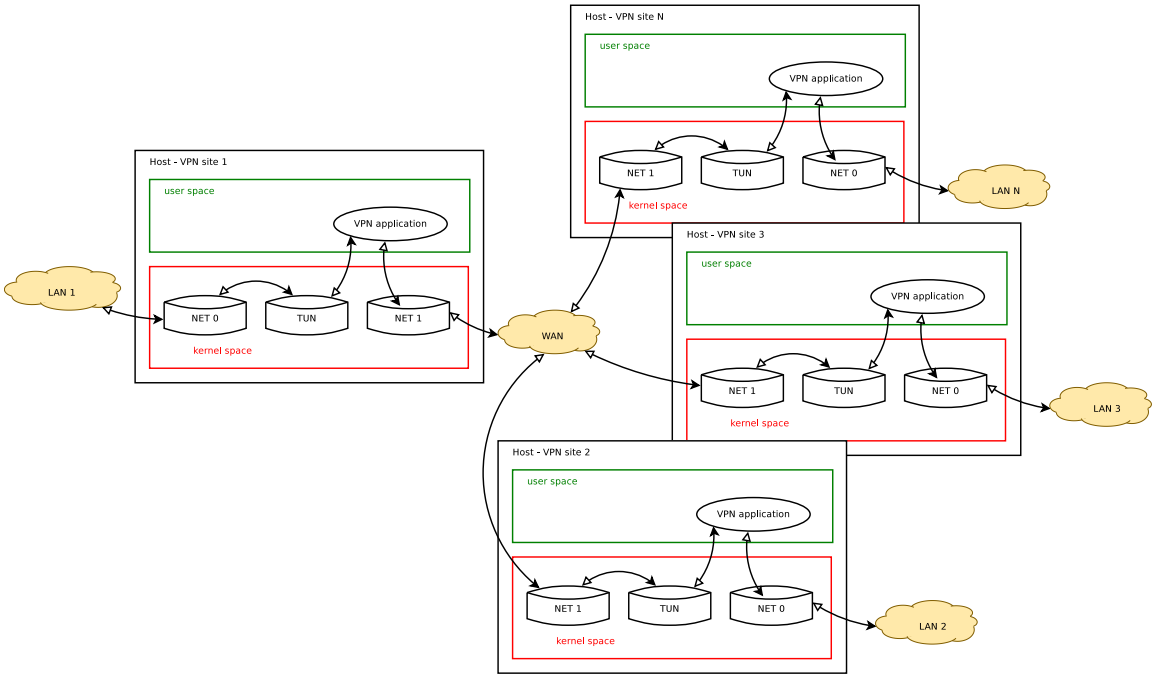


Figure 3.3: A real world record path overview for site to site VPN – user space VPNs

Another part of our system which can introduce noise in benchmarks, are different connected devices. There could be significant difference in benchmarks according to the device used as a server or as a client. If we focus only on Gnu/Linux distributions, VPN applications could be compiled with different compiler flags or there could be different application versions. Moreover, packagers of different distributions can make downstream patches for various issues (compatibility, security, downstream features, etc.). If we focus on hardware, OpenSSL, GnuTLS and even kernel's crypto API use specialized CPU instructions to make encryption faster, if available. Device configuration has a significant impact as well.

We will try to make a testing system as minimalistic as possible to eliminate impact of other parts of the system. If we omit networks and we move benchmarking on one machine we can get rid of various delays in other devices (queues on routers, routing computation, etc.) and we can get rid of impact of potentially other traffic to our system. Thus we can introduce solution on one physical device. Ideally we can make benchmarks on Gnu/Linux distribution installation, but this is impossible to set up for kernel based VPN solutions, since there is only one network device. To make testing environment the same for kernel based VPN solutions and as for user space based VPN solutions, benchmarks were done using Docker container on one system.

Versions of applications used for benchmarks can be found in appendices, section D.

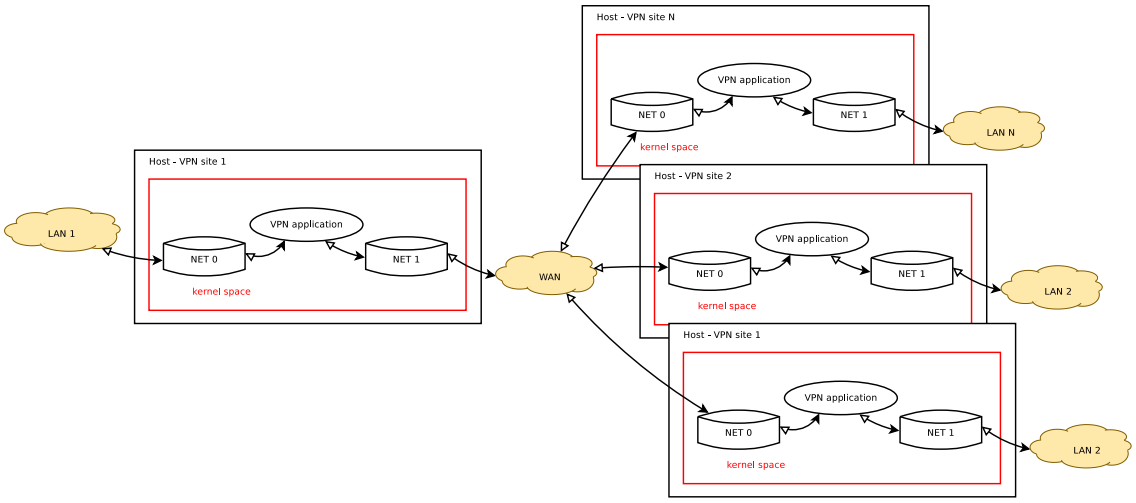


Figure 3.4: A real world record path overview for site to site VPN –IPsec

3.2 Docker

Testing environment differs from real world usage of a VPN solution, so impact of using Docker has to be deeply analyzed. IBM did a deep research of Docker and it's performance impact [34]. Results promise very small impact on actual system performance in solutions based on containers.

Docker [19] is not a virtual machine. There is no hypervisor used and the image, which runs inside Docker (Docker container) does not use its own kernel, see 3.5. Docker is based on Linux feature called Linux control groups (name spaces). Name spaces introduce separation for processes (PID), network, users, IPC, filesystems and hostnames [25]. This approach gave a birth to container technologies, such as Docker.

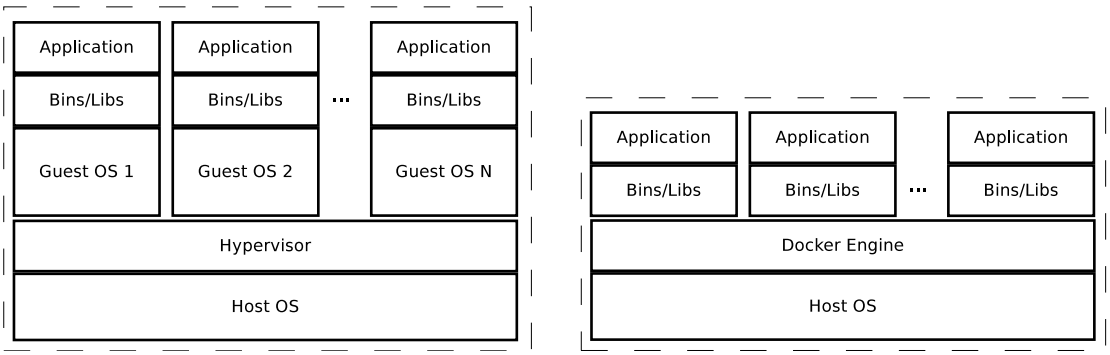


Figure 3.5: Comparison Docker with a virtual machine

Since the Docker container does not need its own kernel, there is no additional layer between host operating system and the container itself. This could be an advantage for applications that do not rely on their own, virtualized kernel and a limitation as well. The choice whether to use containers instead of virtualization depends on the requirements.

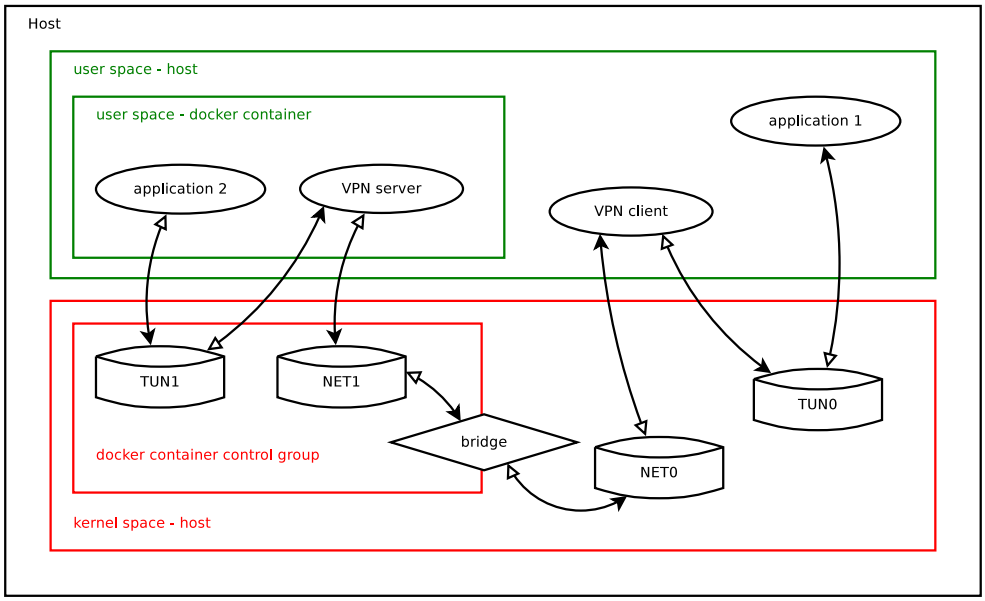


Figure 3.6: A VPN environment schema using Docker container – user space VPNs

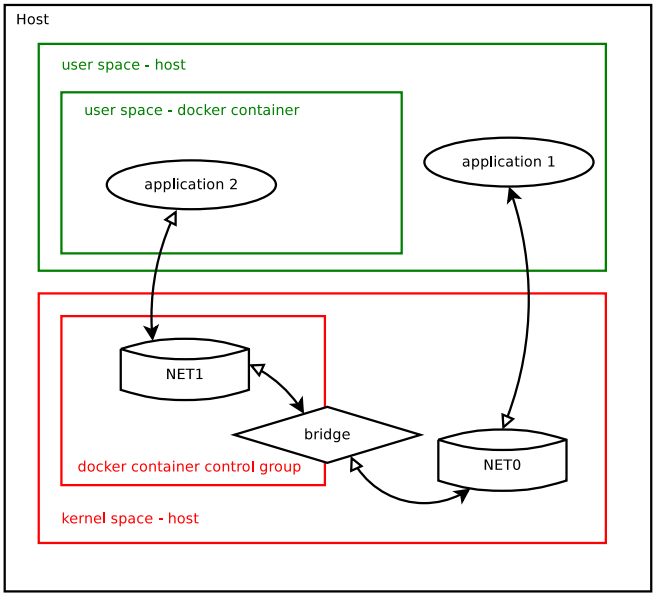


Figure 3.7: A VPN environment schema using Docker container – IPsec, in this setup the VPN server is omitted as it is not part of communication

3.3 Benchmarks of Throughput

To respect various encryption algorithms implemented and used in OpenConnect, there were done tests on all currently supported ciphers. To test transmission speed, `iperf` tool was used. Iperf uses client – server architecture. The `iperf` server was run on VPN server side and the client was transmitting data. Results of the tests can be seen in the table B.4 in appendices.

Tests were done on Intel Core i7-4600U CPU with 2.10GHz and 12GB DDR3 RAM. Concrete CPU flags of the CPU used can be seen in *cpuinfo* available on attached DVD. The CPU has accelerated AES and GCM instructions (see `pclmul` and `aes` flags in *cpuinfo*).

As stated in section 3.1, tests were done using Docker. There was made ready to use, autoconfigured Docker image described by Dockerfile. Docker image used latest to the date Fedora 23 release with all packages actualized. The deployment of Docker image is easy and fully automated process. All VPN solutions run on their standardized ports, these ports are exposed to host operating system. All information necessary to connect to a VPN are printed during setup to standard output. Dockerfile is available on attached DVD.

There should be noted that the current release of OpenVPN does not support transmission over AES GCM which was intended to compare with other VPN solutions, but there is already an open ticket in order to add support[1].

3.4 Benchmarks of Ciphers

This section presents comparison of benchmarks done in the Docker container and it's host system for GnuTLS library, OpenSSL and Linux crypto API. These tests should prove that a Docker container does not have any significant impact for our testing environment.

3.4.1 OpenSSL Benchmarks

Results of OpenSSL benchmarks for host can be found in table 3.8, for the Docker container in table 3.9. As we can see, Docker container does not have any significant impact to cryptographic algorithms in OpenSSL library.

Cipher	Run #1	Run #2	Run #3	Average
AES-128 CBC SHA1	0.61 GB/s	0.60 GB/s	0.61 GB/s	0.61 GB/s
AES-128 GCM	2.49 GB/s	2.52 GB/s	2.54 GB/s	2.51 GB/s

Figure 3.8: Benchmarks on host system, OpenSSL cipher, payload size: 16384 bytes

Cipher	Run #1	Run #2	Run #3	Average
AES-128 CBC SHA1	0.59 GB/s	0.61 GB/s	0.61 GB/s	0.60 GB/s
AES-128 GCM	2.50 GB/s	2.51 GB/s	2.50 GB/s	2.50 GB/s

Figure 3.9: Benchmarks done in Docker container, OpenSSL cipher, payload size: 16384 bytes

[1] https://community.openvpn.net/openvpn/ticket/301

3.4.2 GnuTLS Benchmarks

Cipher MAC	Run #1	Run #2	Run #3	Average
AES-128-CBC-SHA1	0.60 GB/s	0.61 GB/s	0.59 GB/s	0.60 GB/s
AES-128 GCM	2.52 GB/s	2.51 GB/s	2.52 GB/s	2.51 GB/s

Figure 3.10: Benchmarks on host system, GnuTLS cipher-MAC combinations, payload size: 16384 bytes

Cipher MAC	Run #1	Run #2	Run #3	Average
AES-128-CBC-SHA1	0.59 GB/s	0.59 GB/s	0.58 GB/s	0.59 GB/s
AES-128 GCM	2.50 GB/s	2.48 GB/s	2.52 GB/s	2.50 GB/s

Figure 3.11: Benchmarks done in a Docker container, GnuTLS cipher-MAC combinations, payload size: 16384 bytes

As we can see in tables 3.12 and 3.10, there is no significant difference when using Docker container as well. These results conform with the research paper done by IBM [34], where authors emphasize very low cost to Docker container, its resource and memory usage. Since TCP has different path for server and client, the actual impact of the this path can be seen in an IBM research paper as well [34]. Authors state no significant difference on performance when using Docker image. This confirms selection of Docker as a benchmarking and testing platform for VPN comparison.

3.4.3 Linux Kernel Crypto API Benchmarks

There is a kernel module `tcrypt` available which can be used to test Linux Crypto API. This kernel module can be inserted and based on parameters, it can do performance tests of implemented ciphers. Unfortunately it was unable to run AES-GCM (`"gcm(aes)"`) benchmarks due to a bug in this module on tested kernels (4.4.8-300.fc23.x86_64 and 4.4.8-303.x86_64) and AES-128-CBC-SHA1 performance test was not implemented.

On the other hand, it was possible to run benchmarks of RFC 4106 implementation (*The Use of GCM in IPsec Encapsulating Security Payload* [14]). The `tcrypt` kernel module can run performance tests only for payload equal or smaller than 8192 so the benchmarks cannot be directly compared to GnuTLS nor OpenSSL.

Cipher MAC	Run #1	Run #2	Run #3	Average
AES-128 GCM	1.91 GB/s	1.96 GB/s	2.00 GB/s	1.96 GB/s

Figure 3.12: Benchmarks of Linux Kernel Crypto API, payload size: 8192 bytes

Interesting founding can be seen on actual speed trend based on payload size demonstrated in figure 3.14. The performance is significantly increasing until the page size (4096B) is reached. This founding is studied in the section 5.6.1.

3.4.4 Cryptodev-linux kernel module

Cryptodev-linux [1] is a kernel module which delivers all major cryptographic algorithms implemented in Linux Crypto API in kernel space to user space. This gives an advantage

Payload size in bytes	16	64	256	512	1024	2048	4096	8192
Performance in GB/s	0.074	0.268	0.750	1.071	1.511	1.773	1.993	1.994

Figure 3.13: Performance results of RFC 4106 AES-GCM implementation based on payload size

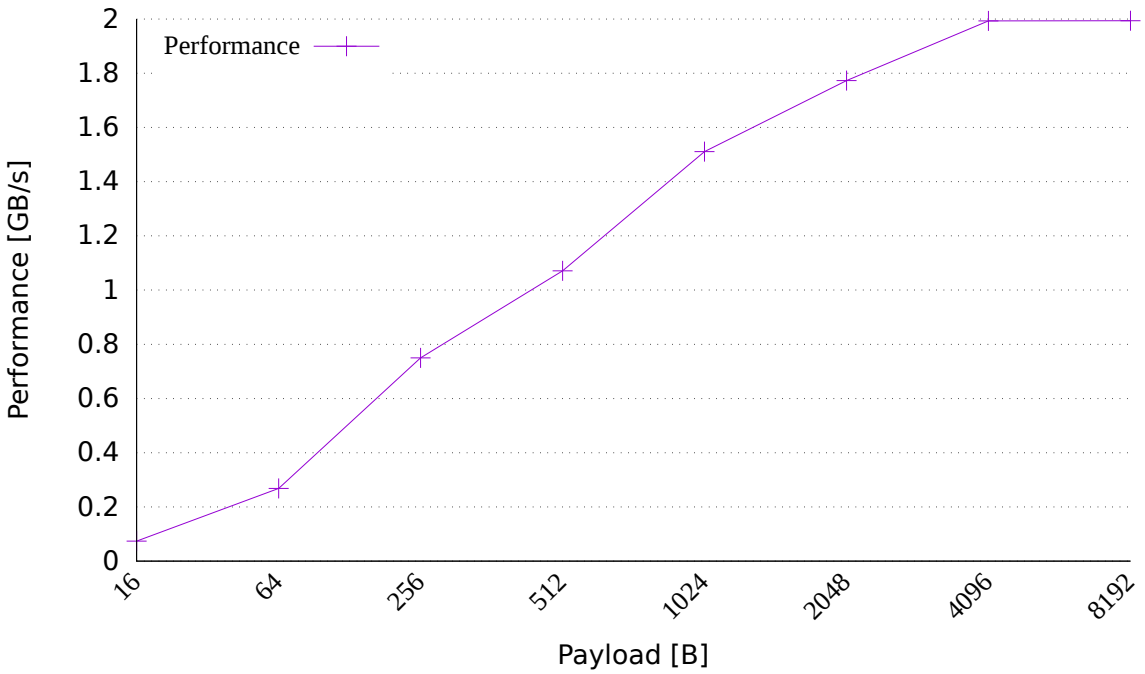


Figure 3.14: Trend of RFC 4106 AES-GCM implementation based on payload size

of using hardware accelerators transparently from user space.

A user space process has to open `/dev/crypto` and issue `ioctl(2)`'s `CIOCGSESSION` call in order to initialize Cryptodev-linux module. The kernel module is state-full. It stores all information, such as cipher type and key for user space session. After initialization, `CIOCCRYPT` is used to encrypt and decrypt messages. A session identifier and all other necessary information, such as plain text or cipher text, initiate vector and operation type (encryption or decryption) are passed via parameters.

Cryptodev-linux kernel module looks like a suitable candidate for optimizing VPN traffic. As stated in section 2.1, the main disadvantage and bottleneck for VPN traffic are context switches. As Cryprodev-linux runs in kernel space, encryption algorithms could be reused and whole traffic could be done in kernel space. Records can be encrypted or decrypted in kernel space and sent to TUN or output interface without sending them to user space just for encryption or decryption.

This kernel module is not part of upstream vanilla kernel, even there was an effort to merge Cryptodev-linux with mainline [32]. The Cryptodev-linux kernel module uses `ioctl(2)`-based API, which uses pointers and variable length data. According to upstream this introduces more error prone code, so lately there is an effort to avoid such module implementations [32].

3.5 Benchmarks of VPN Components

Measuring the throughput is good for comparison of various VPN technologies, but to optimize VPN, we have to also find bottlenecks in the system. Since `iperf` tool measures throughput, it sends and receives generated data of various size. To detect parts where the most of the time is spent in different VPN solutions, tests should be evaluated with same amount of data being sent.

Gnu time [2] was chosen to test the time spent in kernel and user space of an application. On testing environment, Gnu time uses `wait4(2)` system call to receive information about process. This system call does not influence benchmarked process. Information relating to executed process are returned by the kernel after the process termination. This information is stored in task structure (often referred as *Process Control Block (PCB)* in operating system theory) of the benchmarked process, so it does not affect the process at all.

In order to detect parts of the system where records spend most of the time, a file was generated of a size 10MB. This file consists of random bytes generated from `/dev/urandom` to simulate random data. These data were sent using Netcat [20] to handle TCP packetization and connection details. Testing environment did not change – Docker container with the same setup was used. Output file was redirected to `/dev/null` so no filesystem operations were done.

The Linux kernel provides an interface, which is used to do performance tests and debug various issues in kernel. This interface is accessible via a tool called `perf` [13]. Using this tool, one can find useful information, such as number of voluntary and involuntary context switches done during process run, and its children as well. Tracing forks is very important, since some VPN solutions, such as OpenConnect, ocserv respectively, make a separate process for every connected client due to security features.

To deeply understand what system calls are called from a process, `strace` was used. This tool wraps all system calls in order to monitor process's and kernel communication. To trace library calls, similar tool was used – `ltrace` [29]. These tools were used separately to study process communication. They were not run during benchmarks, since they make additional layer between process and library or kernel, which has significant impact on process performance. They also perform output operations, which can slower the process execution also.

In order to evaluate comparison results, tests were performed on ciphers and MAC supported by all tested VPNs.

3.5.1 OpenVPN & OpenConnect Context Switches

In order to test user space VPN applications, we evaluated tests based on communication of a process with the operating system to see how a VPN solution interacts with the operating system. The system was configured to use MTU of size 1500 bytes, which is the most common MTU used in basic traffic nowadays because of Ethernet technology. Moreover, the default value of MTU for OpenVPN and OpenConnect equals to 1500 bytes so there could be an expectation of usage of the same MTU size by a user.

Since we are testing on a simulated network, using higher MTU would cause to lower the ratio of header payload and actual data transmitted, so the protocol would look more efficient to a user. This is an important factor, which has to be kept in mind when doing benchmarks of network applications.

The results for OpenConnect server are harder to distinguish because of `fork(2)` and

VPN	Run #1	Run #2	Run #3	Average	Average in %
OpenVPN	5.15/3.36	4.68/2.97	5.08/3.03	4.970/3.211	60.750/39.250
OpenConnect	4.49/1.32	4.41/1.23	4.39/1.25	4.430/1.267	89.442/10.558

Figure 3.15: Ratio kernel space / user space time for VPN server spent when transmitting 1GB file, encryption AES 128 SHA1 used, results in seconds

VPN	Run #1	Run #2	Run #3	Average
OpenVPN	2061/2060	2299/2298	1894/1893	2084.67/2083.67
OpenConnect	2150/2149	1849/1848	1955/1954	1984.67/1983.67

Figure 3.16: Number of read/write context switches of VPN client when transmitting 1GB file, encryption AES 128 SHA1

VPN	Run #1	Run #2	Run #3	Average	Average in %
OpenVPN	5.11/3.95	4.43/3.58	4.84/3.86	4.793/3.797	55.80/44.20
OpenConnect	4.99/1.68	4.67/1.54	4.94/1.50	4.867/1.573	80.80/19.20

Figure 3.17: Ratio kernel space / user space time for VPN client spent when transmitting 1GB file, encryption AES 128 SHA1 used, results in seconds

VPN	Run #1	Run #2	Run #3	Average
OpenVPN	2061/2060	2298/2297	1894/1893	2084.33/2083.33
OpenConnect	2150/2149	1849/1848	1955/1954	1984.67/1983.67

Figure 3.18: Number of read/write context switches of VPN server when transmitting 1GB file, encryption AES 128 SHA1

simultaneous `writev(2)`, `recvfrom(2)` calls. This needs to be analyzed deeper, but since there is predefined MTU size, OpenConnect has to transmit exactly the same amount of records as ocserv receives in this particular environment.

As can be seen on tests shown in tables 3.15, 3.16 and 3.18, 3.17, OpenConnect VPN spent a lot of time in user space space. Time spent in kernel includes times spent on context switching, copying data from user space memory to kernel space memory and vice versa. Time spent in user space mostly covers time spent in GnuTLS for encryption and decryption.

Values in the table include handshake and application start as well. Number of read-/write before actual data were transmitted are constant and this value can be subtracted in order to get pure data transmission context switches. Connection establishment takes only small part compared to actual data transmission.

Context switches are done only because of records assembling/disassembling and decryption in user space libraries. If we are able to move these operations to kernel space, we can save at least partially context switches. This optimization has to be studied more deeply and it is a future work of this thesis.

3.5.2 Libreswan

Since all encryption and record assembling is done in the kernel, there are no system calls at all during communication. Thus the only indication of Libreswan speed is Libreswan throughput, which is the best compared to OpenVPN and OpenConnect (based on 3.3).

The actual encryption is specified during configuration. The initial handshake is done via IKE (Internet Key Exchange). More about IPsec protocol can be found in CISCO documentation [4].

There was an effort done to make run IPsec with VPNC client. The actual connection establishment passed, but it was not possible to exchange any data. VPNC is written to be CISCO compatible VPN client. Deeper inspection of protocol and exchange messages needs to be made in order to make VPNC work with IPsec. On the other hand, VPNC is just a standalone CISCO compatible client, which does encryption in user space. The compatibility with Libreswan is not guaranteed across all versions. Investigating this issue is out of scope of this thesis and it would not benefit this work anyhow, since VPNC is tightly bound to proprietary CISCO IPsec VPN.

3.6 Choosing VPN Solution for Optimization

Each VPN solution has its pros and cons. The main focus in this thesis is given to SSL based VPN OpenConnect. OpenConnect is, in contrast to Libreswan, an application level VPN. It is easier to configure for a user since there are available user space applications, both, client and server.

Another big advantage of OpenConnect VPN could be standardized TLS/DTLS protocol that is used. This can be seen as a benefit when using OpenConnect on firewalls used for censorship. The traffic cannot be distinguished from a regular HTTPS traffic (in contrast to Libreswan and OpenVPN), so the connection cannot be easily filtered.

OpenConnect by its design enables to do advanced accounting for users. This enables to restrict bandwidth or connection time. OpenConnect also enables to easily configure user authentication across various authorities (such as LDAP or Kerberos) [8].

There are other benefits as well, some of them are listed in the article [42], which was published by one of the authors of OpenConnect.

Chapter 4

A Faster Approach: Implementation

In this section we present implementation and initial optimization design of VPN solutions. There are presented TLS and DTLS records that carry encrypted data.

4.1 Optimization Design

As we can deduce based on info from section 3.5.1, the main bottleneck of OpenConnect implementation appear to be context switches and data copies. There are done two context switches per each record, just to encrypt the record and pass it to appropriate output interface or TUN device. This adds additional CPU time needed just for doing context switches. By designing a kernel module which would be configurable from user space and would do the encryption or decryption in the kernel space we might save additional time and possibly increase throughput.

By moving the whole TLS/DTLS protocol to the kernel, we would probably increase speed. On the other hand, the implementation of the kernel would enlarge and would be really hard to maintain. If we consider, that the handshake is done only from time to time, we don't need to bother kernel space with actual handshake. Moreover, errors which can occur during handshake are easier to handle because of user space client and server implementations.

The actual encrypted communication based on symmetric cryptography is much more interesting to be optimized since it typically accounts for the majority of the transmission. The symmetric encryption/decryption is used on every record which carries actual data and exactly this part we want to optimize. The key concept of optimizing OpenConnect is to move TLS/DTLS record assembling, disassembling and symmetric encryption, decryption of records into kernel space. If anything goes wrong, we should notify user space about errors via standardized return codes of system calls.

There were made several approaches during writing this thesis. All of them are discussed in the following sections.

4.2 TLS and DTLS protocol

The TLS protocol requires an underlying protocol which carries TLS records to be reliable [15]. That means that this protocol has to deliver records in order, duplicates have

to be dropped and underlying protocol has to deal with out of order delivery as well. An example of such protocol is TCP, which is also the primary protocol for TLS. Figure 4.2 describes record structure for AEAD ciphers, such as AES GCM, with 16 bytes tag. Other ciphers have different structure, which is described in RFC 5246 [15].

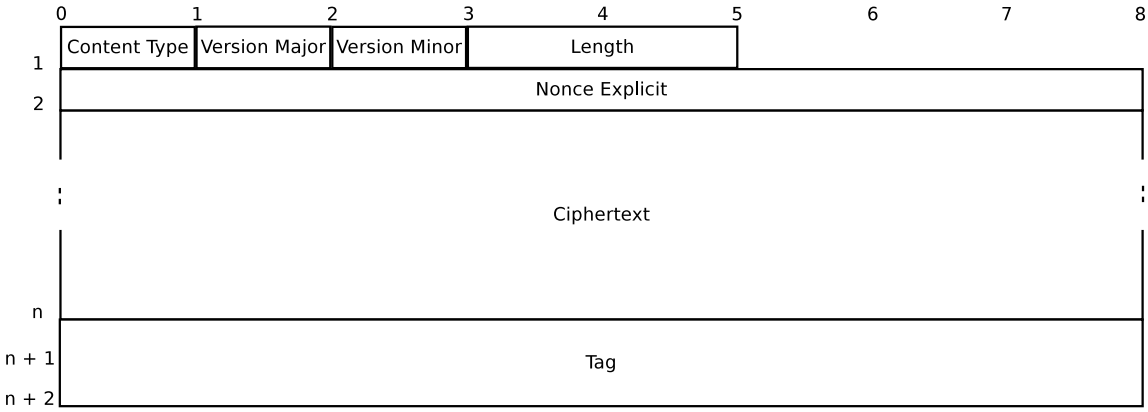


Figure 4.1: TLS 1.2 record for AEAD ciphers with 16 bytes long tag

A record in DTLS of version 1.2 has different structure than TLS of version 1.2. This structure is demonstrated on figure 4.2. DTLS is designed to operate on unreliable protocols [17], such as UDP, where out of order delivery, duplicates or congestion avoidance is not implemented. That means that DTLS has to add some mechanism that would uniquely distinguish records transferred within a session. This requires DTLS header to add epoch and sequence number to the record header. Epoch is incremented every time a re-keying is done within a session, a sequence number uniquely specifies a record in epoch.

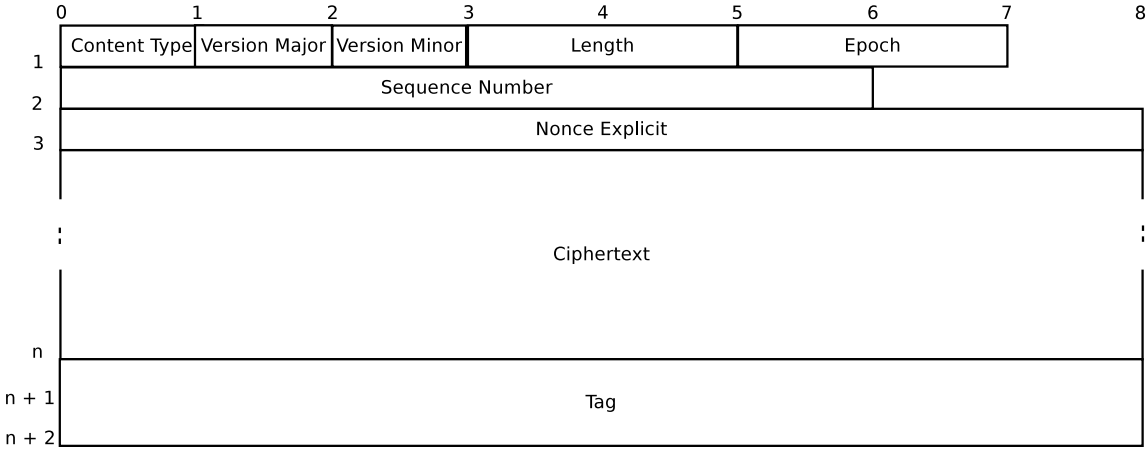


Figure 4.2: DTLS 1.2 record for AEAD ciphers with 16 bytes long tag

DTLS protocol has to guarantee that duplicate records get discarded and implementations of DTLS protocol use a sliding window in order to operate on a constant size window that would discard duplicates. However, out of order delivery is not handled by DTLS protocol [17] for packets within the window.

All examples cover data records (content type is of value `0x17`). Other control messages specified by TLS and DTLS protocol are out of scope of this text. All details can be found in

appropriate RFCs, namely RFC 5246 [15] for TLS 1.2 protocol definition and RFC 6374 [17] for DTLS 1.2 protocol definition.

Both, TLS and DTLS, limit the maximum number of bytes that can be carried within the record payload to 2^{14}. This limit is not easily reachable on real networks nowadays.

From OpenConnect point of view, it is more interesting to focus on DTLS. OpenConnect uses DTLS for data transmission, but fallbacks to TLS if DTLS connection is not possible as described in section 2.2.

4.3 Reusing the Cryptodev-linux Implementation

Our original plan for implementation intended to use Cryptodev-linux kernel module to optimize DTLS. Even the implementation of Cryptodev-linux is not part of upstream source code as stated in 3.4.4, the very first optimizations of OpenConnect protocol started in this module. The design of Cryptodev-linux kernel module was pretty straightforward. Since we want to keep benefits of user and kernel space separation, we had to choose carefully which parts are suitable for kernel space.

After the handshake was done using GnuTLS, the control was then moved to a custom DTLS handling part in Cryptodev-linux. Since DTLS is a standardized protocol, it is possible to use any user space library to do handshake (e.g. even OpenSSL or GnuTLS). The library has to support retrieving key material and other data that are needed for connection.

The implementation that would support Cryptodev-linux was partially finished. It does not handle re-keying and record assembling/disassembling is done in user space, Cryptodev-linux is used for decryption and encryption. There were not any benchmarks made for the implementation. This implementation is slower because of Linux Crypto API `"aes(gcm)"` implementation (see 4.6.8) and two additional context switches that were needed in order to pass key material to Cryptodev-linux 4.6.8 (which were intended to be removed in the future development).

The main reason of dropping Cryptodev-linux support was Facebook's patchset introducing kernel TLS socket [49, 33]. The actual work and implementation was focused on newly introduced TLS implementation by Facebook.

4.4 TLS Kernel Socket by Facebook

As stated in the previous section 4.3, the implementation based on Cryptodev-linux was dropped and the main focus was given to kernel sockets.

On 23th November 2015, Facebook proposed a patchset which implemented TLS encryption in the kernel [33, 49]. The main idea was pretty much the same as the original idea of optimizing OpenConnect – do the handshake in user space and use kernel only for encryption or decryption of actual data. Facebook's patchset was introduced by Dave Watson and it consists of two parts.

The first patch implements RFC 5288 [16] (*AES GCM Cipher Suites for TLS*) on top of an already implemented RFC 4106 [14] (*The Use of GCM in IPsec Encapsulating Security Payload*). These RFCs discusses about AES GCM implementation. The key difference is, that RFC 4106 uses 16 bytes of associated authentication data, whereas RFC 5288 uses 13 bytes which are zero padded to 16 bytes[14, 16].

The second patch implemented actual socket handling. The implementation was based on kernel's `AF_ALG` which exposes kernel Crypto API similar to Cryptodev-linux does. The implementation of `AF_ALG` is part of upstream even so it appears to be slower than Cryptodev-linux according to [1, 48].

The difference in user space API between `AF_ALG` and Cryptodev-linux is, that `AF_ALG` introduces a new protocol family. Based on `AF_ALG` protocol family, the user space can use basic socket operations such as `accept(2)`, `bind(2)`, `sendmsg(2)`, `recvmsg(2)` in order to handle encryption.

The implementation of `AF_ALG` adds a possibility to extend `AG_ALG` with a custom type – custom module implementation wrapped by `AF_ALG`. The implementation of such module has to provide basic operations such as setting keys, setting authentication data size, `sendmsg(2)` or `recvmsg(2)` handling etc.

The Facebook's patchset implemented TLS framing, sending and receiving messages. Moreover, the proposed patch [49] introduced the implementation that can be used with `sendfile(2)` system call.

The original author of Facebook's patchset, Dave Watson, claimed that speed was increased by 2 - 7% when using `sendfile(2)` system call with 128KB buffer [33, 49]. This speed increase was not reproduced on tested hardware. Actually speed decreased by 10% (tested with tool "`ktls`" referenced by Dave Watson on Linux Crypto mailing list on Lenovo ThinkPad T440s with CPU that supports AES-NI instructions). There were proposed three patches to Dave in order to make `ktls` application work on our testing environment.

The TLS socket introduced by Facebook looked very interesting since it shared the idea with our design. After the patchset arrived to mailing list, we studied the patchset and contacted Dave Watson via e-mail. We notified him about our purpose of optimizing VPN. Since the main focus was to optimize DTLS, Dave claimed that Facebook had no use case for optimizing DTLS, but was opened for cooperation and help.

4.5 Reusing `AF_ALG` Socket

The implementation of `AF_ALG` TLS socket was analyzed and studied with respect to its reuse or expansion to support DTLS. Since TLS is using connection oriented communication, such as TCP, the implementation of `AF_ALG` TLS socket is relaying to the supplying TCP socket, which is wrapped by `AF_ALG`.

We have done initial work to support UDP socket and to use DTLS protocol. The result was not easy to handle from user space. There had to be passed some key material via control messages that can be supplied with `sendmsg(2)` system call, some were passed via `setsockopt(2)`. Which could be supplied how is strictly restricted to `AF_ALG` implementation. This makes TLS/DTLS socket handling from user space very difficult and error prone.

Even the handling from user space was difficult, there were necessary three socket instances – raw TCP/UDP socket, `AF_ALG` socket that is used for configuration and the TLS/DTLS socket implemented as a kernel module wrapped by `AF_ALG`, which was used for configuration and sending/receiving as well.

4.6 Introducing Custom `AF_KTLS`

After implementation in Cryptodev-linux and reused `AF_ALG` interface, we decided to introduce a standalone kernel TLS type socket, which would use standard socket interface, would encapsulate the whole process of record assembling, disassembling, encryption, decryption and will be easy to use from user space. The main reasons why `AF_ALG`-based implementation by Facebook was not reused are:

- There are necessary 3 socket instances for each TLS connection – `AF_ALG`, raw TCP socket and socket for the actual TLS transmission.
- The API can be confusing, since some data are passed via `setsockopt(2)`, some are passed via ancillary data (control information) in `sendmsg(2)` call.
- Tight dependency on the `AF_ALG` kernel module.
- Overall, the interface is hard to manage and error prone for user space.

This concept led to introduction of a kernel type TLS and DTLS socket, so called `AF_KTLS`. This socket is implemented as a standalone kernel module and can be used as a built-in module in kernel or it can be inserted on demand just like any other kernel module based on modular Linux kernel architecture.

4.6.1 Zero copy `AF_KTLS`

In order to introduce a new kernel module, there have to be use cases where the module is useful. Optimizing a VPN is one of them.

The main purpose of optimizing was to save context switches, which is possible due to available Linux kernel module system calls such as `sendfile(2)`. Another advantage is to save unnecessary copies of data from kernel space to user space and vice versa as much as possible. Linux kernel by its design can operate internally on pages. Pages are described by `struct page` kernel structure, which carries all the information necessary to be able to correctly handle page usage within kernel subsystems.

Based on kernel modules type, a kernel module can expose its allocated pages to other parts of the kernel, which can operate on them with respect to page usage.

To understand how pages are manipulated within a kernel modules, consider following code snippet (includes and error checks removed intentionally):

Listing 4.1: Explanation of `splice(2)` syscall

```
1  int main(void) {
2    int f1, f2
3    int pipefd[2];
4
5    f1   = open("/etc/passwd", O_RDONLY)
6    f2   = open("output.txt", O_WRONLY|O_CREAT|O_TRUNC);
7
8    pipe(pipefd);
9
10   splice(f1, NULL, pipefd[1], NULL, 10, 0);
11   splice(pipefd[0], NULL, f2, NULL, 10, 0);
12
13   main_end:
14   close(filefd);
15   close(pipefd[0]);
16   close(pipefd[1]);
17
18   return 0;
19 }
```

The code was simplified to demonstrate principles. There are no checks whether calls succeeded. For example if there are no sufficient permissions to create the testing file, program will not work.

The idea of the code snippet 4.1 is pretty straightforward – to read 10 bytes from the input file `/etc/passwd` and place it to the pipe with the first `splice(2)` call. The second `splice(2)` call causes reading 10 bytes from the pipe and writing in to the output file `output.txt`. In this particular example, there will be no copies done in the best case. There will be allocated a page cache for the input file (if not allocated already) and there will be passed appropriate page from the allocated page cache to the pipe which is acting here as an explicit buffer. The pipe will mark page as used so this page will not be removed from the page cache once kernel tries to drop caches. The second `splice(2)` call will cause reading 10 bytes from the pipe and writing it to the output file. Again, there should not be done any copy because appropriate `kernel_sendpage` (see 4.6.5) is called in order to write 10 bytes from the pipe to the output file. This causes that page will be removed from pipe (marked as not used in pipe) and moved to appropriate page cache of the output file. This page will be flushed to disk once kernel issues sync (see `sync(2)`) because of dirty flag in the page cache of the output file. The typical size of a page is 4096 bytes. Even so that 10 bytes can be placed in 10 pages in the worst case, the kernel will try to align kernel buffers (even page cache) to size of a page because of efficiency.

The example 4.1 discusses about zero copy page moving inside kernel. Even so this example demonstrates principles, the underlying implementation is hardly dependent on file system being used and its implementation.

4.6.2 Asynchronous Record Decryption

CPUs nowadays are designed to be multi-core and multi-threaded. This benefits applications that have parallel computation. Even if there is no parallel execution, unused CPU can be occupied by kernel which can operate on a single core or can be spread on all available cores.

From TLS and DTLS point of view, the unused computational power can be used, if

available, to decrypt records asynchronously and prepare them for user space. If user space asks for records, the kernel can offer pre – decryped record with speed of a copy.

4.6.3 DTLS Sliding Window Implementation

DTLS protocol is designed to operate over both reliable and unreliable type of protocols. DTLS protocol does not compensate for lost or re-ordered data traffic, but requires replay detection [17]. According to RFC 6347 which discusses DTLS 1.2 [17], DTLS maintains a bitmap window which is responsible for replay detection.

RFC 6347 does not specifies the size of bitmap window. Implemented `AF_KTLS` kernel module uses bitmap window implemented as sliding window on 64 bits same as GnuTLS does.

4.6.4 User Space `AF_KTLS` socket handling

All protocol families implemented in the kernel have to provide `struct net_proto_family` which describes protocol family constant used for referring protocol family and `create` method which is called when user space issues `socket(2)` system call in order to instantiate a socket of the given family.

System call `socket(2)` requires to define domain (which is `AF_KTLS` for implemented module), type and protocol. Currently type distinguish socket type that will be bound to the socket. The semantics is same as for well – known `AF_INET` socket – `SOCK_DGRAM` if UDP socket will be bound or `SOCK_STREAM` if TCP socket will be bound. The last parameter can specify protocol, currently this parameter can specify OpenConnect protocol if socket will be used in OpenConnect VPN in order to forward only data records on OpenConnect protocol layer.

The Linux kernel has predefined set of operations for sockets. These operations are defined by `struct proto_ops` structure which can be seen in `include/linux/net.h`.

Implemented `AF_KTLS` kernel module implements subset of these operations. This subset was chosen according to `AF_KTLS` and operation semantics; moreover there is no need to implement all of them since some of them are not useful for `AF_KTLS`. Not implemented operations are held by kernel, which fallbacks to default behaviour (propagating appropriate errors such as `ENOTSUPP`).

After socket creation, user space should bind a socket using `bind(2)`. There is introduced a custom `struct sockaddr` called `struct sockaddr_ktls` with following fields:

Listing 4.2: Custom structure definition for `bind(2)`

```
1 struct scokaddr_ktls {
2   __u16 sa_cipher;
3   __u16 sa_socket;
4   __u16 sa_version;
5 };
```

The design is intended to be type safe, so there should be a constant which would specify cipher type for `sa_cipher` field. Even so there is implemented only AES GCM 128 support, there could be easily added support for other ciphers as well in the future. This constant is mapped to appropriate string used in Crypto API that describes cipher in kernel's Crypto API. The mapping is done inside kernel, so there is type safety guaranteed and all implementation details, such as which cipher implementation and its memory organization will be used, is held by kernel transparently to user space.

Field `sa_socket` is referring a socket on which handshake was done and the data transmission will be done. Bound socket should respect `type` parameter supplied when `AF_KTLS` socket was created using `socket(2)` – UDP socket for `SOCK_DGRAM` (in this case DTLS will be used) and TCP socket for `SOCK_STREAM` (TLS will be used).

The last field `sa_version` specifies TLS or DTLS version. Currently there is support for 1.2 TLS/DTLS. Introducing this field makes the implementation of `AF_KTLS` ready to be extended with possible new future standards. Older versions are not considered to be implemented since they should be substituted with the newer ones.

Once there was done `bind(2)` on `AF_KTLS` socket, user space should supply all communication configuration based on handshake (IV, key, salt for receiving and sending). This is done by using `setsockopt(2)` call. The configuration can be received back to user space by calling `getsockopt(2)` with appropriate `optname`.

Once the communication is over, user space should call `close(2)` to correctly deallocate socket and free all resources used by socket in the kernel.

4.6.5 Supported System Calls for Data Transfers

System calls described in subsection 4.6.4 are used to configure `AF_KTLS` for transmission. To actually transfer data, there are supported copy-less system calls such as `splice(2)` or `sendfile(2)`. For `splice(2)` instantiated and correctly prepared `AF_KTLS` can be used as source and destination in copy-less transfers.

To make possible to use `AF_KTLS` socket as a source, there has to be implemented appropriate `splice_read` operation in `struct proto_ops` kernel structure. The implementation introduces `tls_splice_read`, the prototype of such operation is following:

Listing 4.3: Prototype of `splice_read` function

```
1 ssize_t tls_splice_read(struct socket *sock,
2                          loff_t *ppos,
3                          struct pipe_indoe_info *pipe,
4                          size_t size,
5                          unsigned int flags);
```

The very first parameter `sock` is a pointer to instantiated socket structure. Since `splice(2)` system call requires one of the arguments to be a pipe, `pipe` refers to pipe where pages should be given. Argument `size` conforms to data size that was requested to be read and `flags` are used as additional options (see `splice(2)` documentation for list of all available flags).

To support copy-less writing to `AF_KTLS` socket, there was introduced `tls_sendpage` operation, which conforms to `sendpage` operation of `struct proto_ops` structure. The prototype of the implemented function is following:

Listing 4.4: Prototype of `sendpage` function

```
1 ssize_t tls_sendpage(struct socket *sock,
2                       struct page *page,
3                       int offset,
4                       size_t size,
5                       unsigned int flags);
```

The first parameter is, again, a pointer to instantiated socket structure. The second parameter is a list of pages that should be written to `AF_KTLS` socket. Parameter `offset` describes offset (not used in `AF_KTLS`). Besides available user space flags that can be supplied

in `splice(2)` system call, additional flags specified by kernel, such as `MSG_MORE` if there are going to be more pages sent or `MSG_OOB` if out of bound data are transferred (not supported by `AF_KTLS`).

Moreover the implementation supports system calls commonly used for example on Berkeley sockets – `sendmsg(2)` and `recvmsg(2)`. These system calls are the generic ones and are directly mapped to appropriate `kernel_sendmsg()` and `kernel_recvmsg()` operations. These system calls are using `struct msghdr` which can pass additional ancillary data (see `sendmsg(2)` or `recvmsg(2)` for more info). This structure is mapped by kernel to kernel's structure, which adds additional in-kernel related attributes. Mostly user space uses "lightweight" system calls such as `send(2)` and `recv(2)` to simply pass directly buffer with data (not scatter/gather array as in `sendmsg(2)` or `recvmsg(2)`). These system calls are mapped to more generic `kernel_sendmsg()` and `kernel_recvmsg()` implementations by kernel before requested operation is called (the same applies to `sendto(2)` and `recvfrom(2)`). There can be even called `write(2)` and `read(2)` system just like on a file descriptor. These system calls get mapped to `kernel_sendmsg()`, `kernel_recvmsg()` respectively as well.

4.6.6 Synchronized User Space Operations on `AF_KTLS` and Bound Socket

A very important note for user space is how both – bound and `AF_KTLS` sockets should be handled. The `AF_KTLS` socket is basically a kernel space wrapper above the bound socket which does TLS/DTLS record assembling or disassembling, MTU handling in case of `sendpage` operation and encryption or decryption. Important is to notice that `AF_KTLS` does caching as well, based on asynchronous record decryption explained in section 4.6.2. User space has to follow the implementation specific requirements.

It is not possible to use both, `AF_KTLS` and bound socket from user space (without explicit synchronization). Once `AF_KTLS` is used, it is expected that user space uses this socket for data transfers and uses bound socket only if an error occurs or there are received control messages that cannot be processed by `AF_KTLS` socket. If user space uses both `AF_KTLS` and bound socket at the same time, received data and sequence numbers will become inconsistent. If the underlying protocol is TCP, there could be requested a re-handshake since state that has to be maintained due to secured connection becomes inconsistent in user space library (OpenSSL, GnuTLS or any other implementing TLS/DTLS) and in `AF_KTLS`. For UDP socket bound (DTLS in use) there can be received same data multiple times due to sliding window and its state.

If user space is serving any control messages, new key material has to be passed to `AF_KTLS` module if used in order to use renewed session. The cache used within `AF_KTLS` module gets correctly flushed if there is done any operation that requests TLS/DTLS state change to guarantee TLS/DTLS session consistency.

It is *recommended* for user space to respect these restrictions otherwise applications could be prone to reply attacks.

4.6.7 Supported and Unsupported Flags for Operations

Currently there are supported two bound socket types – TCP and UDP sockets. The implementation design allows to extend support with other socket types as well (such as Unix sockets, or socket type for SCTP protocol).

Each socket can have different requests. One of the requirements that `AF_KTLS` has, is to have `MSG_PEEK` support on a bound socket. This flag tells socket to read data but not

to remove them from the receiving queue, which will be removed only if the record could be processed. If an error occurs during record processing, there has to be left record in receiving queue for user space to handle it.

Handling other flags defined on sockets (such as `MSG_PEEK`) is worth to consider for `AF_KTLS` and they require a discussion at the kernel mailing list.

There has to be also stated, that even `AF_KTLS` uses kernel's generic `kernel_recvmsg()` and `kernel_senmsg()` interface there are differences in socket handling. For example removing a record from receiving queue differs for UDP and TCP sockets. This removal shouldn't copy data from receiving queue (since they were already copied once peek was done). On UDP socket this can be performed by reading zero bytes from the bound socket. On the other hand, TCP requires to fully specify length (since TCP is streaming protocol, there is no explicit end of the record like on a UDP datagrams) and `MSG_TRUNC` flag.

4.6.8 Linux Crypto API and `AF_KTLS`

Thy Linux Crypto API was used in order to do encryption and decryption for TLS and DTLS records. AES cipher with key size 128 bits was chosen for the traffic encryption in Galois Counter Mode (AES GCM) because of its usage.

The Linux Crypto API provides a suite of implemented ciphers available within the kernel. These ciphers can be used in various modes and can have architecture dependent implementation. There is a way how to describe cipher that should be used, and mode which should be applied. The chosen cipher AES GCM has a string identifier `"gcm(aes)"`. By supplying this parameter besides all of the necessary configuration, Linux Crypto API does AES GCM encryption or decryption.

The very first implementation of `AF_KTLS` was using `"gcm(aes)"` implementation of AES GCM. Unfortunately the decryption and encryption was too slow, much slower than user space AES GCM implementation in GnuTLS on tested hardware. This issue was analyzed. When using `"gcm(aes)"`, Linux crypto API uses AES implementation, which does not use optimized AES-NI instructions. On x86_64 architecture, Linux crypto API chooses AES implementation which is implemented with basic x86_64 instructions. Moreover, the GCM part is computed separately independently on desired cipher type (GnuTLS can be forced to override CPU flags that clarify whether AES-NI instructions are available during run time by exporting `GNUTLS_CPUID_OVERRIDE=1` environment variable, this is applicable for OpenSSL as well by exporting `OPENSSL_ia32cap=~0x200000200000000`).

Nevertheless, there are available implementations which support AES-NI instructions. There is available `"rfc4106(gcm(aes))"` implementaton that was intended to be used with IPsec Encapsulating Security Payload (ESP) [14]. This implementation was done by Intel and it uses AES-NI optimized instructions, moreover GCM part is not done separately, so `"rfc4106(gcm(aes))"` on x86_64 is a standalone cryptographic driver (whereas `"gcm(aes)"` uses `"gcm"` driver for GCM computation and `"aes"` driver for AES encryption/decryption). The implementation of `"rfc4106(gcm(aes))"` consists of multiple implementations of AES GCM depending on targeted CPU. If targeted CPU supports AVX of version 2 (which contains AES-NI instructions), there is chosen the most powerful optimization available. There are also available optimized version for AVX of version 1, SSE and basic (core) x86_64 instructions. Since the implementation is hardly dependent on CPU used, one can find implementation under `arch/x86/crypto/aesni-intel-glue.c` in Linux git repo tree.

The reason why `"rfc4106(gcm(aes))"` implementation covers implementation using

x86_64 that could be reused from `"gcm(aes)"` instructions is twofold. The first one is, that Linux crypto API uniquely identifies driver by its name. So if there would be no AVX extension nor SSE available, `"rfc4106(gcm(aes))"` would not have any implementation available.

The second reason is, that `"rfc4106(gcm(aes))"` requires different memory organization in passed scatter lists for encryption and decryption (see RFC 4106 for more info how associated authentication data are organized [14]). This is also reason, why it is harder to switch to different crypto driver in `AF_KTLS`, since memory organization for different crypto drivers can differ. That's one of the reasons why should `AF_KTLS` kernel module encapsulate crypto API and its configuration (in comparison the first `AF_ALG` implementation where crypto driver was specified from user space).

Facebook proposed one patch in sent patchset [49], that added RFC 5288 [16] support to Linux Crypto driver. Instead of introducing own AES GCM implementation, Facebook reused the one from RFC 4106 [14]. The difference is, that RFC 5288 requires associated data to be padded to 21 bytes and "key" is of length 20 bytes, where 16 bytes are actual key and 4 bytes are salt (nonce implicit). Explanation of AES GCM and different RFCs is out of scope of this text. For more details refer to appropriate RFC [14, 16].

4.7 Optimization Design Based on Specific Scenario

Following sections discuss designed optimizations based on available Linux kernel system calls. These optimizations reflect particular scenarios which are possible to optimize based on the current `AF_KTLS` kernel module implementation.

4.7.1 Optimization of File Transfer

With `sendfile(2)` optimization, we want to optimize copy content from a file to user space just for encryption and then copying it to socket sending queue:

Listing 4.5: Use case for `sendfile(2)` optimization

```
1 read(fd, buf1, size)
2 encrypt(buf1, buf2, size)
3 send(sd, buf2, size)
```

Calling `read(2)` causes one context switch. During this context switch, data from file are copied to `buf1`. Call that is issued in the kernel is tightly bound to file type `fd` and it depends on Linux Virtual File System layer. It can be a local file on a local hard drive or even a file from a network file system. Ideally this file can be cached in kernel's page cache memory but the copy has to be issued every time `read(2)` is called in order to fill user's buffer `buf1`.

After the content is copied to `buf1`, context can be switched back to user space where is issued encryption. Implemented cipher AES GCM is more effective when destination buffer differs from source buffer. After the encryption, encrypted data can be sent to socket descriptor `sd`.

If we summarize this approach, there can be seen 2 context switches – one for `read(2)`, one for `send(2)`. We do not allocate any memory, however we do 2 copies – one when `read(2)` is called to copy the content and one in `send(2)` in order to queue encrypted content of `buf2` in socket sending queue.

If we look at originally proposed optimized scenario with `sendfile(2)`, we get one only following system call:

Listing 4.6: Optimized use case of sending a file

```
1 sendfile(fd, sd, /*offset*/NULL, size)
```

We do not copy anything to user space and all work is done in kernel space. It costs us one context switch, but if we look more closely we have to read content of the file and pass it to the socket descriptor `sd`.

The current implementation is using already implemented kernel routines on VFS layer that does copy-less reading (it passes pages from page cache), if pages are not cached. After it, pages are passed to `sd` via `kernel_sendpage()` adapter, which issues `tls_sendpage()` function of `AF_KTLS` socket. This function accepts page (among others parameters), does encryption and record assembling based on MTU and sends encrypted content to the socket.

AES GCM is more effective when the encryption is not done in situm. This involves allocation of memory that would be passed to the socket. Current implementation preallocates memory on socket creation and reuses this memory each time a `tls_sendpage()` is issued. After encryption there is called `kernel_sendmsg()` on appropriate socket. Function `kernel_sendmsg()` is an adapter for appropriate `sendmsg(2)` implementations based on socket type. Implementations of TCP and UDP sockets does copy in `sendmsg(2)` routines. The memory used in `sendmsg(2)` for content cannot be queued in the socket queue, because it can be reused after `sendmsg(2)` call.

4.7.2 Optimization of OpenConnect VPN

The scenario we want to optimize is listed in 4.7. This scenario is applicable for OpenVPN as well.

Listing 4.7: OpenConnect use case to be optimized

```
1 recv(tun, buf1, size)
2 decrypt(buf1, buf2, size)
3 send(sd, buf2, size)
```

There are used two syscalls - `recv(2)` and `send(2)`, so we have two context switches involved. By calling `recv(2)`, kernel issues appropriate `recvmsg(2)` implementation of TUN device socket in `kernel_recvmsg()` which does one copy. By calling `send(2)` there is done one more copy to `sd` socket sending queue. To sum it up, we have two copies, two context switches.

The optimized version can use `splice(2)`, with `AF_KTLS` socket `sd` like listed in 4.8. This optimization can be seen on figure 4.3, which conforms to optimized version of transmission demonstrated in figure 2.2.

Listing 4.8: Optimized OpenConnect use case (simplified pseudo – code)

```
1 splice(tun, pipe)
2 splice(pipe, sd)
```

There cannot be used `sendfile(2)` (see section 5.1 for more info). With this approach, context switches will not be saved. There are still two context switches involved, but there is no copy done. Section 5.1 analyzes `splice(2)` system call and its implementation details.

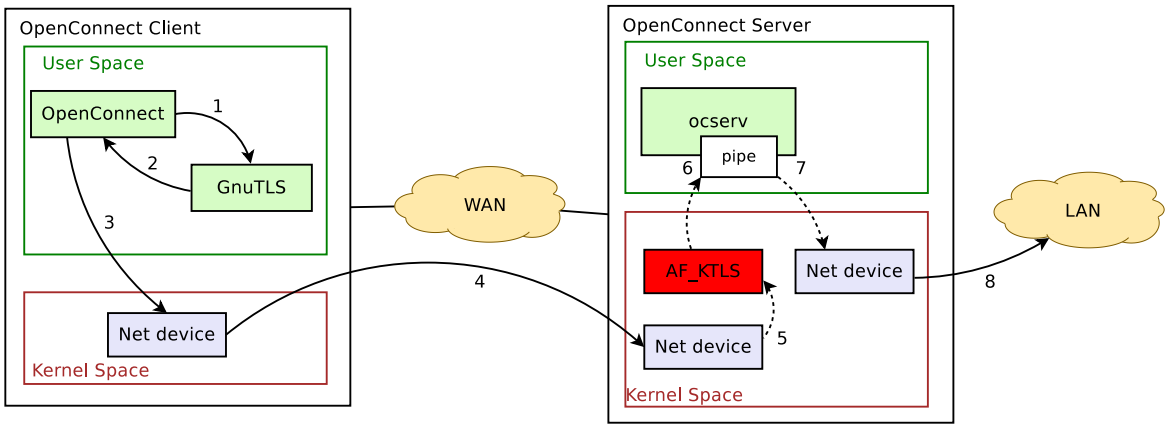


Figure 4.3: A diagram of transmission in the implemented `AF_KTLS`, dotted arrows pinpoint copy-less operations

HAProxy

All of the optimizations designed in section 4.7.2 are also applicable for HAProxy. HAProxy stands for High Availability Proxy [3]. It is a reliable and very fast implementation of an open source load balancer. According to project's homepage it is an open source standard for load balancing of TCP and HTTP protocols. It is used by big clouds solutions and the optimization designed in OpenConnect VPN could be applicable in HAProxy.

On figure 4.4, there can be seen a basic principle of HAProxy usage. It does not use TUN device as OpenConnect does. Instead, HAProxy reads records from one interface and sends them to another. HAProxy can be configured to use TLS in order to serve encrypted traffic (there are specific versions that support SSL). If HAProxy is configured to use TLS, the traffic is decrypted before it is sent to an output interface. In that case, the scenario of sending records is exactly the same as for OpenConnect (but TUN device is substituted with a socket). This makes HAProxy another candidate for optimization with `AF_KTLS` kernel module.

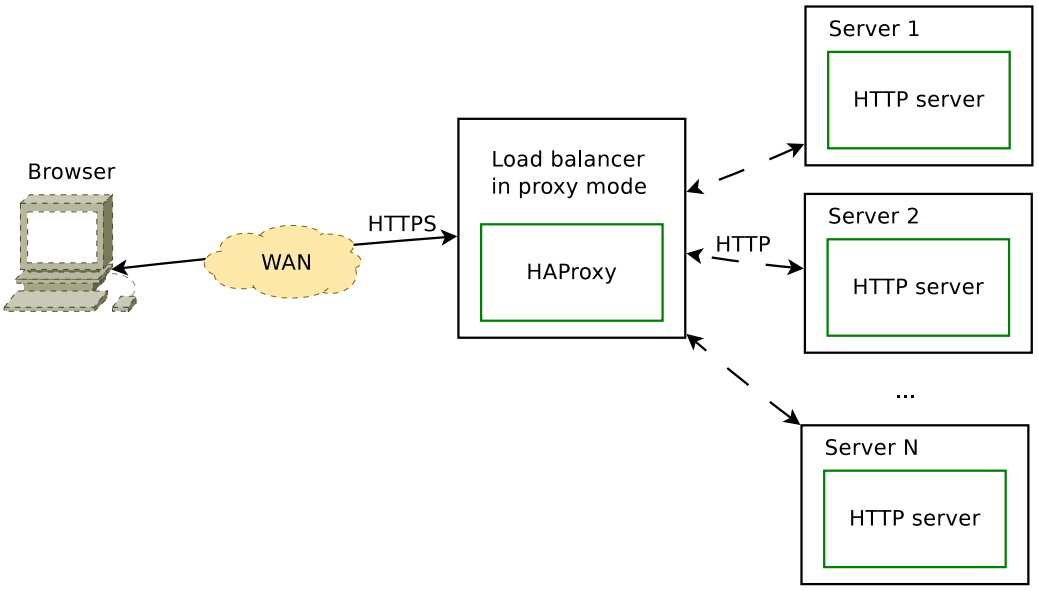


Figure 4.4: A diagram of a load balancing in HAProxy

Chapter 5

Verification and Testing of The Implementation

In order to estimate performance improvement of previous implementation, in this section we introduce our designed tool for benchmarks based on OpenConnect, HAproxy implementation and file transfer use cases. There is also introduced the main idea of testing and implementation verification.

5.1 Limitations of `sendfile(2)` System Call

The original idea of optimizing copies to user space was based on use `sendfile(2)` syscall. This system call has following prototype:

Listing 5.1: Prototype of `sendfile(2)` function

```
1 ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

The purpose of this system call is to transfer data from `in_fd` to `out_fd` without copying data to user space. This is handy when one wants to transfer data efficiently without necessary copying to user space. For example a web server can use `sendfile(2)` syscall to send a file from a local disk drive to a socket without need to iterate over content of file, which would be copied to user space and from user space to the socket.

Basically, `sendfile(2)` transfers `count` bytes from file referenced by `in_fd` file descriptor to file descriptor `out_fd`. If `offset` parameter is not `NULL`, `offset` describes offset within the file (to be more accurate, see `sendfile(2)` manual page).

The implementation of `sendfile(2)` in Linux kernel is based on `splice(2)` system call. The prototype of `splice(2)` is following:

Listing 5.2: Prototype of `splice(2)` function

```
1 ssize_t splice(int fd_in,
2                 loff_t *off_in,
3                 int fd_out,
4                 loff_t *off_out,
5                 size_t len,
6                 nsigned int flags);
```

This system call was designed to transfer data between file descriptors without copying to user space like `sendfile(2)` does. The difference here is that `splice(2)` syscall operates

on pipes, so one of the arguments has to be a pipe (source or destination). This system call is more universal and there can be implemented `sendfile(2)` syscall using `splice(2)` in user space. However this would lead to more context switches and a need of a pipe creation.

System call `splice(2)` is not part of POSIX standard. It is not portable, for example BSD does not introduce same system call. However, there can be seen system calls like `sosplice(2)`, `somove(2)` on BSD systems with different semantics, not part of POSIX as well.

System call `sendfile(2)` is Linux specific too, it is not considered to be portable. On the other hand, BSD systems introduce system call with exactly same name but slightly different semantics (and different prototype).

The implementation of `splice(2)` is very effective and does zero copy of data when transferring. The implementation operates directly on pages because of effectiveness. When there is issued `splice(2)` in the Linux kernel, pages are transferred from source to destination without any need of copying. This implementation is very effective, but involves special care of pages. For example when pages are moved to a pipe, they cannot be overwritten (to be more precise, there cannot be overwritten parts of pages that are given to a pipe). The underlying logic is discussed in 4.6.1.

There can raise a question why there is used a pipe as a mediator. This was heavily discussed on mailing list when `splice(2)` system call was introduced [28]. One of the core limitations of `sendfile(2)` is a restriction that this system call cannot operate on two or more buffers. There cannot be supplied any flag like `SPLICE_F_MORE` for `splice(2)`. Calling `sendfile(2)` involves iteration over in-kernel VM page cache so each requested page is sent via appropriate `sendpage` operation (with respect to offset and size that needs to be processed). Once the last page is going to be processed, `sendpage` operation is called without `MSG_MORE` flag that tells `sendpage` implementation that no more pages are going to be processed in the current action.

By calling `splice(2)`, user space can process multiple input buffers by explicitly telling that there are more caches to be process by supplying `SPLICE_F_MORE`. Since `splice(2)` uses pipes, this flag is useful mostly for destination file descriptors. Code snippet 5.3 demonstrates a usage of `splice(2)` call when multiple input buffers are processed (this cannot be simulated using `sendfile(2)` as Linus Torvalds pointed out [28]).

Listing 5.3: Example of `splice(2)` chaining

```
1 splice(fd_in1, NULL, pipe_out, NULL, len1, 0);
2 splice(fd_in2, NULL, pipe_out, NULL, len2, 0);
3 // fd_out will consist of fd_in1, fd_in2 without user space copy
4 splice(pipe_in, NULL, fd_out, NULL, len1 + len 2, 0);
```

Both system calls, `sendfile(2)` and `splice(2)` use pages. System call `sendfile(2)` is implemented on top of `splice(2)` system call with use of a private pipe, but requires `mmap` operation defined (available file page cache). This approach limits `sendfile(2)` so it cannot be used with file descriptors that refer to a streaming device. On the other hand this is one of the key benefits of using `splice(2)`. Having explicit pipe gives user space an ability to correctly recover if an error occurs on receiving file descriptor. If data are read from source file descriptor (that could be a streaming device for example) and written to the pipe using `splice(2)`, subsequently there is issued `splice(2)` call which transfers data from pipe to destination file descriptor and an error occurs on destination file descriptor, data are still available in the pipe so user space can decide what to do in such error situation. If there would be no pipe, data would be (possibly irrecoverably) lost.

Not all file descriptors (or socket descriptors) support `sendfile(2)` and `splice(2)` system call. A file descriptor can be used as a source or as a destination. On implementation level, we are considering the file descriptor to be source of data or receiver of data, source of pages or consumer of pages. For socket implementation it means, that we need to define `splice_read` operation when defined socket is used as a source in `splice(2)` call or `sendpage` when defined socket is used as a destination in `splice(2)` system call (analogically for `sendfile(2)`). Explanation to socket operations is given in section 4.6.5 section.

Not all sockets in Linux kernel can be used as source or destination in `splice(2)` system call. For example datagram socket from `AF_INET` family (UDP socket) cannot be used as a source in `splice(2)` system call, because there is no `splice_read` operation defined in `proto_opts` structure.

5.2 The Cost of a Context Switch and Data Copy

Operating system by its definition provides a set of system calls that can a process issue. Processes are running in unprivileged mode so there is supported encapsulation and safety between processes. The mechanism behind executing system call is very difficult and there have to be given spacial care, since system calls are the only way how can a process communicate with the world.

If we look introspectively on Intel x86 architecture family, which is the leading architecture on desktops, the way how to perform a system call was to issue `int 0x80` interruption in the past. This instruction is pretty expensive. Some operating systems implemented system calls by issuing invalid instruction in order to avoid the cost of `int 0x80` instruction [30].

For executing instruction that does system call on Linux is responsible standard C library. Functions from this library do all the work necessary to perform system call (such as arguments preprocessing if needed). C on Linux x86 in ELF format uses `cdecl` calling convention because of architecture support. C declaration calling conversion (`cdecl`) expects parameters on the stack, Gnu libc wrapper should ensure that these parameters are passed in correct registers and that correct number of system call is issued (on x86 architecture, system call number is passed in `AL`, `AX`, `EAX`, `RAX` register respectively based on architecture bit version). How arguments of a system call are passed to the kernel is heavily dependent on architecture and kernel's ABI for particular architecture.

Once the system call is executed, the return value of system call is handled by Gnu libc wrappers, which names correspond to system calls. The convention of return value says that negative return values mean errors. If a negative number is returned from kernel, the absolute value of return value (on x86 is return value placed in `AL`, `AX`, `EAX` or `RAX`) is assigned to `errno` variable (Gnu libc implements `errno` as a macro, that ensures that the access of `errno` value is thread-safe). On error, Gnu libc wrappers return -1 to indicate an error. There can be other post-processing done depending on the actual system call semantics.

There is a possibility to call directly a system call that does not have implemented Gnu libc wrapper. More info can be seen in `systemcall(2)` man page.

Since system calls are quite common, there was optimization done on x86 architecture. This architecture tends to be always backwards compatible with previous generations, so there can be still executed `int 0x80` interrupt in order to call kernel's system calls. Modern operating systems, as Linux is, implement a mechanism called *Virtual Dynamically Linked*

Shared Objects, more known in an abbreviated form `vDSO`. This mechanism makes system calls faster. By using `vDSO`, there can be mapped kernel space routines into user space process. The routines that can be mapped have to be carefully chosen – there can mapped only routines that are safe to handle from user space (like `gettimeofday(2)`, `getpid(2)`). More difficult routines (such as `open(2)`, `sendmsg(2)`, etc.) require big kernel processing, checks and logic, so there is issued a "real" system call. More information regarding vDSO can be found in `vdso(7)` man page.

On x86_64 architecture, system call is currently implemented by using `syscall` instruction (this instruction is not available on AMD processors and is invalid on x86 32 bit architecture; there is available `sysenter` with different handling).

There is also significant difference in system call type. If a system call can immediately return there is a *fast path* mechanism in Linux kernel. This can be applied to system calls such as `sendmsg(2)`, which should not block. On the other hand, there are system calls that can block. It can be expected that there will be a context switch so kernel prepares for it.

Once we enter kernel code, there has to be ensured that memory management is consistent, so the kernel can correctly handle pointers (both, from user space and in the kernel). To ensure this, Linux maps kernel memory into each process memory. The process cannot directly use this memory, but once the process does context switch, the kernel does not have to invalidate memory management unit (MMU) for correct addressing (if there is not scheduled another process). This has a serious impact, since the process cannot use possibly whole range of a pointer. On 32 bit architecture there is no possibility to address whole range (2^{32} – 4GiB) without any special care. On 64 bit architecture, this is not a big issue now because limit 2^{64} is currently not reachable. In the kernel, there can be seen terms like *highmem* and *lowmem*, which correspond to kernel virtual address space – *highmem* mapped into upper part of process virtual address space (*lowmem*).

If we consider these optimizations, measuring a context switch is very hard. There were approaches to measure the cost of a context switch [38], but all of them approximate actual cost because of complexity and factors that affect a context switch time. Nevertheless it appears that primary focus on a context switch is not necessarily a bottleneck on x86 architecture. On the other hand, OpenConnect can be used on different architectures, such as ARM or MIPS where design and cost of a context switch is different.

5.3 Benchmarks Design

There were introduced ideas behind OpenConnect implementation in section 4. OpenConnect reads TLS or DTLS records from a socket, decrypts them and sends them to TUN device in order to do routing in the kernel. All benchmarks tend to simulate scenarios listed in sections 4.7.1 and 4.7.2.

Since TUN device is a file of a special type, the implementation qualifies TUN device as a special socket type (see `drivers/net/tun.c` of Linux git tree for implementation details). In order to focus as strictly as possible on bandwidth optimization, there was written a tool that does separate benchmarks to specific sections, which should be benchmarked.

ehe implementation of this tool is part of appendices, it uses GnuTLS library in order to perform handshake and based on an established connection, passes key material with sequence numbers to `AF_KTLS` (see `ktls.c` for implementation details). There are also implemented scenarios that simulate OpenConnect, HAProxy and unoptimized sending a file as it is used nowadays.

The following subsections discuss implemented benchmarks and what scenarios they simulate. The implementation of benchmarks can be found in source file `action.c` of implemented `af_ktls-tool`.

5.3.1 Benchmarks of file transfer

To benchmark Facebook's approach of sending a file using `sendfile(2)` system call, there were designed appropriate scenarios. Optimization design can be seen in section 4.7.1.

Since we want to compare results, by supplying `--sendfile-user FILE` one can test currently used approach of sending files via TLS or DTLS protocol. The application sequentially reads file to a buffer, encrypts it and sends it via user space library (GnuTLS). There can be specified payload which describes buffer size (payload carried within TLS or DTLS record).

DTLS deserves a special note. Since benchmarks that use DTLS use a UDP socket, there is no congestion and delivery guaranteed. If a user supplies a file that size is too big, most likely some parts of the file will be lost, since receiving queue fulfills and kernel starts to drop UDP datagrams if the server does not read datagrams fast enough.

Optimized version uses implemented `AF_KTLS` Linux kernel module, that does record assembling and encryption based on handshake done in GnuTLS. To run `sendfile(2)` with `AF_KTLS` socket, there can be specified `--sendfile FILE`. Implemented tool correctly instantiates implemented `AF_KTLS` socket and uses this socket to send desired file (the payload is configurable via `setsockopt(2)` call and implemented tool can adjust payload by supplying `--senfile-mtu SIZE`).

Linux kernel offers an ability to map a file into memory and operate on file's content in memory. This operation can be done using `mmap(2)` system call. This approach was also studied. A file can be mapped by supplying `--sendfile-mmap FILE` option. The content of the file will be mapped to memory and implemented tool will iterate over the content, it will encrypt it using GnuTLS and send it. The payload for transfer can be also adjusted.

There were also tested scenarios when there was no TLS/DTLS encryption and decryption involved. This helped to compare the cost of system calls used without TLS/DTLS impact. Since `sendfile(2)` can be emulated using `splice(2)`, this scenario was also studied. These scenarios can be evaluated by supplying `--raw-sendfile FILE`, `--raw-splice-emu FILE` and `--raw-sendfile-mmap FILE`.

5.3.2 Benchmarks of OpenConnect and HAProxy simulation

In order to test OpenConnect and HAProxy scenario, there were designed benchmarks, which focus on particular `splice(2)` optimization as stated in section 4.7.2, there were designed benchmarks based on OpenConnect implementation. These benchmarks are also applicable to HAProxy as stated in section 4.7.2. The current implementation of OpenConnect and HAProxy uses approach listed in section 4.7.2.

Since OpenConnect uses TUN device, there was designed a scenario, which reads from an `AF_KTLS` socket `ksd` (which does decryption and record disassembling) and writes to a pipe because of `splice(2)` system call design. With the second `splice(2)` system call the decrypted record is sent to a socket `sd`, which simulates TUN device. For testing purposes, there was initialized one connection with a server. The server was receiving unencrypted data, which were encrypted and echoed back in TLS/DTLS record. This approach corresponds to designed optimization listed in section 4.7.2 (on client side). The

key part of benchmark is listed in 5.4. There has to be done initial send of data, that are transmitted between the client and the server.

Listing 5.4: Optimized OpenConnect and HAProxy use case

```
send(sd, buf, mtu, 0); // initial send
start_timer();
while (!benchmark_should_finish) {
  splice(ksd, NULL, pipe[1], NULL, TLS_RECORD_MAX_LEN, 0)
  splice(pipe[0], NULL, sd, NULL, TLS_RECORD_MAX_LEN, 0)
}
```

The implementation uses `splice(2)` so there is no copy to user space involved even there has to be a pipe used. There is no possibility to use `sendfile(2)` system call in order to save one context switch. The reason why `sendfile(2)` is not possible to use is analyzed in section 5.1.

This approach was also `AF_KTLS` involved – just with raw TCP/UDP sockets without TLS/DTLS. Refer to `--splice-echo-time TIME` and `--raw-splice-echo-time TIME` arguments in designed benchmark tool. The payload can be adjusted by supplying `--payload SIZE` argument (defaults to 1400).

5.4 Notes to Benchmarks and their Visualization

Even there is implemented encryption using user space library GnuTLS, OpenSSL was not studied. The reason is that OpenSSL shares cryptographic code implementation that does AES GCM encryption and decryption with GnuTLS. The implementation of `AF_KTLS` socket was tested with OpenSSL by Dave Watson from Facebook who proposed one patch to make `AF_KTLS` work with OpenSSL. I accepted this patch.

There were used sockets of type `AF_INET`, but `AF_INET6` could be used as well (for supporting other families, there has to be appropriate implementation for each socket family). Implemented socket `AF_KTLS` works with `AF_INET6`, but there is no performance difference expected from IPv6.

The original Facebook's patch implementing TLS with `AF_ALG` was not included in benchmarks, as mentioned in 4.4, there was no positive impact reproduced on tested hardware. Moreover, the test case for OpenConnect and HAProxy scenario is not possible to reproduce, since Facebook's patch does not support `splice_read` operation.

There is implemented a server which can run in a separate thread. Running the server in a thread affects benchmarks results because timer used counts time spent in user space and kernel space for each thread. It is recommended to run the server part as a separate process.

It is important to choose the correct timer to measure time. There are different timers available on Linux. Since we want to analyze time spent in kernel space and in user space, we have to choose appropriate timer. There was used `ITIMER_PROF` which decrements when process spends in kernel space and user space as well. Upon expiration, `SIGPROF` signal is delivered. See `setitimer(2)` man page for more info.

Another important note implies to operating system cache, since operating system does caching in order to eliminate slow hard disk drive access (compared to memory access). Before each run of a test there has to be forced cache flush. Linux kernel enables this by writing appropriate value to `/proc/sys/vm/drop_caches` kernel's virtual file. Dropping cache can be explicitly done in benchmarks by supplying `--drop-caches` (root needed).

To avoid side channels introduced by filesystem or disk operations, tests were also evaluated using `tmpfs` file system (which maps directly into memory).

There were implemented two types of benchmarks for OpenConnect (and HAProxy) scenario. There are available timed benchmarks, which run specified benchmark for n seconds (see `*-time` parameters). There can be also done specified number of transmissions (see `*-count` parameters). It is recommended to run benchmarks for specified time in order to avoid impact of various side channels. CPU runs in a power safe mode by default, only if there is a heavy computation, the frequency is raised. This helps to save energy, but has bad impact on tests, since CPU is not using its whole potential and frequency can vary across tests.

5.5 Benchmark Results

The designed and implemented tool for benchmarks can run all studied scenarios based on command line arguments passed. Benchmarks output is a readable text which is suitable for a human. To process results of benchmarks automatically, there can be supplied `--json` argument, which tells the application to output results of benchmarks in JSON format. There was designed automated visualization tool written in Python, which processes JSON, computes statistics and makes visualization in Gnuplot or in an HTML format.

Automated bash script `benchmarks.sh` (part of appendices) executes all possible use cases that are suitable for analysis and comparison. Benchmarks were executed on two laptops with different configuration. Both laptops have support of AES-NI instructions.

5.5.1 Benchmark Results on Lenovo ThinkPad T540p

Initially were benchmarks done on a laptop Lenovo ThinkPad T540p with CPU Intel Core i7-4900MQ, 2.80 GHz and 8GB RAM. The main issue was hard drive. There was no solid state drive (SSD) and it appeared as a main bottleneck in optimization of file transfer scenario.

When optimization of file transfer scenario was evaluated, there were big variances in benchmark results. These variances were caused by inconstant access time on rotating platters. Because of big variance, hardware was changed to Lenovo ThinkPad T440p with a solid state drive. Nevertheless, results of benchmarks for T540p model with rotating platters are also available on attached DVD but because of big variance caused by platters, they are not analyzed here.

5.5.2 Benchmark Results on Lenovo ThinkPad T440p

The second run of benchmarks was done on Lenovo ThinkPad model T440p with CPU Intel Core i7-4600U, 2.10GHz and 12GB RAM. This laptop has an SSD drive so there can be expected smaller variances in access time. The benchmarks presented in following sections were done on `tmpfs` filesystem because of constant access time.

There were evaluated test cases for different payload size – for payload of size 1280, 1400, 4000, 6000, 9000, 13000 and 16000 (each was run three times, these tests are also available for T540p model as stated in section 5.5.1). Payload of size 1280 is relevant because it is minimal payload that can be carried in IPv6 packets. MTU of size 1400 is close to Ethernet limit 1500 (1400 is just payload, there are additional headers for protocols). Bigger payloads

(16000, 1300) are nowadays not seen in regular traffic, but there can be seen jumbo frames that can be up to 9000B on gigabit switches.

Benchmarks also covered both implemented protocols in `AF_KTLS` – TLS and DTLS. For each benchmark scenario there were evaluated three runs and the highest rated result was chosen. For comparison, raw TCP and UDP were also studied.

Since TCP and UDP records can follow different kernel code path when *localhost* is used, there was created a testing environment using 2 laptops – Lenovo ThinkPad T440s and Lenovo ThinkPad T540p (same as stated in sections 5.5.1 and 5.5.2). These laptops were directly connected using crossed cable with manually configured IP addresses. Benchmarks with server and client on the same host that were evaluated are available on attached DVD as well. These benchmarks do not conform to the typical networking usage and because of the different in – kernel paths, the results are not relevant for desired benchmark purposes. Studying different kernel paths inside kernel for *localhost* in use is out of scope of this thesis.

All results of benchmarks are available on attached DVD as an HTML describing each benchmark configuration and comparison. There are also available benchmarks on DVD, which are not presented here (see 5.5.1).

Benchmarks of Transmission

In order to compare bandwidth, we evaluated benchmarks of `sendmsg(2)` and `recvmsg(2)` on `AF_KTLS` socket in comparison to GnuTLS library – `gnutls_record_send()` for sending and `gnutls_record_recv()` for receiving. Results of benchmarks can be seen on figures 5.1 and 5.2. The analysis is given in section 5.6.

The concrete numbers with comparison can be seen in appendices C.1 and C.2. As can be seen on the plot, `AF_KTLS` socket implementation is slower than GnuTLS for both – TLS and DTLS.

Benchmarks of Optimized File Transfer

Scenarios that were evaluated can be seen in simplified pseudo-codes 5.5, 5.6, 5.7 and 5.8. The pseudo-code listen in 5.5 is a typical approach of sending a file over TLS/DTLS using GnuTLS (OpenSSL can be used as well). This approach can be optimized with `mmap(2)` system call, which maps the content of a file into memory (if there is enough available memory). There was also studied `sendfile(2)` and `splice(2)` approach, the pseudo-codes can be seen in 5.6 and 5.8. The same idea is shared with benchmarks, that test TCP and UDP sockets (there is used TCP/UDP socket instead of `AF_KTLS` and GnuTLS call `gnutls_record_send()` is substituted with `send(2)` with appropriate parameters).

For comparison figures 5.3 and 5.4 demonstrate raw TCP and UDP traffic. The `splice(2)` benchmark was implemented to simulate `sendfile(2)`, but the pipe was explicitly in user space. As stated in section 5.1, `sendfile(2)` system call is implemented in the kernel using routines as `splice(2)` would be called. The `splice(2)` scenario was intended to simulate and prove that `sendfile(2)` is as fast as `splice(2)` system call (if we consider cheap context switch, see 5.2). This contention was proven by benchmarks. As we can see, `sendfile(2)` and `sendfile(2)` emulation using `splice(2)` outperforms user space sequential reading and sending a file because of zero copy optimization.

There was transmitted 100MB file. It was ensured that the whole file was transmitted without UDP drops in case of UDP and DTLS benchmarks.

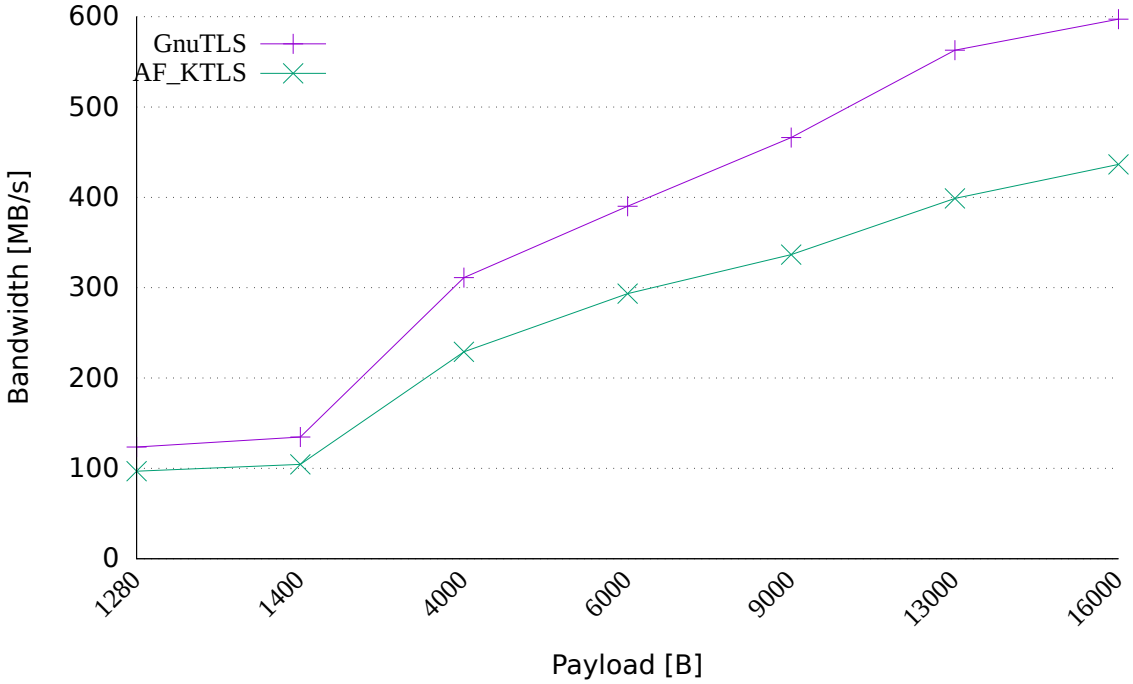


Figure 5.1: Results of transmission benchmarks – DTLS

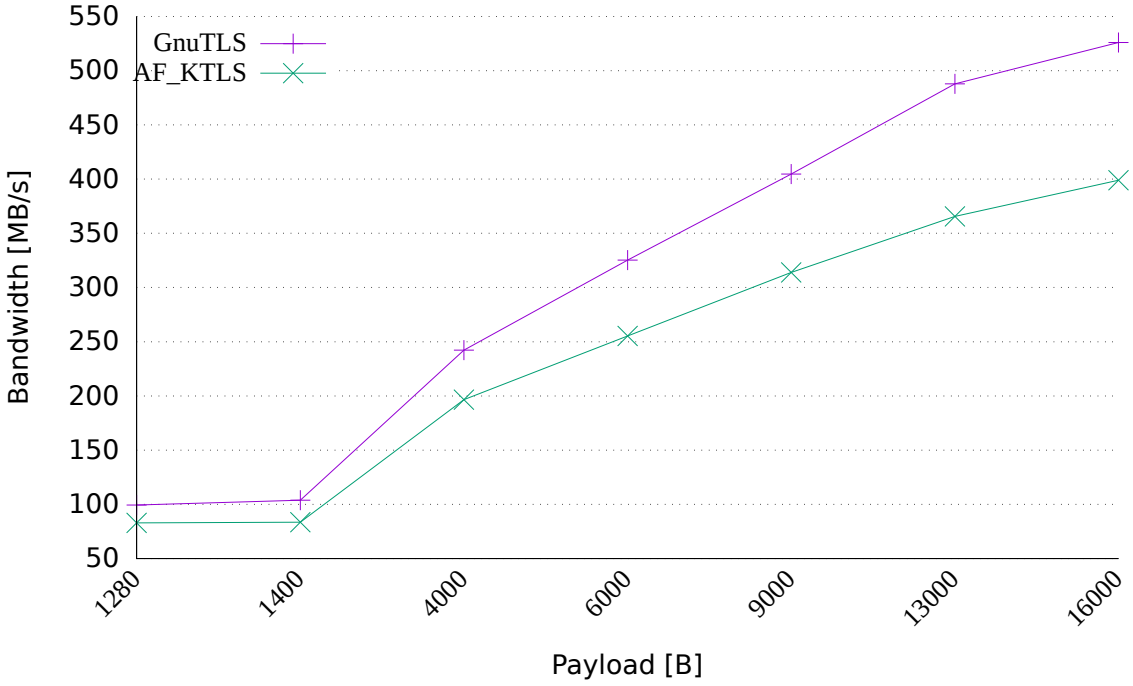


Figure 5.2: Results of transmission benchmarks – TLS

Listing 5.5: A simplified user space file transfer scenario – `user-send`

```
for (size_t total = 0; total != filesize; total += payload) {
  read(fd, buf, payload);
  gnutls_record_send(session, buf, payload);
}
```

Listing 5.6: A simplified `sendfile(2)` file transfer scenario

```
// ksd is initiated AF_KTLS socket descriptor
senfile(ksd, fd, NULL, filesize);
```

To analyse implemented `AF_KTLS` Linux kernel module, there were evaluated similar benchmarks to raw TCP and UDP sending a file, but the file was encrypted. For `user-send` and `mmap(2)` scenarios, the file was encrypted using GnuTLS library with user space copy; for `sendfile(2)`, `AF_KTLS` was used. The `sendfile(2)` emulation using `splice(2)` was not studied because of shared implementation with `sendfile(2)`.

Listing 5.7: A simplified `sendfile(2)` file transfer scenario

```
mem = mmap(NULL, filesize, PROT_READ, MAP_PRIVATE, fd,/*offset*/ 0);
for (size_t sent = 0; sent < filesize; sent += payload) {
  gnutls_record_send(session, mem + sent, MIN(payload, filesize - sent));
}
```

Listing 5.8: A simplified `splice(2)` transfer scenario – user space `sendfile(2)` emulation

```
for (size_t total = 0; total != filesize; total += payload) {
  splice(fd, NULL, pipe[1], NULL, payload, 0);
  splice(pipe[0], NULL, sd, NULL, payload, 0);
}
```

As can be seen on figures 5.5 and 5.6, `AF_KTLS` outperforms user space reading and encrypting by 11%.

The exact numbers with comparison of presented benchmarks are available in appendix C.3 and C.6.

Benchmarks of OpenConnect Scenario

The last presented benchmarks tend to simulate OpenConnect scenario – reading from TUN and sending to the appropriate socket as and its optimized form using `splice(2)` system call as described in 4.7.2.

As can be seen, the current approach used nowadays using GnuTLS outperforms `AF_KTLS`. The exact numbers of presented benchmarks for each TLS/DTLS payload tested with comparison are available in appendices C.9 and C.10.

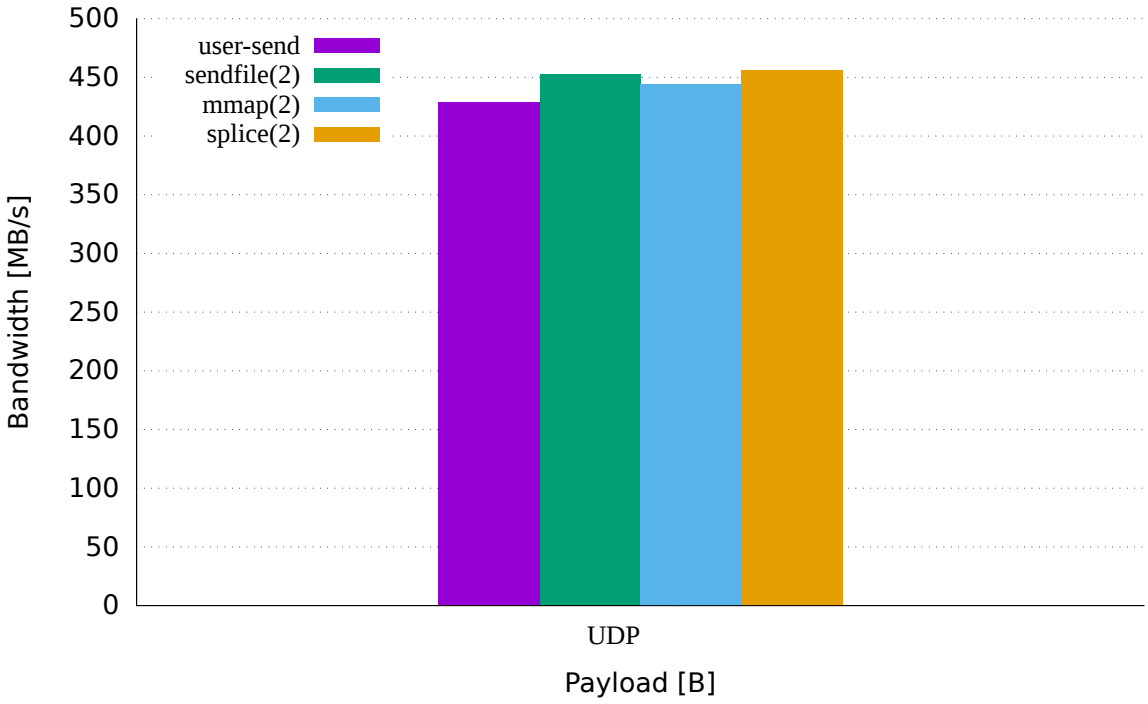


Figure 5.3: Results of sending a file from `tmpfs` – UDP with payload 1400B

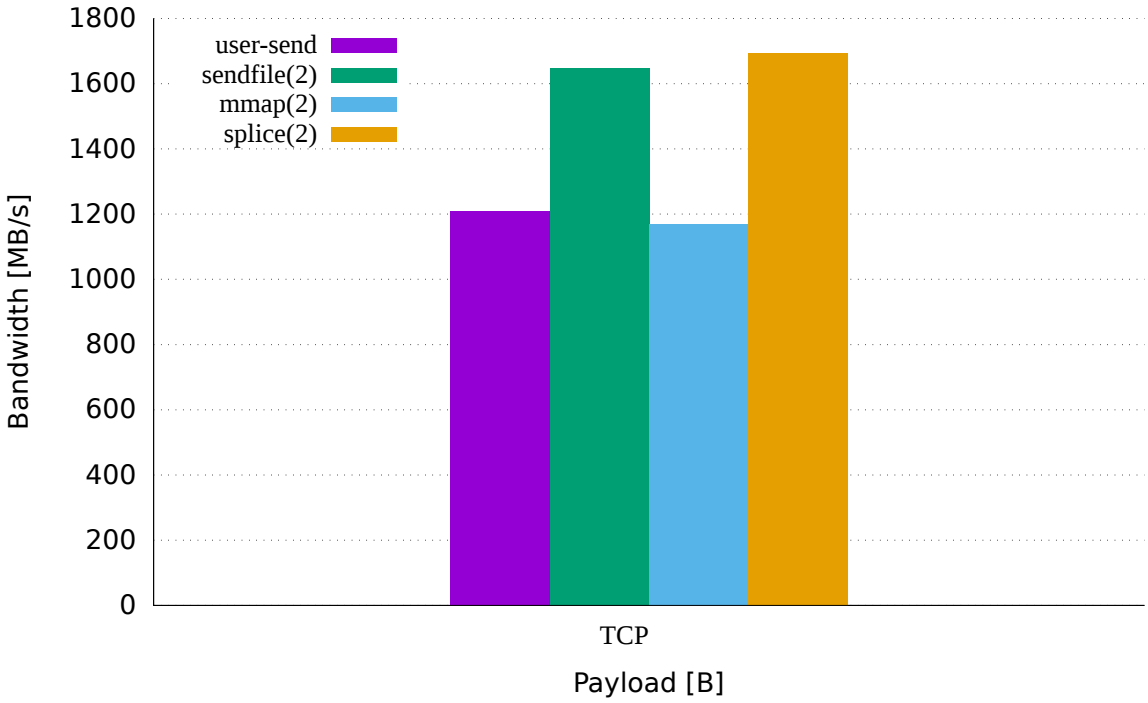


Figure 5.4: Results of sending a file from `tmpfs` – TCP with write of 1400B chunks

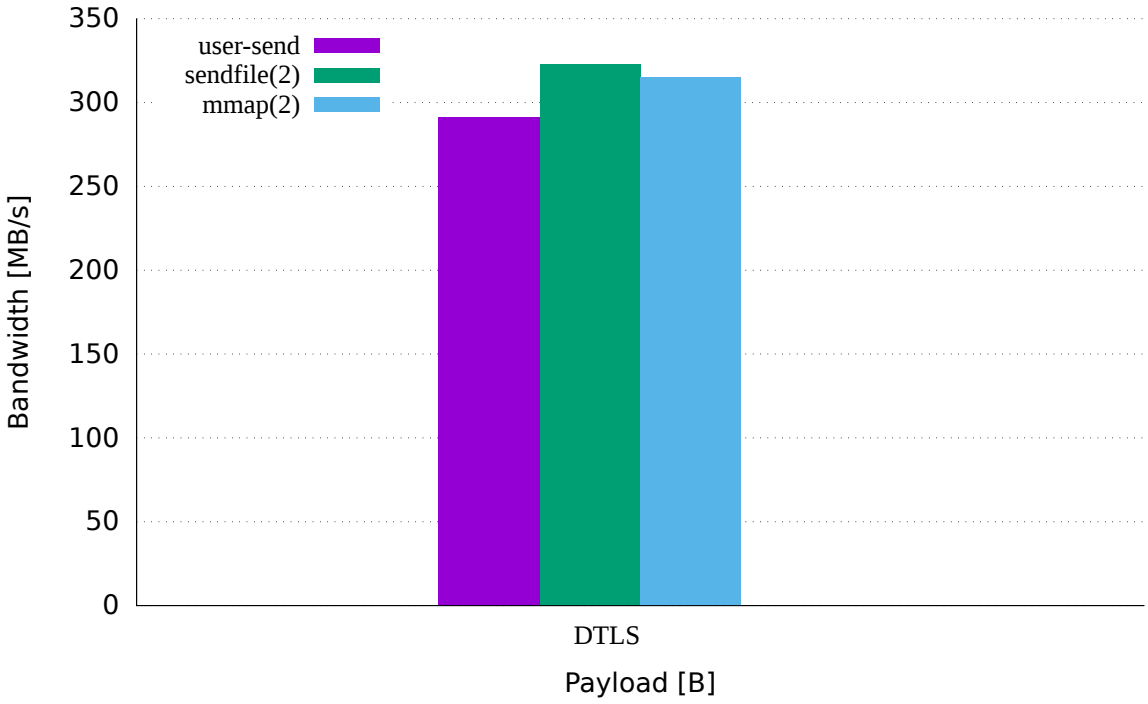


Figure 5.5: Results of sending a file from `tmpfs` – DTLS with payload 1400B

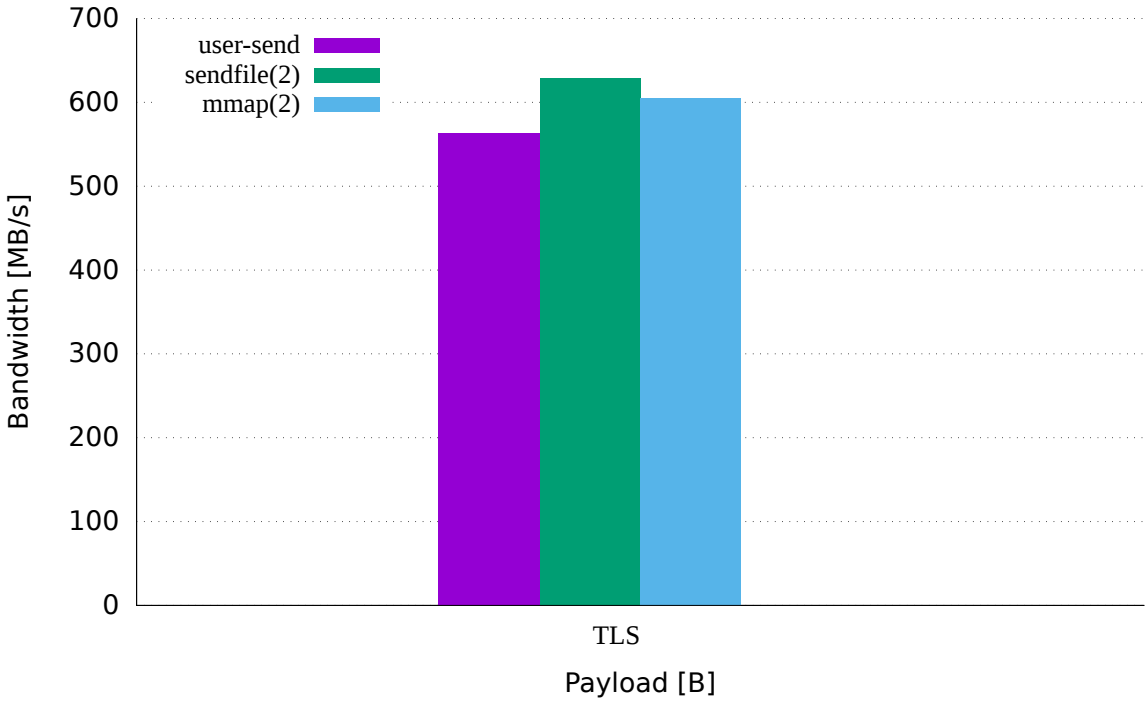


Figure 5.6: Results of sending a file from `tmpfs` – TLS with write of 1400B chunks

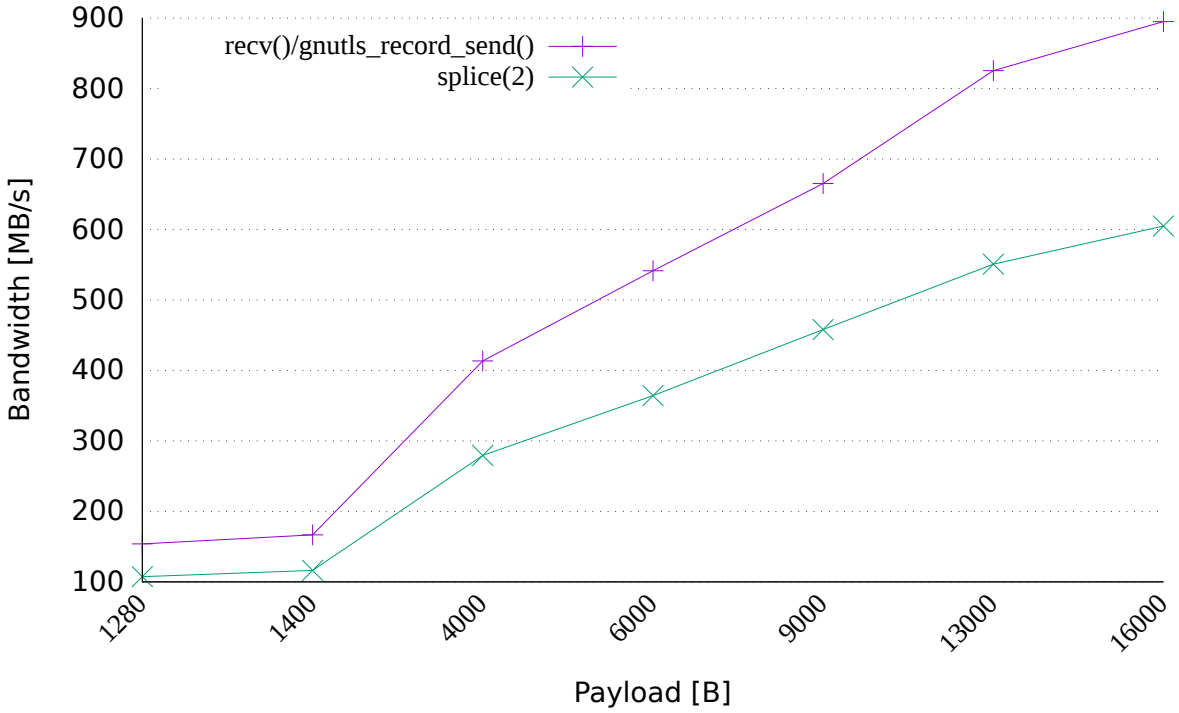


Figure 5.7: Results of benchmarks simulating OpenConnect – DTLS

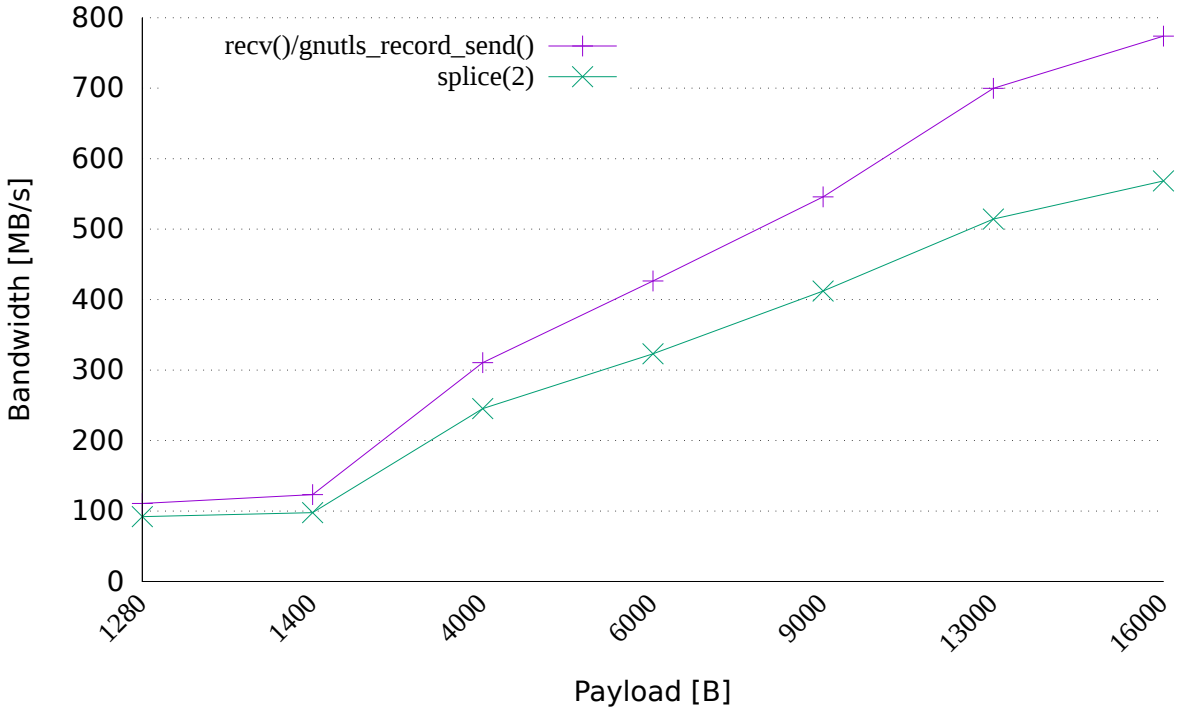


Figure 5.8: Results of benchmarks simulating OpenConnect – TLS

5.6 Analysis of AF_KTLS Implementation and Limitations

According to benchmarks introduced in section 5.5, current AF_KTLS implementation appears to be slower than user space libraries. To study the bottlenecks, there were evaluated benchmarks with `perf`, that recorded all the data necessary to diagnose bottlenecks in implementation based on introduced optimization in section 5.3. There was transferred a file, generated from `/dev/urandom` of size 1GB, with highest possible MTU using TLS. According to `perf`, 99.74% of whole application run was spent in the kernel. Kernel used *fast path* method for context switches (see section 5.2) – time spent in kernel to user context switches was so negligible that perf reported 0%.

Besides other kernel processing (like permissions check, `sendfile(2)` system call overhead, mapping to page reads etc.), there was only 66.61% spent in `tls_sendpage()` implementation and its subsequent calls. 47.45% of total time was spent in RFC 5288 implementation, where 30.05% was the actual encryption. Besides that 11.73% of total time was spent in data copy and 1.24% in additional memory allocation.

On the other hand, on sending there was spent 17.32% in `kernel_sendmsg()`. Actual transmission took only 7% and 4.23% of total time was spent on copying record to sending queue. If we consider that the AES GCM encryption could be optimized to avoid copy and allocation there could be ideally saved 12.97%. In our code can be also avoided `kernel_sendmsg()` that does copy and rather use `kernel_sendpage()`, which ideally directly stores pages in sending queue of a socket. This approach could possibly save additional 4.23% of total time. In total, ideally 17.20% could be saved by introducing additional optimizations.

There were also studied bottlenecks for OpenConnect and HAProxy optimization. In benchmarks, 44.24% of total time spent in `kernel_sendmsg()`, where 38.28% of total time spent in `tcp_push()` - on actual sending, 1.15% of total time spent in allocation socket buffers (`skb_stream_alloc_skb()`) and nearly 2% on copy from kernel vector (in `copy_from_iter()`, `memcpy_erms()`). On the other hand, 33.14% of total time spent in `splice_read` operation, where 13.14% of total time spent in `kernel_recvmsg()` and another 2% on copy and allocation (`skb_copy_datagram_iter()`, `copy_page_to_iter()`). Allocation and copy could be avoided as well to save additional 5%.

The main bottleneck in the transmission benchmarks is Linux Crypto API allocation and copy (which is done twice in this scenario – when receiving and sending).

Even there were diagnosed bottlenecks, introducing new approaches with substituted `kernel_sendpage()`, removed copy when receiving records and optimized RFC 5288 implementation could uncover another issues and performance drawbacks due to different call paths. Test was also evaluated with DTLS, results were similar so they are not explicitly stated here (the only significant difference was time spent on actual transmission which was lower because UDP does not need wait for acknowledgments).

5.6.1 GnuTLS and Linux Crypto API Comparison

To compare GnuTLS encryption and Linux Crypto API, there was evaluated a benchmark. Linux Crypto API can be benchmarked with Cryptodev-linux tool, but at the time of writing, there was a bug in Cryptodev-linux that made it impossible to run this benchmark tool.

In order to do comparison, there was simulated transmission with `af_ktls-tool`, that send and received 1,000,000 times a record with payload 1400 bytes (the data size is con-

stant). Time spent in encryption and decryption is shown in table 5.9 (reported by `perf`).

Operation	Linux Crypto API	GnuTLS
AES GCM encrypt	10.35%	7.15%
AES GCM decrypt	11.40%	8.42%

Figure 5.9: Time spent in encryption and decryption when transmitting 1,000,000 times a TLS record with payload 1400B

According to `perf`, 1.69% was spent on copy and 0.86% was spent on allocation in Linux Crypto API (0.47% `kfree()` and 0.39% `kmalloc()`). This allocation is done only if associated authentication data, plaintext and tag do not fit within a single page. The current `AF_KTLS` implementation is chaining associated authentication data and tag with preallocated pages for plaintext, so the allocation and copy are done always. By optimizing Linux Crypto API with allocation and copy removal, similar results as in user space using GnuTLS could be seen.

5.6.2 TUN/TAP Socket Implementation and its Limitations

In order to use TUN/TAP device with `splice(2)` system call, even TUN/TAP socket has to have implemented `splice_read` and `sendpage` operations. Unfortunately, TUN/TAP implementation currently supports only `sendmsg(2)` and `recvmsg(2)` system calls.

Lack of `splice_read` and `sendpage` implementation is a serious issue, since the original idea of optimizing OpenConnect and HAProxy was to use `splice(2)` on a TUN device socket. Since there is no implementation for `splice_read`, TUN cannot be used as a source of data in `splice(2)` call. Lack of `sendpage` routine implementation makes it impossible to use TUN as a destination in `splice(2)` and `sendfile(2)` calls.

To make OpenConnect or HAProxy optimization possible with `AF_KTLS`, there has to be added implementation for `splice_read` and `sendpage` socket routines.

5.7 Testing the Implementation

In order to automatize testing and verify implementation, there were designed tests. Since different system calls can have different paths inside kernel (as stated in the section 4.6.5), there were designed benchmarks which test all possible cases.

Implemented `af_ktls-tool` tries to test implementation based on command line arguments. If `--verify-sendpage` was supplied, there is tested `kernel_sendpage` implementation. On the other hand, if `--verify-transmission` was supplied, there are tested `sendmsg(2)` and `recvmsg(2)` system calls (and system calls that are transparently mapped to `kernel_sendmsg()` and `kernel_recvmsg()` as stated in section 4.6.5). The last supported transfer test issued on `--verify-splice-read` tests kernel `splice_read` implementation.

All of the tests are performed on different payload that is transmitted. There are covered test cases when data transmitted fit within one single page or they are spread across multiple pages (or size cannot be sent due to TLS/DTLS record limit).

Data transmission is not the only part that needs to be tested. There has to be guaranteed correct socket error handling from kernel's point of view so the kernel implementation

does not cause crash when wrong arguments are supplied. To avoid such cases, there were implemented tests, which can be run via `--verify-handling`.

In order to test `AF_KTLS` window implementation, the implementation of `AF_KTLS` was isolated and there was designed a test suite. Nikos Mavrogiannopoulos proposed a patch that enhanced sliding window testing with `cmocka` support (a lightweight library to simplify and generalize unit tests for C[1]).

The implementation of sliding window can be seen in `dtls-window.c` and tests can be seen in `tests/dtls-window.c`. Tests can be automatically run by issuing `make check`.

[1] https://cmocka.org

Chapter 6

Future Work & Vision of AF_KTLS

This chapter summarizes work that needs to be done in order to get AF_KTLS module upstream. There are discussed possible uses of AF_KTLS that could benefit other user space applications if the module would get merged by upstream.

6.1 Community Feedback

Even so the implementation was tested and is working, there was not proposed upstream patch yet. The reason is, that there are still open issues and space for optimization that should be considered, but the work was already published on GitHub [1] with all the notes related to implementation.

Since Facebook was interested in optimization of sending files via `sendfile(2)`, Facebook engineer that was the author of original Facebook approach, Dave Watson, was contacted. Based on source code review, he pointed out that current implementation of AF_KTLS uses `kernel_sendmsg()`, which does a copy of the content to the sending queue of a socket and `kernel_sendpage()` should be used instead. The communication is active and there is possibility of future cooperation in order to get AF_KTLS upstream.

Lately Intel proposed a patch that was implementing TLS-type encryption into Linux kernel [47] (there were also proposed TLS 1.0 support in 2014 [46]. The author of proposed patch, Tadeusz Struk, was contacted on mailing list with a reference to AF_KTLS implementation. He pointed out that current implementation is hard to extend with AES-CBC-HMAC-SHA1 cipher. Nikos Mavrogiannopoulos, the principal author of GnuTLS, argued that this cipher needs a lot of hacks to be implemented correctlyand it is not suitable for kernel space implementation. This cipher is also banned in HTTP/2.0 and it will be dropped from next TLS version, TLS 1.3 [18, 27].

We expect more community feedback once implementation of AF_KTLS will be proposed on kernel mailing list.

6.2 Work Needed to Optimize an SSL VPN

Based on work done in order to implement AF_KTLS and its analysis, there were found bottlenecks in the current implementation. By solving these bottlenecks, AF_KTLS would have better results, which would lead to OpenConnect optimization.

[1] https://github.com/fridex/af_ktls

- The implementation of RFC 5288 should be optimized in order to avoid copy and allocation. This allocation and copy is done when source does not fit within one single page. The original author of Facebook's patch, Dave Watson, is already working on a patch that would avoid additional copy and allocation.

- To transparently support different actions on `AF_KTLS` socket based on decrypted data, such as OpenConnect protocol support, there has to be added support for Linux Socket Filtering, which is derived from Berkeley Packet Filtering. This will be added once API of `AF_KTLS` will be stable.

- There should be introduced peek support for TCP and UDP that would not pop records from receiving queue. This peek would return directly allocated pages in socket receiving queue buffer (`skbuff`) so there would not be a copy involved.

- There should be substituted currently used `kernel_sendmsg()` call in `AF_ALG` with `kernel_sendpage()` which should avoid copies in most situations. Nevertheless, TCP and UDP implementations of `kernel_sendpage()` can do a copy based on sending queue size.

- TUN device currently does not support `splice_read` and `sendpage` operations in socket implementation (see section 5.6.2). In order to use TUN device with `AF_KTLS` socket using `splice(2)`, there should be introduced appropriate copy-less implementations in TUN device socket.

6.3 Work Needed to Merge Mainline Kernel

Even the implementation of `AF_KTLS` is tested, there are still open issues, that need to be done before the `AF_KTLS` will be proposed upstream to be merged with mainline. In order to propose `AF_KTLS` to upstream, there should be done work stated in section 6.2, but there is additional work that needs to be discussed with upstream and community.

The very first issue is API design. Since Linux kernels are backwards compatible, once there is added functionality, it should be consistent and ready to be supported.

One of the current API issues is lack of DTLS over TCP support. Even so DTLS was designed to be protocol that would operate over unreliable underlying protocol (such as UDP), but even over reliable protocol (such as TCP) [17], the current API design does not allow to run TLS over UDP, which is correct (since TLS requires underlying protocol to be reliable). There has to even added possibility to specify other underlying protocols (like SCTP) for future enhancements.

Another issue are return values. Return values returned to user space are the only way how a user can space diagnose what went wrong (note that libc assigns these values to `errno` global variable as stated in 5.2). These return values have to uniquely distinguish errors that occurred in the kernel space (such as bad decryption, bad record type, etc) so user space can adopt control flow based on errors that occurred. There have to be deeply analyzed errors and return values from crypto API, used sockets and `AF_KTLS` socket itself (such as no memory when allocating etc.), so they can be correctly propagated to user space. If there occurs a collision of return values with slightly different meaning, return values have to be remapped to correctly distinguish errors.

Even so AES GCM is one of the currently most used modern ciphers, `AF_KTLS` could be easily extended with support of different ciphers as well. The list of supported ciphers

should be selected with respect to cipher usage, Linux Crypto API support and possible visions to future use.

Current implementation of `AF_KTLS` asynchronous record decryption handling uses one worker per module. It should be considered to add a worker per socket instance, since workers are woken up every time a record arrives to the socket. If there is a socket which is used heavily and another one that is not used so often, but asynchronous decryption can be done occasionally, scheduled asynchronous decryption can uselessly wait in worker queue and once it will be ready to serve a record, the record could be already received by user space. Per socket worker would solve this, since worker will be instantiated for every socket instance and all work scheduled would be only for one particular socket.

6.4 Other in–kernel TLS Implementations

As the code patches by Facebook, Intel indicate, our approach of a kernel TLS type encryption and decryption is not new. Following subsections discuss about other kernels that directly support TLS.

6.4.1 Netflix's TLS SSL `sendfile(2)` Optimization

The same use case of sending files over TLS via `sendfile(2)` was already studied by Netflix. Netflix implemented TLS into FreeBSD kernel in order to be benefited by `sendfile(2)` zero-copy optimization [45]. The implementation is not public, but based on paper published, the implementation was bound to OpenSSL library where authors introduced a new library call `SSLsendfile()`. The idea of dealing with handshake and re-handshake is shared with `AF_KTLS` socket implementation – these control messages were dealt by user space library and all session configuration was sent to kernel by OpenSSL, which probed first if desired cipher was supported. The main difference is that Netfix used kernel's implementation only for sending. If desired cipher was not supported by FreeBSD's Open Crypto Framework[35, 37], there was done fall-back to OpenSSL's buffered sending implementation. Receiving was still made by adopted user space library.

Netfix's SSL `sendfile(2)` optimization was implemented even to nginx HTTP and reverse proxy server[2] in order to test performance impact. The results were not as good as authors were expecting. The performance increased from 8.5Gps to 9Gps (cca 5.26%). There were located reasons why performance gains were less then expected are the following. The implementation of AES GCM in OpenCrypto Framework did in-place encryption (which is not effective for AES GCM) and data were copied, since data coming from disc drive were marked as read–only in FreeBSD. Another issue was AES-NI implementation, which used floating point unit. FreeBSD normally does not save and restore FPU's state when a context switch occurs, but since Open Crypto Framework used FPU, the state had to be stored and restored on context switch, which required additional computational power that did not occur within OpenSSL library original approach [45].

6.4.2 Solaris's SSL in Kernel – `kssl`

Solaris has an interesting feature called `kssl` [43]. It is a full TLS protocol implementation in the kernel, but rather than implementing TLS on a socket, Solaris implemented a kernel proxy configurable from user space (via `ksslcfg`) which does TLS completely transparently

[2] http://nginx.org

to application layer. This benefits applications, since there is no need to add additional TLS logic into applications but only TCP communication. The application can be bound to port, that could be restricted by firewall not to be available on unwanted interface. SSL proxy is then configured to transmit data between configured ports (the one that application is bound to and SSL proxy port where application should be exposed). More info can be found at [43].

6.5 Possible `AF_KTLS` Socket Usage

The implemented `AF_KTLS` kernel module can have various uses. DTLS, but mostly TLS protocol is widely spread nowadays because they are standards of encrypted communication on the Internet. The module is suitable for kernels 4.2 and newer. It is not possible to use `AF_KTLS` with older kernels because of change in Crypto API interface [3].

There are use cases where `splice(2)` and `sendfile(2)` system calls are very efficient. These system calls can be used on `AF_KTLS` socket, which fully supports them. These system calls were designed for use cases when user space does not need an actual copy of the content like OpenConnect VPN or HAProxy.

Since TLS and DTLS protocols were designed to be used on the Internet, `AF_KTLS` is mostly, but not necessarily, suitable for network applications. Since there is implemented asynchronous decryption based on record arrivals, `AF_KTLS` is suitable for applications which do not wait in receiving queue. Based on asynchronous optimization design discussed in section 4.6.2, `AF_KTLS` will benefit applications on multi-threaded CPUs, which could execute decryption asynchronously. With this technique involved a user space application will have decrypted content of a record available with cost of a copy.

Another benefit of `AF_KTLS` kernel module is the fact that there is reused well known socket API transparently. This means that applications, that could be configured to operate in both encrypted (TLS/DTLS) or plaintext mode (TCP/UDP), they can reuse core implementation because of `AF_KTLS` transparent use. Handshake and initialization of `AF_KTLS` could be done separately and the application logic can be reused. This possibility was limited since libraries like OpenSSL and GnuTLS offers functions, that wrap the whole sending or receiving process above Berkeley sockets (`gnutls_record_send()`, `gnutls_record_recv()` for GnuTLS, `SSL_read()`, `SSL_write()` for OpenSSL).

[3] https://lkml.org/lkml/2015/6/22/112

Chapter 7

Conclusion

This thesis provides an analysis of the available software Virtual Private Network (VPN) solutions and its performance on the Linux system. This analysis is then used as a basis to determine performance bottlenecks, suggest performance improvements and further design and implement the most promising of them.

During our analysis we identified that context switches from user space to kernel space, which are common in software VPNs, are not considered a serious bottleneck on the Intel x86 family of processors. Our focus was set on data copies from user space to kernel space and vice versa during the VPN packet transmission. The latter, according to our analysis, poses the most significant performance bottleneck on current software VPN implementations based on TLS or Datagram TLS.

The result of this thesis is a Linux kernel module which does TLS and DTLS transmission and reception in kernel space. The module utilizes key material established during a TLS or DTLS handshake in user space. This reduces the required data copies and context switches and hence potentially improving the performance of software TLS-based VPNs in modern hardware, whenever used.

However, the benchmarks we performed on our implementation showed an unexpected discrepancy between the expected performance benefits and the actual performance. Further investigation uncovered a few rough edges on the Linux kernel's internal interfaces, which are summarized below. The current TCP and UDP internal API implementation does not allow zero-copy operations, something that neutralises our data copy elimination from and to user-space. Furthermore, the current TUN device implementation does not support the operations necessary for zero-copy operation. This limits the usage of our developed kernel module on TLS based VPNs without additional kernel modifications.

Another drawback we identified was located in the Linux Crypto API which is used for AES GCM encryption within TLS/DTLS transmission. The current implementation of AES GCM does allocation and copies the provided data to ensure alignment, something which affects performance significantly. Consequently, we require few changes to the existing Linux kernel in order for our module to achieve its full potential for our VPN use case.

On the other hand, despite the fact that our developed module was designed for use by VPNs we identified several other use-cases which can take advantage of our module. That is, encrypted file transfer over TLS or DTLS and packet forwarding (proxying) between two sockets, one being TLS or DTLS – enabled. In the encrypted file transfer case, we identified that our implemented Linux kernel module outperforms any available file transfer mechanism over TLS/DTLS.

Bibliography

[1] Cryptodev–linux module. [online], [cited 2015-12-18].
Retrieved from: http://cryptodev-linux.org/

[2] Gnu time. [online] Last modification: 1999-12-11, [cited 2015-12-18].
Retrieved from: https://www.gnu.org/software/time/

[3] HAProxy: The reliable, High performance TCP/HTTP Load Balancer. [online], [cited 2016-04-01].
Retrieved from: http://haproxy.org

[4] Introduction to CISCO IPsec Technology. [online], [cited 2015-12-18].
Retrieved from: http://www.cisco.com/c/en/us/td/docs/net_mgmt/vpn_solutions_center/2-0/ip_security/provisioning/guide/IPsecPG1.html

[5] LibreSSL. [online], [cited 2015-12-18].
Retrieved from: http://www.libressl.org/

[6] Libreswan VPN software. [online], [cited 2015-10-16].
Retrieved from: https://libreswan.org/

[7] OpenConnect VPN Client. [online], [cited 2015-07-15].
Retrieved from: http://www.infradead.org/openconnect/

[8] OpenConnect VPN Server. [online], [cited 2015-07-15].
Retrieved from: http://www.infradead.org/ocserv/

[9] OpenSSL. [online], [cited 2015-07-15].
Retrieved from: https://www.openssl.org/

[10] OpenStack Platform. [online], [cited 2015-10-16].
Retrieved from: https://www.openstack.org/

[11] OpenVPN – Open source VPN. [online], [cited 2015-07-15].
Retrieved from: https://openvpn.net/

[12] OpenVPN cryptographic layer.
Retrieved from: https://openvpn.net/index.php/open-source/documentation/security-overview.html

[13] perf: Linux profiling with performance counters. [online], [cited 2015-12-18].
Retrieved from: https://perf.wiki.kernel.org/index.php/Main_Page

[14] RFC 4106: The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP). [online], [cited 2016-04-01].
Retrieved from: https://tools.ietf.org/html/rfc5288

[15] RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2. [online], [cited 2015-10-16].
Retrieved from: https://tools.ietf.org/html/rfc5246

[16] RFC 5288: AES Galois Counter Mode (AES GCM) Cipher Suites for TLS. [online], [cited 2016-04-01].
Retrieved from: https://tools.ietf.org/html/rfc5288

[17] RFC 6347: Datagram Transport Layer Security Version 1.2. [online], [cited 2015-10-16].
Retrieved from: https://tools.ietf.org/html/rfc6347

[18] RFC 7540: Hypertext Transfer Protocol Version 2 (HTTP/2). [online], [cited 2016-04-01].
Retrieved from: https://tools.ietf.org/html/rfc7540

[19] The Docker Solution. [online], [cited 2015-10-16].
Retrieved from: https://www.docker.com/

[20] The Gnu Netcat project. [online] Last modification: 2013-06-07, [cited 2015-12-18].
Retrieved from: http://netcat.sourceforge.net/

[21] The Gnu Operating System and Free Software Movement. [online], [cited 2015-07-15].
Retrieved from: http://www.gnu.org/

[22] The GnuTLS Manual. [online], [cited 2015-10-16].
Retrieved from: http://www.gnutls.org/manual/gnutls.html

[23] The GnuTLS Transport Layer Security Library. [online], [cited 2015-10-16].
Retrieved from: http://www.gnutls.org/

[24] The Internet Engineering Task Force. [online], [cited 2015-07-15].
Retrieved from: https://www.ietf.org/

[25] The Linux Documentation: cgroups. [online], [cited 2015-10-16].
Retrieved from:
https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

[26] The Linux Documentation: Scatterlist Cryptographic API. [online], Last modification: 2015-12-28 [cited 2016-01-02].
Retrieved from:
https://www.kernel.org/doc/Documentation/crypto/api-intro.txt

[27] TLS 1.3 Draft: The Transport Layer Security (TLS) Protocol Version 1.3. [online], [cited 2016-04-01].
Retrieved from: https://tools.ietf.org/html/rfc7540

[28] Axbe, J.: Linux Kernel Mailing List: splice support #2. 2006-03-30. [online], [cited 2016-04-01].
Retrieved from: https://lkml.org/lkml/2006/3/30/130

[29] Branco, R. R.: Ltrace Internals. [online], [cited 2015-12-18].
Retrieved from:
https://www.kernel.org/doc/ols/2007/ols2007v1-pages-41-52.pdf

[30] Chen, R.: The hunt for a faster syscall trap. [online], [cited 2016-04-01].
Retrieved from:
https://blogs.msdn.microsoft.com/oldnewthing/20041215-00/?p=37003/

[31] Chimata, A. K.: Path of a packet in the Linux Kernel Stack. 2005-07-11. [online], [cited 2016-04-01].
Retrieved from:
http://www.hsnlab.hu/twiki/pub/Targyak/Mar11Cikkek/Network_stack.pdf

[32] Edge, J.: An API for user-space access to kernel cryptography. 2010-08-25. [online], [cited 2016-04-01].
Retrieved from: http://lwn.net/Articles/401548/

[33] Edge, J.: TLS in the Kernel. 2015-12-02. [online], [cited 2016-04-01].
Retrieved from: http://lwn.net/Articles/666509/

[34] Felter, W.; Ferreira, A.; Rajamony, R.; et al.: IBM Research Report: An Updated Performance Comparison of Virtual Machines and Linux Containers. [online], Last modification: 2014-07-21 [cited 2016-01-02].
Retrieved from: http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf

[35] Keromytis, A. D.; Wright, J. L.; de Raadt, T.: The Design of the OpenBSD Cryptographic Framework. [online], [cited 2016-01-02].
Retrieved from: http://www.openbsd.org/papers/ocf.pdf

[36] Krasnyansky, M.; Yevmenkin, M.: Universal TUN/TAP device driver. [online], Last modification: 2000, [cited 2015-07-15].
Retrieved from:
https://www.kernel.org/doc/Documentation/networking/tuntap.txt

[37] Leffler, S. J.: Cryptographic Device Support for FreeBSD. [online], [cited 2016-01-02].
Retrieved from: https://www.usenix.org/legacy/event/bsdcon03/tech/leffler_crypto/leffler_crypto.pdf

[38] Li, C.; Ding, C.; Shen, K.: Quantifying The Cost of Context Switch. [online], [cited 2016-04-01].
Retrieved from: http://www.cs.rochester.edu/u/cli/research/switch.pdf

[39] Mavrogiannopoulos, N.: OpenConnect Draft. [online], Last modification: May 2015, [cited 2015-07-15].
Retrieved from: https://github.com/openconnect/protocol/blob/master/draft-openconnect-01.txt

[40] Mavrogiannopoulos, N.: TLS in embedded systems. [online] Last modification: 2012-04-01, [cited 2015-12-18].
Retrieved from:
http://nmav.gnutls.org/2012/04/in-some-embedded-systems-space-may.html

[41] Mavrogiannopoulos, N.: TLS in embedded systems. [online], [cited 2016-04-01].
Retrieved from:
http://nmav.gnutls.org/2012/04/in-some-embedded-systems-space-may.html

[42] Mavrogiannopoulos, N.: Why do we need SSL VPNs today? [online], [cited 2016-04-01].
Retrieved from:
http://nmav.gnutls.org/2016/02/why-do-we-need-ssl-vpns-today.html

[43] Moellenkamp, J.: Less known Solaris Features: `kssl`. [online], [cited 2016-04-01].
Retrieved from: http://www.c0t0d0s0.org/archives/5575-Less-known-Solaris-Features-kssl.html

[44] Shue, C. A.; Gupta, M.; Myers, S. A.: IPSec: Performance Analysis and Enhancements. [online], [cited 2016-01-02].
Retrieved from: https://web.cs.wpi.edu/~cshue/research/icc07.pdf

[45] Steward, R.; Gurney, J.-M.; Long, S.: Optimizing TLS for High-Bandwidth Applications in FreeBSD. [online], [cited 2016-04-01].
Retrieved from: https://people.freebsd.org/~rrs/asiabsd_2015_tls.pdf

[46] Stoica, C.: Linux Kernel Patch: Add TLS record layer encryption module. 2014-07-29. [online], [cited 2016-04-01].
Retrieved from: http://www.mail-archive.com/linux-crypto%40vger.kernel.org/msg11666.html

[47] Struk, T.: Linux Kernel Patch: crypto: af_alg: add TLS type encryption. 2016-03-05. [online], [cited 2016-04-01].
Retrieved from: http://www.mail-archive.com/linux-crypto%40vger.kernel.org/msg17015.html

[48] Sutter, P.: Linux Kernel Mailing List: comparison of the AF_ALG interface with the /dev/crypto. 2011-08-30. [online], [cited 2016-04-01].
Retrieved from: http://www.mail-archive.com/linux-crypto%40vger.kernel.org/msg06010.html

[49] Watson, D.: Linux Kernel Patch: Crypto Kernel TLS Socket. 2015-11-23. [online], [cited 2016-04-01].
Retrieved from: http://www.mail-archive.com/linux-crypto%40vger.kernel.org/msg18080.html

Appendices

List of Appendices

Appendix A

Content of Attached DVD

Directory	Description
`af_ktls/`	C source codes of `AF_KTLS` kernel module online: https://github.com/fridex/af_ktls
`af_ktls-tool/`	C source codes of the tool used for benchmarks and verification with shell script to automate testing online: https://github.com/fridex/af_ktls-tool
`af_ktls-visualize/`	Python source codes of visualization tool for automated statistics in HTML or Gnuplot generation online: https://github.com/fridex/af_ktls-visualize
`af_ktls-window/`	testing application for `AF_KTLS` sliding window implementation online: https://github.com/fridex/af_ktls-window
`cryptodev-linux-dtls/`	C source codes of unfinished implementation in Cryptodev–linux
`bin/`	binaries built from source codes
`docker-vpns/`	shell scripts and `Dockerfile` to automatically set up testing environment for Libreswan, OpenConnect and OpenVPN online: https://github.com/fridex/docker-vpns
`html/i7-4600U-2.10GHz/`	results of benchmarks in HTML performed on CPU Intel i7–4600U
`html/i7-4900MQ-2.80GHz/`	results of benchmarks in HTML performed on CPU Intel i7–4900MQ
`latex/`	LaTeX sources of this thesis
`vpn-benchmarks/`	scripts and text files with results of VPN benchmarks based on Docker container in `docker-vpns/`
`projekt.pdf`	PDF version of this thesis built from template in `latex/`

Appendix B

Benchmarks of VPN Solutions Based on Ciphersuite Used

Cipher Type	Run #1	Run #2	Run #3	Average
AES128 GCM	770	817	800	795.67
AES256 GCM	813	787	747	782.33
AES128 SHA	477	473	489	479.67
DES CBC3 SHA	122	122	119	121.00

Figure B.1: Results of benchmarks for various cipher types when transferring 1GB file, using OpenConnect, results in Mbps

Cipher Type	Run #1	Run #2	Run #3	Average
RC2 CBC SHA1	198	194	193	195.00
DES EDE CBC SHA1	124	124	124	124.00
DES EDE3 CBC SHA1	123	121	123	122.33
DESX CBC SHA1	228	222	227	225.67
BF CBC SHA1	290	288	301	293.00
RC2 40 CBC SHA1	197	196	200	197.67
CAST5 CBC SHA1	282	268	266	272.00
RC2 64 CBC SHA1	194	191	192	192.33
AES 128 CBC SHA1	433	433	420	428.67
AES 192 CBC SHA1	415	420	411	415.33
AES 256 CBC SHA1	380	374	436	396.67
AES 192 CBC SHA1	409	417	434	420.00

Figure B.2: Results of benchmarks for various cipher types when transferring 1GB file, using OpenVPN, results in Mbps

Cipher Type	Run #1	Run #2	Run #3	Average
AES 128 SHA1	455	482	687	541.33
AES 128 SHA2 256	600	505	471	525.33
AES 128 MD5	562	659	566	595.67
AES 256 SHA1	674	608	505	595.67
AES 256 SHA2 256	456	574	391	473.67
AES 256 MD5	517	615	542	558.00
AES 128 CCM	330	328	282	313.33
AES 192 CCM	273	314	312	299.67
AES 256 CCM	257	305	296	286.00
AES 128 GCM	1440	1310	1230	1326.67
AES 192 GCM	1220	965	1250	1145.00
AES 256 GCM	1110	1060	1240	1136.67

Figure B.3: Results of benchmarks for various cipher types when transferring 1GB file, using Libreswan IPsec, results in Mbps

Cipher	Libreswan	OpenConnect	OpenVPN
RC2 CBC SHA1	–	–	195.00
RC2 40 CBC SHA1	–	–	197.67
RC2 64 CBC SHA1	–	–	192.33
CAST5 CBC SHA1	–	–	272.00
AES 128 CBC SHA1	–	–	428.67
AES 192 CBC SHA1	–	–	415.33
AES 256 CBC SHA1	–	–	396.67
3DES CBC	–	–	124.00
Bluefish CBC	–	–	293.00
3DES MD5	90.27	–	
AES SHA1	541.33	479.67	–
AES SHA2	525.33	–	–
AES 128 MD5	595.67	–	–
AES 256 SHA1	595.67	–	–
AES 256 SHA2	473.67	–	–
AES 256 MD5	558.00	–	–
AES 128 CCM	313.33	–	–
AES 192 CCM	299.67	–	–
AES 256 CCM	286.00	–	–
AES 128 GCM	1326.67	795.67	–
AES 192 GCM	1145.00	–	–
AES 256 GCM	1136.67	782.33	–

Figure B.4: Results of benchmarks for various cipher types when transferring 1GB file – average comparison, results in Mbps

Appendix C

Results of AF_KTLS Benchmarks

Payload	GnuTLS	AF_KTLS
1280	123.659	96.818
1400	134.796	104.527
4000	311.184	229.034
6000	390.075	293.407
9000	466.284	336.609
13000	562.823	398.756
16000	597.171	436.447

Figure C.1: Benchmarks of transmission from section 5.5.2, figure 5.1 – DTLS, results in MB/s

Payload	GnuTLS	AF_KTLS
1280	99.325	82.904
1400	103.893	83.514
4000	242.261	196.514
6000	325.226	255.397
9000	404.630	313.895
13000	487.797	365.553
16000	526.075	398.937

Figure C.2: Benchmarks of transmission from section 5.5.2 5.2 – TLS, results in MB/s

File size	user-send	sendfile(2)	mmap(2)	splice(2)
TCP	1210.955	1648.777	1168.306	1694.829
UDP	428.994	453.025	444.264	456.262

Figure C.3: Benchmarks of sending a file using TCP/UDP, based on comparison from section 5.5.2, results in MB/s

	user-send	sendfile(2)	mmap(2)	splice(2)
user-send	-	73.324%	103.479%	71.332%
sendfile(2)	136.380%	-	141.125%	97.283%
mmap(2)	96.638%	70.859%	-	68.934%
splice(2)	140.190%	102.793%	145.067%	-

Figure C.4: A percentage comparison of a file transmission benchmarks – TCP

	user-send	sendfile(2)	mmap(2)	splice(2)
user-send	-	94.695%	96.563%	94.024%
sendfile(2)	105.602%	-	101.972%	99.291%
mmap(2)	103.559%	98.066%	-	97.370%
splice(2)	106.356%	100.715%	102.701%	-

Figure C.5: A percentage comparison of a file transmission benchmarks – UDP

File size	user-send	sendfile(2)	mmap(2)
TLS	563.054	628.780	604.504
DTLS	291.062	322.825	314.801

Figure C.6: Benchmarks of sending a file using TLS/DTLS, based on comparison from section 5.5.2, results in MB/s

	user-send	sendfile(2)	mmap(2)
user-send	-	89.547%	93.143%
sendfile(2)	111.673%	-	104.016%
mmap(2)	107.362%	96.139%	-

Figure C.7: A percentage comparison of a file transmission benchmarks – TLS

	user-send	sendfile(2)	mmap(2)
user-send	-	90.161%	92.459%
sendfile(2)	110.913%	-	102.549%
mmap(2)	108.156%	97.514%	-

Figure C.8: A percentage comparison of a file transmission benchmarks – DTLS

Payload	recv()/gnutls_record_send()	splice(2)
1280	153.914	107.360
1400	166.635	116.248
4000	413.391	279.442
6000	541.561	364.345
9000	665.361	458.023
13000	825.372	550.724
16000	895.004	605.048

Figure C.9: Benchmarks of OpenConnect optimized scenario from section 5.5.2, figure 5.8 – TLS, results in MB/s

Payload	recv()/gnutls_record_send()	splice(2)
1280	110.828	91.924
1400	123.404	97.751
4000	310.469	245.362
6000	426.563	323.098
9000	545.686	412.173
13000	699.848	514.279
16000	773.900	568.378

Figure C.10: Benchmarks of OpenConnect optimmized scenario from section 5.5.2, figure 5.7 – DTLS, results in MB/s

Appendix D

Version of the Used Software

Software	Version
Operating System	Fedora 23 Workstation
Linux Kernel	4.3.3-303.fc23.x86_64
iperf	2.0.8-3
GnuTLS	3.4.8-1
OpenSSL	1.0.2g-1
Docker	1.10.0
OpenConnect	7.06-1
ocserv	0.9.0-2
OpenVPN	2.3.10-2
Libreswan	3.16-1

Figure D.1: Listing of software versions used in the thesis