BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

ACCELERATION OF DTLS PROTOCOL IN LINUX KERNEL

AKCELERACE PROTOKOLU DTLS V JÁDŘE SYSTÉMU LINUX

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **FRANTIŠEK KRENŽELOK**
AUTOR PRÁCE

SUPERVISOR **Ing. MATĚJ GRÉGR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

Abstract

This thesis offers a overview of the DTLS1.3 protocol, focusing on critical features pertinent to both userspace applications and kernel-level integration. Key outcomes include the implementation of DTLS1.3 within GnuTLS and the extension of this protocol to the Linux kernel's TLS module. Additionally, the thesis evaluates the utility of this kernel module in conjunction with emerging technologies such as eBPF and projects like ktls-util.

Abstrakt

Tato práce poskytuje náhled do protokolu DTLS1.3, se zaměřením na klíčovou funkcionalitu důležitou pro implementaci v rámci kryptografických knihoven a integraci na úrovni systémového jádra. Hlavní výsledky zahrnují implementaci DTLS1.3 v rámci knihovny GnuTLS a rozšíření tohoto protokolu do linuxového jadra v rámci existujicího modulu TLS. Práce nadále hodnotí užitečnost tohoto systémového modulu ve spojení s nově vznikajícími technologiemi, jako je eBPF, a projekty jako je ktls-util.

Keywords

TLS, DTLS, Linux kernel, kTLS, kDTLS, acceleration, security, optimization, GnuTLS, OpenConnect, DTLS1.3

Klíčová slova

TLS, DTLS, Linux kernel, kTLS, kDTLS, akcelerace, bezpečnost, optimalizace, GnuTLS, OpenConnect, DTLS1.3

Reference

KRENŽELOK, František. *Acceleration of DTLS Protocol in Linux Kernel.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Matěj Grégr, Ph.D.

Acceleration of DTLS Protocol in Linux Kernel

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Matěj Grégr Ph.D. The supplementary information was provided by Mr. Daiki Ueno Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
František Krenželok
May 9, 2024

Contents

List of Figures

Chapter 1

Introduction

(D)TLS, or Datagram Transport Layer Security, is a crucial cryptographic and communication protocol designed to secure computer networks. Its primary functions are to authenticate the parties engaged in communication, establish a secure channel through a handshake process, and manage the encryption and decryption of data being exchanged.

Unlike TLS (Transport Layer Security), which is geared towards stream-based applications like file transfer and messaging, DTLS is specifically tailored for datagram-based applications. This specialization makes it particularly suitable for applications such as video and audio streaming or gaming, where data packets are sent independently.

Currently, DTLS is deployed in user-level applications only, not in the Linux kernel. This is in contrast to TLS, which does have a kernel-based implementation known as kernel TLS (kTLS). Bringing DTLS into the Linux kernel offers potential advantages. It can lead to quicker encryption and decryption processes, these tasks can be additionally offloaded from the CPU to specialized network interface cards (NICs). This offloading is especially beneficial in reducing CPU load, an advantage that cloud-based streaming service providers could find particularly valuable.

In terms of its structure, the (D)TLS protocol is divided into two main parts: the handshake and the record. In the context of integrating DTLS into the Linux kernel, only the record component will be incorporated. The record is responsible for the actual encryption and decryption of application data. The kernel implementation will depend on user-space applications to manage the handshake phase and to supply the necessary keying material. This division of responsibilities ensures that the kernel handles the most performance-critical tasks while leaving the initial setup and authentication processes to the userspace, thereby maintaining the security and efficiency of the system.

The Kernel DTLS (kDTLS) protocol, initially a component of the TLS kernel module, has not yet been integrated into the mainstream Linux kernel. This thesis investigates the feasibility of incorporating kDTLS into the existing kTLS module, setting a foundation for future developments and identifying potential use cases. The study aims to determine how kDTLS can enhance system performance, specifically through improvements in speed and latency, and more efficient resource utilization. These enhancements include reducing power consumption, minimizing context switches, and decreasing memory duplication across userspace and kernel space, thereby optimizing overall system operations.

Terminology

The following terms, abbreviation, acronyms... are frequently used throughout the work.

Abbreviations and Acronyms

(D)TLS denotes common feature of protocols TLS and DTLS

k(D)TLS denotes the common feature of both protocol in the Linux kernel module

sequence number used interchangeably with "record number".

Technical Terms

kernel core component of an operating system, managing communication between software and hardware. It handles system resources, manages memory and device operations. The kernel operates at a high privilege level.

userspace part of the operating system where user applications run, separate from the kernel space. It interacts with the kernel through system calls. The isolation ensures system stability. The userspace operates in unprivileged mode.

UDP (User Datagram Protocol) is a datagram-based transport layer communication protocol. It is unreliable, meaning it does not guarantee the delivery and reordering of packets, nor does it check for duplicates of packets. It also has some advantages, such as low latency and high throughput.

TCP (Transmission Control Protocol) is a connection-oriented transport layer communication protocol. It provides reliable, ordered, and error-checked delivery of a stream of data between applications running on hosts communicating over an IP network. Unlike UDP, TCP ensures that data packets are delivered in the correct order and retries sending packets that are lost during transmission. This reliability comes at the cost of higher latency and lower throughput compared to UDP.

QUIC a transport layer protocol aimed to replace TLS over TCP for Web usage by increasing performance and achieving lower latency by establishing several multiplexed UDP connections per session.

ciphertext Ciphertext refers to the result of cryptographic transformation of plaintext using an encryption algorithm. This transformation ensures that the data is unreadable to unauthorized parties. Ciphertext can only be understood and reverted to plaintext if the decryption key is known.

plaintext Plaintext is any readable data used as input for encryption. It represents the original message or data before any cryptographic operations are applied.

Units of measurement

Mb/s (Megabit per second) — also [Mbit/s], used as a measure for network bandwidth, i.e., how many million bits per second can travel through the given media.

Chapter 2

DTLS

(D)TLS is a cryptographic/communication protocol handling computer network security. The protocol provides means to authenticate the communicating parties with certificates, establish a secure connection via a (D)TLS handshake, and encrypting the communication. It consists of two main parts, handshake and record layers.

DTLS is ideal for applications that don't need reliable transport but require low delays. It is being deployed in many types of applications such as VoIP (Voice over IP), WebRTC (Web Real-Time Communication), IOT (Internet of things), Online gaming and Cloud streaming services.

2.1 Overview of (D)TLS

2.1.1 Handshake layer

The handshake is used to establish the connection by:

a) authenticating communicating parties using certificates.

b) negotiating all the security parameters, such as highest supported (D)TLS Version, ciphersuite, elyptic-curve groups. [7, Section 4.1.1]

c) deriving a shared master secret by means of asymmetric cryptography to be later used by the record layer 2.1.2

2.1.2 Record layer

The record handles encryption of the application data by means of symmetric cryptography with keys derived by handshake. Most data transfer and communication happen with occasional interruptions by Alerts, or post-handshake messages, after which the execution is sent back to the Handshake 2.1.1.

2.1.3 Alert layer

The alert layer is used to handle alert messages received from peer.

2.1.4 (D)TLS Session

A (D)TLS session describes an established secure connection between a client and a server. It is initiated using the (D)TLS handshake 2.1.1, after which data is transmitted using

(D)TLS record 2.1.2. Thanks to session resumption, a session can span multiple underlying network connections. This feature allows for a quicker resumption of the session compared to establishing a new one by leveraging a Pre-shared Key (PSK) 4 exchanged during the previous session.

2.1.5 (D)TLS Epoch

In a Transport Layer Security (TLS) session, data transmission is organized into phases called 'epochs'. Each epoch corresponds to a specific phase in the TLS handshake or data transfer process, and is characterized by a unique set of cryptographic keys. These keys are used to ensure the confidentiality and integrity of the data as it is transmitted over a network. Epochs in TLS help manage the transition between different security contexts during a session, allowing for changes in encryption and authentication methods as the session progresses. Initially, messages may start unencrypted but become increasingly secure through subsequent epochs as new keys are negotiated and applied. Below is a detailed table that outlines specific epochs and their associated cryptographic functions in a TLS session, refer to Table 2.1 below.

Epoch	Description
0	Unencrypted DTLS messages including ClientHello, ServerHello, and HelloRetryRequest.
1	Used for messages encrypted with keys derived from the `client_early_traffic_secret`. Omitted if no early data is provided by the client.
2	Protects messages using keys from `[sender]_handshake_traffic_secret`, covering initial handshake messages like EncryptedExtensions and CertificateVerify. Does not include post-handshake messages.
3	Messages are encrypted with keys from the initial `[sender]_application_traffic_secret_0`, including certain post-handshake messages.
$\geq 4 \ldots 2^{64} - 1$	Messages are secured with keys from `[sender]_application_traffic_secret_N`, where N>0.

Table 2.1: Cryptographic Functions of TLS Epochs

2.2 DTLS protocol

DTLS is implemented for applications running on datagram-based transport protocols such as UDP and STCP. As those transports are unreliable, 1. There is a heavy emphasis on diverging from the TLS protocol as little as possible, yet DTLS protocol needs to provide means to:

a) make the Handshake reliable with Acknowledge (ACK) messages 2.2.1.

b) protect against replay attack 2.3.2 using sliding window and sequence number.

c) keep the session (connection) alive after alternation of endpoint addresses by using Connection ID extension 2.3.3

2.2.1 ACK message (a)

The ACK message in DTLS1.3 indicates which handshake records have been received and processed, utilizing a separate content type (code point 26) to avoid inclusion in the handshake transcript. It consists of a record numbers list, marking received and either processed or buffered handshake messages in sequence. This mechanism is critical to prevent deadlocks by ensuring only relevant records are acknowledged. During a handshake, ACKs are confined to the current flight, reflecting DTLS's handshake sequential structure. Post-handshake, ACKs may cover multiple flights, offering flexibility in managing extended communication scenarios.

2.3 DTLS version 1.3

DTLS1.3 comes with many improvements, many of which were adapted from the QUIC protocol 1. This section describes only the improvements to the record part of the protocol. For a comprehensive overview of the enhancements of both TLS1.3 and DTLS1.3, see. [9].

2.3.1 Demultiplexing Unified Header

The unified header is variable in length. The content type DTLS version are hidden in the ciphertext; The DTLS1.3 version is instead distinguished by the first byte of the header as an identifier, as well as a header structure descriptor; see Figure 2.1. The most significant 3 bits are a fixed value of 001; they indicate the presence of DTLS1.3 unified header. The following bits represent the **presence** of the connection ID, the **8 or 16 bit** size sequence number, and the **presence** of a 16 bit length value in this order. The last 2 bits represent the least significant bits of the **epoch number**.

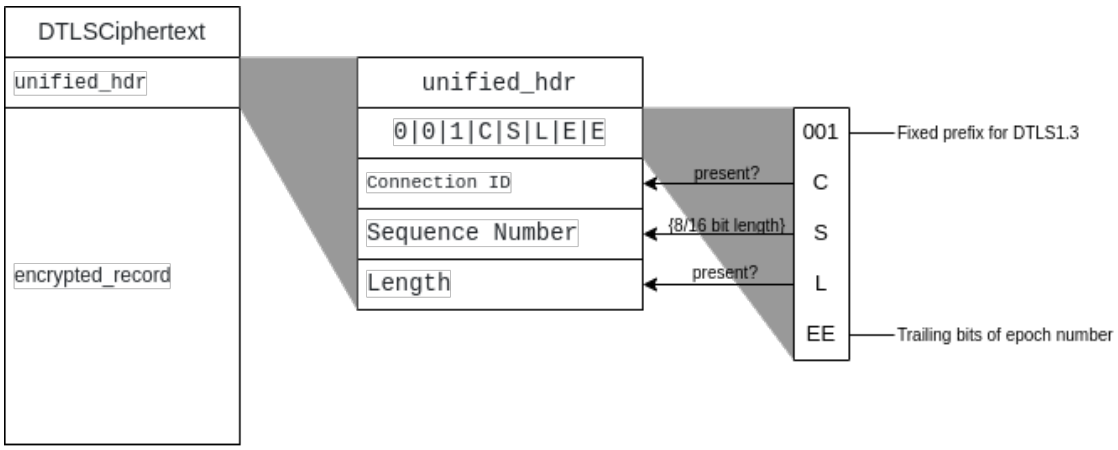


Figure 2.1: DTLS Ciphertext Unified Header

Epoch

The **Epoch** is resolved by comparing the 2 low bits of the current epoch stored locally with the on-wire bits included in the `unified_hdr`; see Figure 2.1. If they do not match the most recent epoch, they are compared with the previous epoch, if it still does not match it is discarded.

Sequence number

In Datagram Transport Layer Security (DTLS), the *sequence number* is used to protect against replay attacks and ensure data integrity. Each DTLS record contains a sequence number, which is incremented for each transmitted record. This mechanism helps in verifying the order and uniqueness of the received records, enhancing security in unreliable, connectionless communication environments.

AEAD (Authenticated Encryption with Associated Data) is a method that ensures both the confidentiality and integrity of data. It encrypts data for privacy (*encryption*) and authenticates both the encrypted and additional associated data for integrity and authenticity (*authentication*)

AES (Advanced Encryption Standard) is a widely-used symmetric encryption algorithm, notable for its efficiency and security. It encrypts data in fixed-size blocks using symmetric keys. AES can operate in various modes, in this context it will be **Electronic Codebook (ECB)** 2.3.1

ChaCha20-Poly1305 - This algorithm merges the ChaCha20 stream cipher and Poly1305 MAC, providing efficient encryption and data integrity, suitable for varied data lengths and secure communications.

ECB (Electronic Codebook) — This is one of the most basic encryption modes. In ECB, each plaintext block is encrypted separately using the same cryptographic key. Compared to other modes, such as CBC (Cipher Block Chaining), which incorporates the previous block's ciphertext as an initialization vector (IV) to enhance security, ECB is considered less secure. CBC mode requires possession of both the current and preceding ciphertext block, as the previous block is XOR-ed with the plaintext, thereby increasing message obfuscation and complicating cryptanalysis. However, CBC and any other chaining modes of operation are not ideal for UDP connections, where messages might arrive out of sequence or be missing altogether.

At the start of a new epoch, the epoch's master secret is used to generate the `sn_key` (sequence number key). The sequence number of a message is deciphered by encrypting the first 16 bytes of the message ciphertext with the `sn_key`, producing a 16-byte mask. The first one or two bytes (depending on the number of sequence bytes in the header 2.3.1) of this mask are XOR-ed with the header's sequence bytes to extract one or two bytes of the sequence number. These bytes are then compared to the least significant bits of the next expected sequence number, incrementally adjusting the compared sequence number until a match is found.

> [8] When the AEAD 2.3.1 is based on AES, then the mask is generated by computing AES-ECB 2.3.1 on the first 16 bytes of the ciphertext:

```
Mask = AES-ECB(sn_key, Ciphertext[0..15])
```

When the AEAD is based on ChaCha20, then the mask is generated by treating the first 4 bytes of the ciphertext as the block counter and the next 12 bytes as the nonce, passing them to the ChaCha20 block function (Section 2.3 of [CHACHA]):

```
Mask = ChaCha20(sn_key, Ciphertext[0..3], Ciphertext[4..15])
```

The sn_key is computed as follows:

```
sn_key = HKDF-Expand-Label(Secret, "sn", "", key_length)
```

2.3.2 Replay protection

Replay attack

A replay attack is a specific type of man-in-the-middle attack in which an attacker intercepts legitimate network transmissions. During this process, the attacker may capture a single packet or a series of packets, which are either stored for future use, such as decryption when the employed ciphers become obsolete, or retransmitted to the intended recipient. The purpose of the latter is to deceive the receiver into accepting the retransmitted packet as original and legitimate.

For example, consider a scenario involving a bank transaction where an attacker, positioned to intercept the data, records a transaction where they are designated as the beneficiary. The attacker could then repeatedly resend this transaction, potentially initiating multiple unauthorized transfers to benefit themselves.

Another illustrative case of a replay attack occurs when an attacker captures all session packets from a communication session. The attacker's goal is to later retransmit these packets to impersonate a legitimate user, thereby gaining unauthorized network access. This type of attack takes advantage of security systems that fail to differentiate between original and retransmitted data, relying on capturing authentication details or session tokens that are reused in the fraudulent session to deceive security protocols.

To combat replay attacks, various preventive measures can be employed. These include the use of time-stamping to verify the timeliness of each packet, challenge-response mechanisms that require participants to prove their identity in real-time, and finally sequence enumeration combined with a sliding window technique to check for duplicate packets, which is the case for DTLS.

Sliding window

A bit-mask keeping track of sequence numbers that have been received. The size of the sliding window is for the implementation to decide, but at least a 64bit mask is recommended. The right-most cell represents the highest sequence number received. When a new packet arrives, its sequence number is first decrypted and then accepted or discarded based on the sliding window. If it is smaller than the leftmost value, it is discarded; if it fits into the window and the cell indicates that it was not yet received, or if it is higher than the right-most cell value, it is accepted and marked in the bit-mask as received.

Note that if a higher than right-most value is received, the sliding window is shifted to accommodate the newly received packet. It is unlikely that such a packet would be received for a large enough window, e.g., 64bit, as most of the implementations shift the window periodically. But on the rare occasion when this happens, it is better to potentially discard past messages than to lose newer ones.

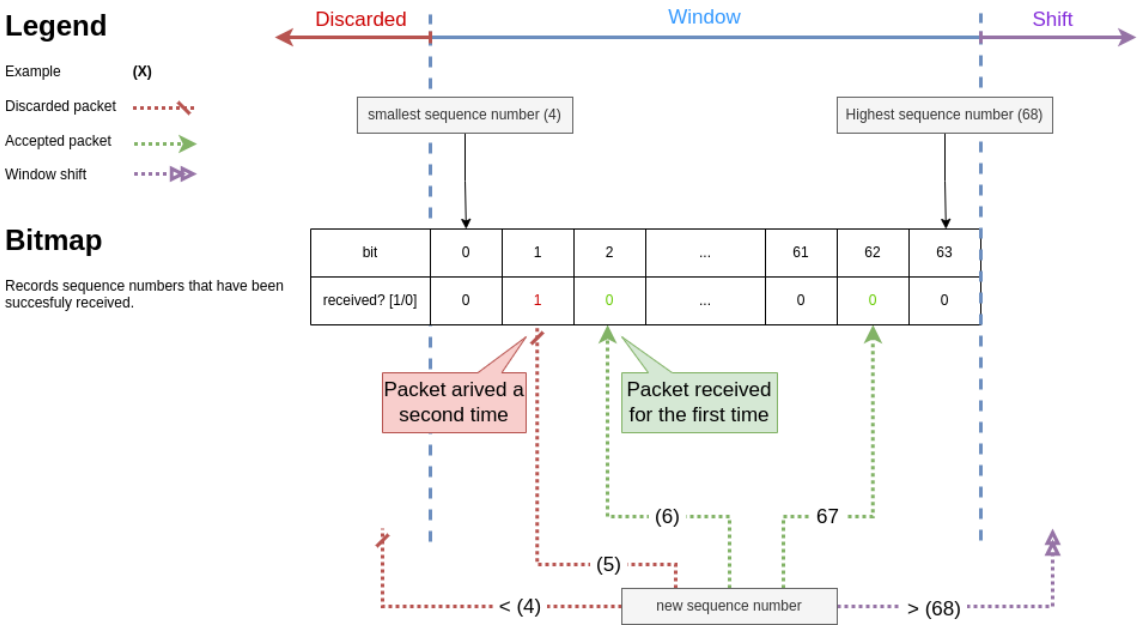


Figure 2.2: Sliding window

2.3.3 Connection ID

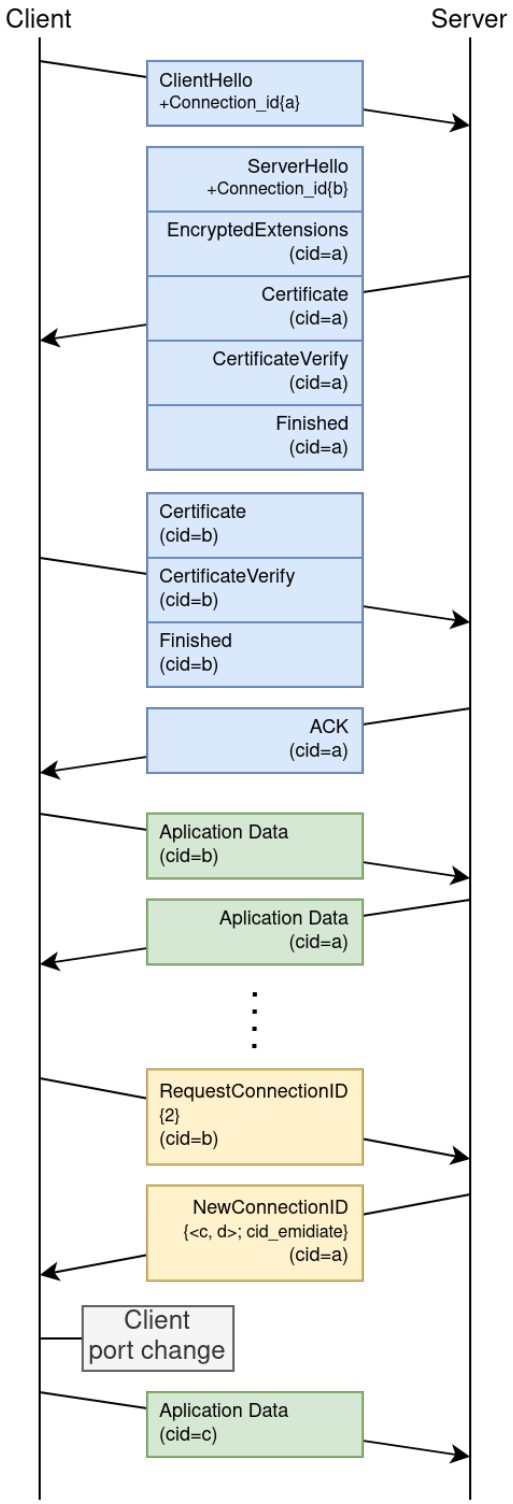


Figure 2.3: Connection ID example

UDP connection is usually identified by the 5-tuple (source IP address, source port, destination IP address, destination port, transport protocol); this falls short on occasions such as NAT rebinding, in which case the 5-tuple would be altered, which will inevitably results in connection failure. In DTLS1.3 this problem is solved by negotiating connection IDs to identify the session. It is implemented as an optional DTLS extension that holds the cid the sender wishes the endpoint to use. The zero-length cid in the extension indicates that the sender is willing to use the endpoint's Connection ID when sending records, but does not wish to receive its cid in incoming messages. The negotiation begins with the client attaching the Connection cid extension to the `Client_Hello` message. Upon receiving the message, if the cid size is nonzero, the server will use this value in `unified_header` for outgoing messages after the handshake is complete. If the server does not wish to receive a cid in messages from the client, it will send a zero-size cid in the extension; if it wishes otherwise, it will send a cid of its choice. DTLS1.3 supports renegotiating the CID with a **NewConnectionID** message that contains a list of CID values to use and a usage field indicating the way to use the new CIDs; that is, as a **spare** for when the connection parameters change or **immediate**, telling the receiver to use one of the provided CIDs for the following communication. There is also a possibility of requesting that the other side send the `NewConnectionID` with spare CIDs by sending **RequestConnectionId**. This is done in anticipation of changes in the 5-tuple values.

Implementation must drop the packet if CID is received, while neither side expects it.

Chapter 3

GnuTLS

A cryptography library, distributed under the GNU Lesser General Public License version 2.1[1], facilitates secure communication by implementing SSL, TLS, and DTLS protocols alongside related technologies[2]. In this work, the library is utilized as the userspace implementation for the DTLS1.3 handshake, from which record keys are derived. Additionally, the library supports kTLS[3], greatly simplifying the integration of kDTLS. This is due to the existing functionality for provisioning the keyring material to the kernel and having custom pull and push function aligning with the kTLS API, for the minor modification needed to also support the kDTLS see. sections 3.2 and 5.3.
At the time of writing of this work, the library doesn't support DTLS1.3; therefore, a implementation was necessary to facilitate the development and testing of kDTLS.

3.1 DTLS1.3 implementation

This section describes the changes introduced by the provisioned path. (see. attached CD A.1 for original or Merge Request[4] for latest development)

3.1.1 DTLS1.3 specific functions

- `uint16_t _dtls13_resolve_epoch();`
  Resolves the epoch number 2.1 for a DTLS1.3 session. It uses the session state and 2 epoch bits 2.3.1 from the unified header. The 2 bits represent the lowest order bits of the epoch and are matched against the currently active session epoch to determine the exact epoch number, which is crucial for session state management.

- `int _dtls13_resolve_seq_num();`
  Determines the sequence number for a given epoch after deciphering the sequence number bits as described in subsection 2.3.1.

- `int _dtls13_encrypt_seq_num();`
  Encrypts the sequence number for transmission 2.3.1. It takes into account the size of the sequence number and the ciphertext, using the specified record parameters. It is also used by `_dtls13_resolve_seq_num()` for sequence number decryption.

[1] https://www.gnu.org/licenses/old-licenses/lgpl-2.1.html
[2] https://www.gnutls.org/
[3] https://fedoraproject.org/wiki/Changes/KTLSSupportForGnuTLS
[4] https://gitlab.com/gnutls/gnutls/-/merge_requests/1667

- `inline unsigned _dtls13_calculate_header_length();`
  Calculates the length of the DTLS1.3 protocol unified header. This inline function streamlines the processing of the header based on the header tag.

- `int gnutls_dtls13_recv_ack();`
  Handles the reception of an acknowledgment (ACK) messages 2.2.1 in a DTLS1.3 session. This function is used for confirming the successful receipt of handshake messages, ensuring reliable data transfer during the DTLS handshake.
  Commonly used by the client to receive acknowledgment of successful delivery of its final handshake messages

- `int gnutls_dtls13_send_ack();`
  Manages the sending of an ACK messages 2.2.1 in a DTLS1.3 session. It ensures that the other party in the session is informed of the successful reception of messages, maintaining protocol integrity.
  Commonly send by the server to acknowledge the final round of handshake messages from the client

3.1.2 Handshake/Negotiation modifications

- Use the new TLS1.3 Hello **Retry** Request message instead of the old DTLS1.2 Hello **Verify** Request for DTLS1.3.

- Disable/Enable the appropriate TLS extensions.

- exclude the fragmentation information for the PSK4 and *client handshake finished*[5] messages digest.

- add The ACK 3.1.1 messages to the handshake state machine for both the client and server.

Encryption/Decryption

- The DTLS1.3 unified header 2.3.1 containing the **non-encrypted** sequence number is now being used as the associated data part of the AEAD 2.3.1, for encrypting the plaintext. This means that in order to decipher the plaintext, the record number must be resolved first.

3.2 GnuTLS kDTLS

3.2.1 Enabling k(D)TLS

- Load the kernel module 5.3.2

- kTLS is not yet in most distributions by default. It can be enabled by adding `ktls = true` in the `[global]` section[6] of the GnuTLS configuration file[7]

[5] https://tls13.xargs.org/#client-handshake-finished
[6] https://www.gnutls.org/manual/html_node/System_002dwide-configuration-of-the-library.html
[7] https://fedoraproject.org/wiki/Changes/KTLSSupportForGnuTLS#How_To_Test

3.2.2 Implementation

- Extended the `gnutls_record_get_state()` function with a new API function `gnutls_record_get_state_sn()` to also retrieve sequence number key alongside other cryptographic primitives such as Master Secret, IV, which are passed to be to the kernel via the `crypto_info` structure.
- The setsockopt now check the underlying transport protocol, that is TLS or DTLS and uses `TCP_ULP` and `UDP_ULP` respectively.

Chapter 4

OpenconnectVPN

OpenConnect VPN 4 is an SSL VPN client initially created to support Cisco's AnyConnect SSL VPN. It has since evolved to support protocols beyond Cisco's original SSL VPN specification. OpenConnect is open-source and has been designed to be easy to use and configure. It has been used in the past for very similar purposes by Fridolín Pokorný and his work "Linux VPN Performance and Optimization" [6]

In the scope of this work, the library is used for performance testing of the DTLS1.3 implementation.

VPN Virtual private network, used for accessing Local are network by creating an encrypted channel between VPN client and server. Nowadays, used also for privacy by serving as a proxy.

ocserv A openconnect's VPN server utility.

openconnect A openconnect's VPN client utility

PSK Pre-Shared Key, used by (D)TLS to establish connection without key-exchange, thus increasing security and potentially improving on session set-up time as only one round trip is necessary.

- **In band** — The PSK is exchanged during an already established (D)TLS connection for future connections, i.e., session resumption.
- **Out of band** — The PSK is distributed beforehand by any secure means. It is then used during the initial (D)TLS session handshake.

4.0.1 Connection establishment

The connection is first established using TLS (TCP)[1] to perform authentication of the connecting user, for instance by user:password pair. Once authenticated, the ocserv creates a PSK 4 and shares it for the negotiation of the actual application connection using DTLS.

4.0.2 Implementation

- The DTLS1.3 doesn't use `connection ID` field of a handshake message which the ocserv uses for sharing application specific ID, this was anticipated with the introduction

[1] https://ocserv.openconnect-vpn.net/technical.html

of TLS1.3 so the ocserv also supports receiving it via a custom TLS extension with the ID of `48018`, The TLS extension was added via the `gnutls_session_ext_register()`[2] which allows adding extension dynamically to the GnuTLS session as a part of openconnect implementation.

- The ocservs custom transort push and pull functions which are called after the encryption and decryption respectively had to be dissable as they dissalow the use of k(D)TLS.
- ocserve obsoleted the non zero copy version of the `dtls_push` so it was updated and the zero-copy version which directly conflicts with the kDTLS implementation by using different system calls was dissabled.

[2] https://www.gnutls.org/manual/gnutls.html#TLS-Hello-Extension-Handling

Chapter 5

DTLS1.3 kernel module implementation

Kernel Module is an object file containing code that can extend the functionality of the operating system kernel without the need to reboot the system. These modules can be dynamically loaded and unloaded at run-time, allowing for flexible modification of the kernel's capabilities as needed. Kernel modules are particularly useful for adding support for new hardware (like device drivers), file systems, or even new networking protocols.

5.1 Kernel TLS

kTLS[1] is a Linux kernel implementation of the TLS record 2.1.2. It relies on a user-space application to perform the TLS handshake, that is, establish the connection and derive key material, which is then passed to the kernel using `setsockopt()`. The application data is then passed to the kernel unencrypted, via a `sendmsg()` where is it encrypted, and handed over to the transport layer for sending. For the receiving part, the data is read from the socket by the kernel, decrypted, and then received in the userspace via recvmsg() call. The actual encryption/decryption can be performed either by the CPU itself or, as a means of offloading the former, by a peripheral such as special network interface cards.

5.1.1 kTLS sendfile()

When encryption or decryption is performed, the data have to go to userspace which increases memory consumption and CPU usage. You can see this in the picture 5.1a. First the data is read from the disk to a user space buffer, and then it is encrypted and saved to yet another buffer from which it is then send to a socket. kTLS overcomes this with `sendfile` and `splice` functions that can be used to send data directly from the disk through the kTLS module to a socket without ever entering the userspace. Thus, saving us from 2 content switches and 2 data copies. We can see this in the picture 5.1b. The GnuTLS negotiates a connection and provides all the cryptographic keyring material needed for encryption to the kTLS. Data goes directly from Disk to kTLS where it is encrypted and sent to a socket.

[1] https://www.kernel.org/doc/html/latest/networking/tls.html

(a) Non kTLS (b) kTLS

Figure 5.1: kTLS sendfile() scheme

5.1.2 Implementation

kTLS uses the function pointer dispatch to associate the system socket calls `sendmsg()` and `sendpage()` with their corresponding custom versions prefixed with `tls_`. The custom software versions, `tls_sw_sendmsg()` and `tls_sw_sendpage()`, are in charge of retrieving the data to be sent either from the userspace or the kernel, applying a BPF evaluation 8.1.2, encrypting the data, and both employ the `tcp_sendpage` function to transfer the encrypted data to the transport layer of the networking stack.

5.2 kDTLS

This is not a first attempt of implementing DTLS into the kernel, a thesis focused on enhancing the VPN connections using the kernel crypto was conducted by Fridolín Pokorný [6] resulted in the creation of the TLS kernel module which the DTLS support that didn't make it into the mainline kernel. Since then, the Linux kTLS module has undergone significant evolution. The current work represents a new attempt to implement DTLS in the kTLS module.

5.2.1 Differences between TCP and UDP in regard to kTLS

Sending

There are no notable dissimilarities as the underlying transport protocol handles the specifics of each of the protocols.

Receiving

As opposed to TCP, UDP does not guarantee reliable data delivery nor packet order in which the packets arrive. This eliminates the need for packet buffering, as the packets can be independently both decrypted and provided to the userspace.

5.3 Implementation details

The patch modifies the following Linux kernel files

- Header files `tls.h` and `uapi/linux/tls.h`.

- Source files under the `net/tls` directory.

The protocol is implemented as part of the existing kernel TLS module(tls.ko) as most of the cryptography related code is the same, the changes are mostly in the handling of the underlying transport protocol.

As there is no `UDP_ULP` (UDP Upper Layer Protocol) equivalent to `TCP_ULP` (used by kTLS) in the kernel, a patch [1] from a series of patches designed to implement the QUIC protocol within the kernel was applied.

Although the UDP is connectionless meaning that it does not bind the socket to one connection with specific destination using the `connect()` system function as is the usual case for TCP, It is still possible to use the `connect()` which comes in handy in this situation as all the necessary information for the transmission is set directly to the socket, so there is no need to resolve it while manipulating the application data in the kernel. To support a connection-less kDTLS socket, a lookup table of connections indexed by routing data and their corresponding cryptographic primitives would be required, thus introducing additional overhead.

5.3.1 DTLS Sliding window

To monitor the sequence numbers of received packets 2.3.2, a circular buffer[2], implemented as a binary ring array 5.2, was employed. This structure is particularly memory-efficient since the memory it occupies is exactly equivalent to the size of the sliding window 2.3.2, measured in bits. In terms of performance optimization, the buffer uses a pointer to monitor segments that become invalid when the buffer rotates 5.5. Should an attempt be made to access these invalidated segments, the system automatically clears and re-validates the entire affected section 5.4. If a message with a sequence number exceeding the window size is received, the ring is shifted to accommodate the change. You can see this in the figure 5.6.

To monitor the sequence numbers of received packets, a circular buffer[3], implemented as a binary ring array, is employed (see Figure 5.2). This structure is particularly memory-efficient since the memory it occupies is directly proportional to the size of the sliding window (discussed in Subsection 2.3.2), and is measured in bits.

In terms of performance optimization, the buffer utilizes a pointer to track segments that become invalid as the buffer rotates (illustrated in Figure 5.5). Should there be an attempt to access these invalidated segments, the system automatically clears and re-validates the entire affected section, as shown in Figure 5.4.

Furthermore, if a message with a sequence number that exceeds the current window size is received, the buffer is dynamically shifted to accommodate the new sequence number, ensuring continuous data integrity and flow. This adjustment process is depicted in Figure 5.6. The actual implementation may rotate the window more than is shown in the figure to account for the likelihood of receiving even higher sequence numbers in subsequent messages.

The sliding window structure 5.3, situated within the `tls_sw_context_rx`, employs the ring structure 5.2. It aligns the most recently received sequence number(`start_sn`) with the initial element of the bit ring.

[2] https://en.wikipedia.org/wiki/Circular_buffer
[3] https://en.wikipedia.org/wiki/Circular_buffer

```
typedef struct bit_ring_s {
    u64 size; //bit size
    u64 start; // position of the first bit
    u64 inactive; // position after whcich data is not valid
    bit_ring_segment_t *data;
} bit_ring_t;
```

Figure 5.2: Structure of the circular buffer

```
struct dtls_sliding_window {
    u64 window_size;
    u64 start_sn;
    bit_ring_t ring;
};
```

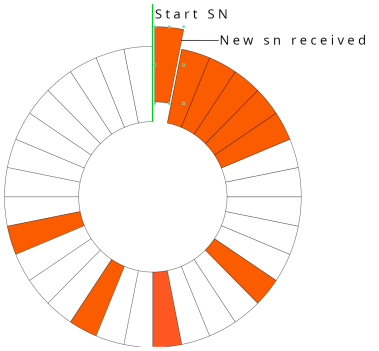
Figure 5.3: Structure of the DTLS sliding window

With the proposed changes, when a sequence number not complying to the sliding window is received, the `recvmsg()` 5.3.3 system call return `EAGAIN` telling the caller to call the `recvmsg()` function again.

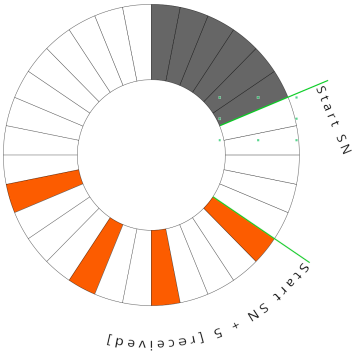
(a) Initial state

(b) Messages {7, 8, 11, 12, . . . } received

Figure 5.4: Bit-ring

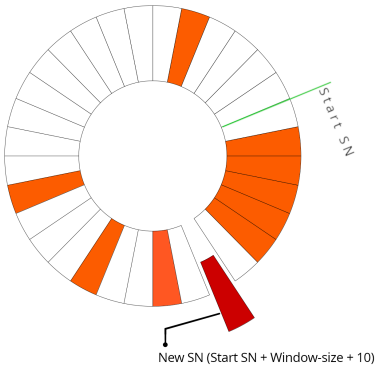


(a) continuous batch

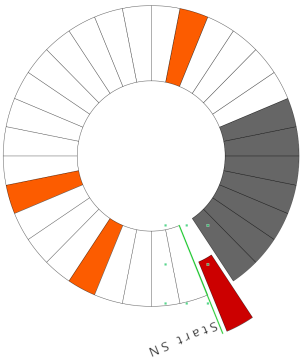


(b) Shift on continuous batch

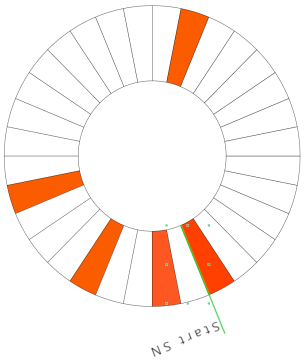
Figure 5.5: Bit-ring shift on continuous batch



(a) SN exceeding window size



(b) Shifting



(c) Result

Figure 5.6: Bit-ring shift on exceeded window size

(a) Access to inactive block

(b) Inactive block restored

Figure 5.7: Bit-ring restore inactive block

5.3.2 Setup

Enabling kDTLS

The module has to be first loaded using either `modprobe tls` or `insmod <module_path>/tls.ko`.

The setsockopt function[4] is used to enable kDTLS on the socket in a manner similar to that of kTLS.

```
setsockopt(sd, SOL_UDP, UDP_ULP, "dtls", sizeof("dtls");
```

Providing cryptographic primitives

Changes were made to the `tls12_crypto_info...` structures to accommodate the DTLS1.3 record header encryption?? by adding a `sn_key` (sequence number key) item.

The API for providing the cryptographic primitives to the kernel remains the same as for the kTLS.

```
int setsockopt(int sock, int level, int option,
               const void* info, size_t info_len);
```

5.3.3 Record implementation

recvmsg()

The `dtls_sw_recvmsg()` function was introduced, its only dissimilarity to the `udp_recvmsg()` is that it calls the `tls_decrypt_sw()` function once there is a message. The encrypted messages are provided to the decrypt function via the `struct tls_decrypt_arg`. For the scope of this work, kDTLS is supported in zero-copy mode only. The kTLS module uses strparser[5] to handle a queue of messages, which is of no use for DTLS, but as it is used throughout the kTLS implementation, the DTLS implementation uses one `struct strp_msg` per connection which holds the `full_len` of the message and an `offset`.

[4] https://linux.die.net/man/2/setsockopt

[5] https://www.kernel.org/doc/Documentation/networking/strparser.txt

sendmsg()

Contradictory to recvmsg(), sendmsg does not handle any queue of messages, thus the only required changes are the AAD and header fabrication, subsequent sequence number encryption, and changing of the underlying transport protocol for sending the data.

sendfile()

The `sendfile()` call calls the `sendpage()` loads the data directly from the file descriptor and calls the same underlying function for encryption as does the sendmsg. This has a big impact on the performance as the data is moved inside the kernel only, never entering the userspace.

DTLS1.3 header and AAD

In the context of DTLS1.3, the header is utilized as the AAD (Associated Data) as described in 2.3.1.

The header construction occurs within the `dtls_make_aad()` function to be used as AAD, the header is additionally copied to a designated header portion of the `msg_en` structure. After the data encryption process, the record sequence number in the header is encrypted using the `dtls_encrypt_seq_num()` function.

The `crypto_info` structure for each cipher suite had to be modified to accommodate the record sequence number encryption by adding a `sn_key` (sequence number key) item.

For the receiving part, the header is handled in `tls_decrypt_sg()` and the record number is resolved using the `dtls_resovle_seq_num()`

Chapter 6

Flamegraphs

Flame graphs[1] are a graphical representation of stack-traced data obtained from a profiled application or system. The data are displayed in vertical columns, resembling flames; hence the name. There is also an inverted version of this graph, known as Iciclegraph. Those graphs help to easily track the code path and also to identify the number of CPU cycles.

6.0.1 Test subject

The examples were taken on a server that echoes the client's messages. The client sends 1000 times the message 13 bytes of text `"Hello world!"`, so that enough samples are captured so that all the transport functions are clearly visible and span a great portions of the flamegraph. So that we get a nice call stack for the receive and send functions.

6.0.2 Flamegraph generation

- diagnostic data used to generate the flamegraphs were obtained using `pref(1)`[2].

  ```
  $ perf record -a -g --call-graph dwarf -- ./server
  ```

- recorded data is then processed according to the guide[3]

  Note, the `perf.data` generated by step one has to be present in the `FlameGraph` folder obtained below.

  ```
  # git clone https://github.com/brendangregg/FlameGraph
  # cd FlameGraph
  # perf record -F 99 -a -g -- sleep 60
  # perf script | ./stackcollapse-perf.pl > out.perf-folded
  # ./flamegraph.pl out.perf-folded > perf.svg
  # firefox perf.svg  # or chrome, etc.
  ```

If expected function are missing the FlameGraph, increasing the `-F num` will result in more captured samples per second

[1] https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html
[2] https://man7.org/linux/man-pages/man1/perf.1.html
[3] https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html

The flamegraphs produced do not present a timeline, but instead display an approximate count of cycles spent within each function. If a function is invoked from another, the cycles are also accumulated in the calling function, and this accumulation continues recursively up the call stack. This method effectively illustrates the call hierarchy, as demonstrated in the case of the receive and send functions, both with and without kernel DTLS (kDTLS) enabled.

For full interactive `.svd` *flame-graphs, see. Non-kDTLS*[4] *kDTLS*[5]

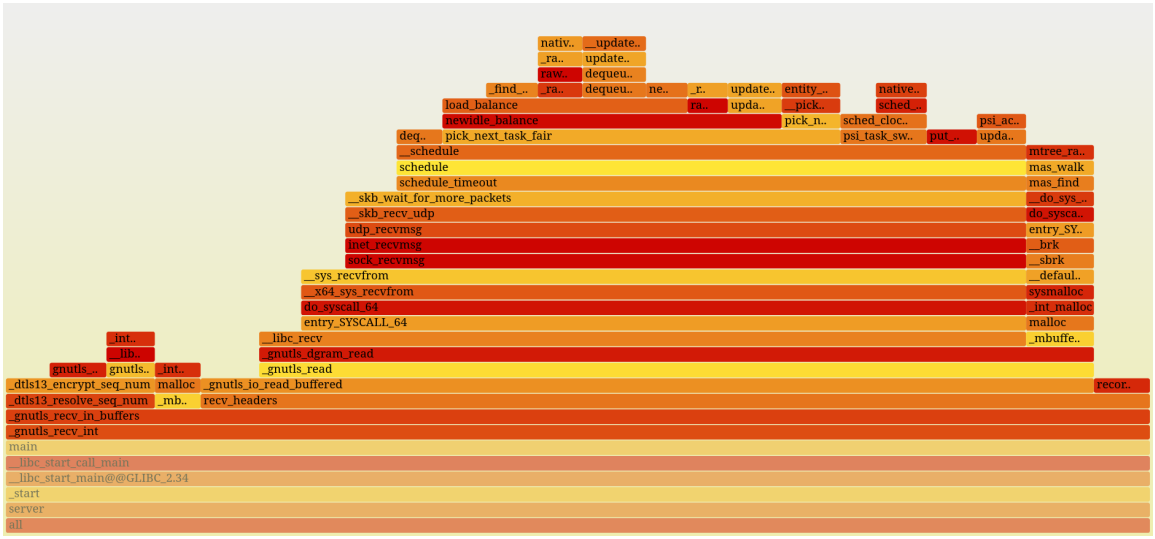


Figure 6.1: flamegraph — `gnutls_recv_int()` without kDTLS disabled

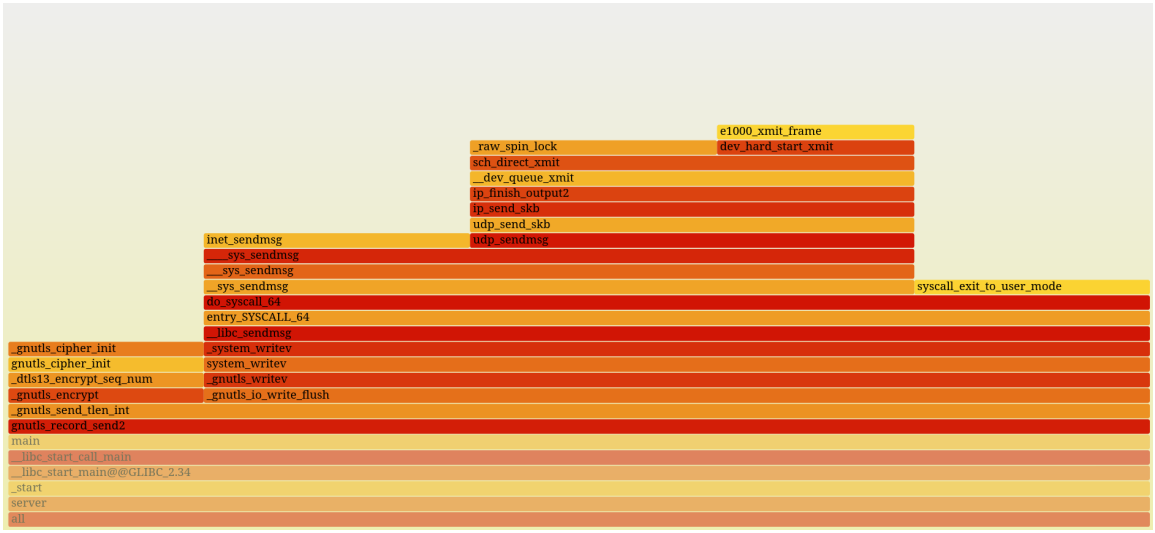


Figure 6.2: flamegraph — `gnutls_record_send2()` kDTLS disabled

An example of utilizing flamegraphs to detect bottlenecks can be seen during the development of this work'kernel module. Initially, the sequence number encryption function called `crypto_alloc_cipher(„aes", CRYPTO_ALG_TYPE_CIPHER, CRYPTO_ALG_ASYNC)`

[4]non-kDTLS: https://www.fkrenzel.cz/thesis/flamegraphs/flamegraph-recvmsg-GnuTLS.svg
[5]kDTLS: https://www.fkrenzel.cz/thesis/flamegraphs/flamegraph-recvmsg-kDTLS.svg

each time it was invoked; this occurred every time a message was sent or received, as illustrated in Figure 6.3.This was very taxing on the performance and fortunately was identified with the help of a flamegraph.

In the optimized version of the module, shown in Figure 6.4,the proportion of time spent in `tls_tx_record` which manages the transition of encrypted data has significantly decreased. Additionally, `tls_push_record`, which is responsible for encrypting both the application data and the sequence number, shows a marked improvement. Notably, `dtls_encrypt_seq_num` does not appear at all in the flamegraph of the optimized version, indicating that it was not captured within the selected sampling rate of the `perf` command. A similar issue was also traced in the GnuTLS implementation.

The flamegraph come in handy in situations when an application is hard to debug using conventional tools such as GDB[6] due to security reasons. This was the case for openconnect so flamegraphs were frequently used during the development.

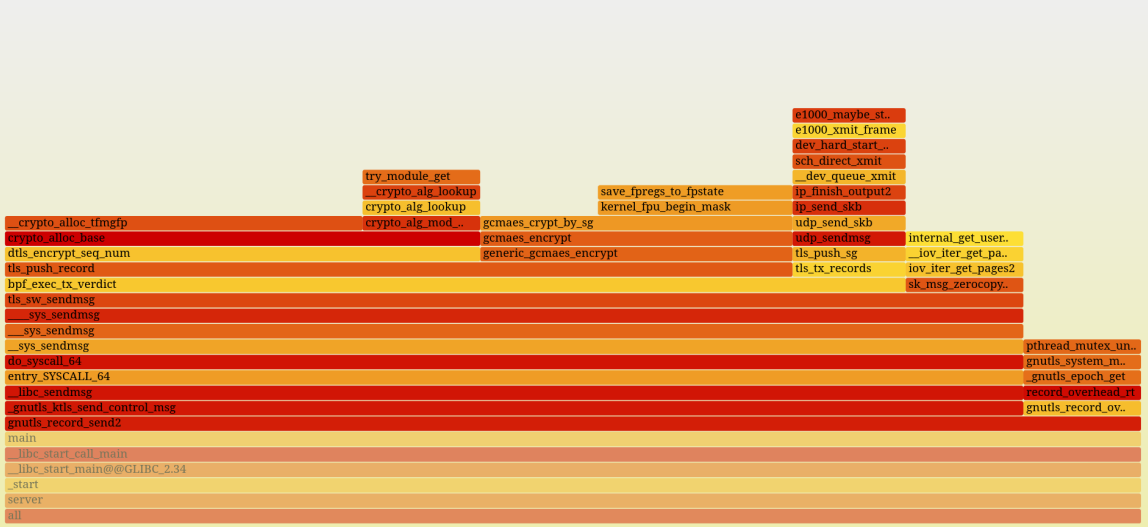


Figure 6.3: flamegraph — gnutls_record_send2() kDTLS enabled **obsolete**

[6] https://sourceware.org/gdb/

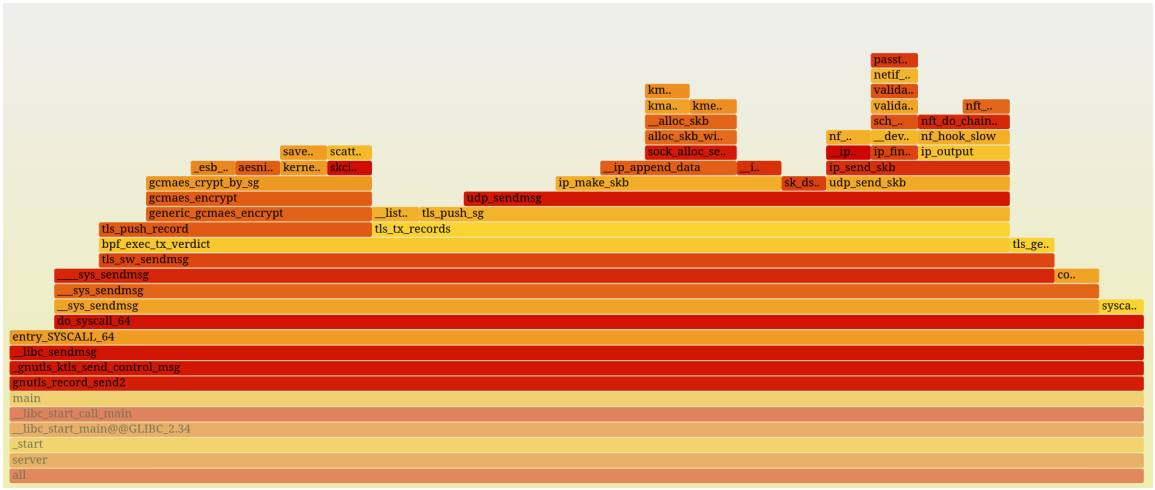


Figure 6.4: flamegraph — gnutls_record_send2() kDTLS enabled

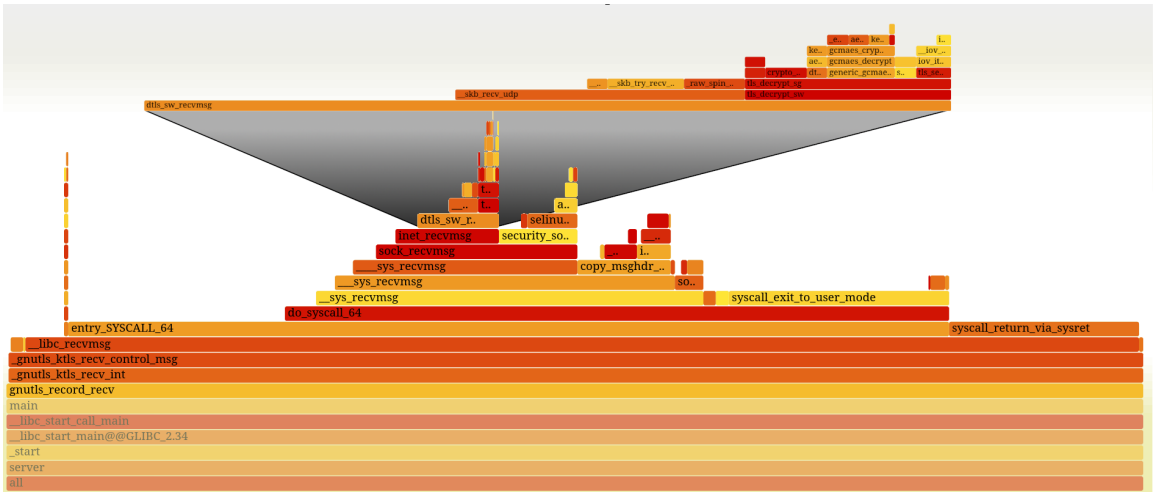


Figure 6.5: flamegraph — `gnutls_recv_int()` kDTLS enabled

Chapter 7

Performance statistics

7.1 Testing setup

Server

Specification	Details
Processor	Up to 7th Gen Intel Core i7
Memory	16GB DDR4 2133 MHz
Storage	NVMe SSD
Ports	RJ45 [1 Gb/s]
Operating System	Fedora 39 (server)

Table 7.1: Lenovo ThinkPad T470s Specifications

Client

Specification	Details
Processor	Intel Core i7 (11th Gen)
Memory	32GB LPDDR4x 4266MHz
Ethernet	RJ45 via thunderbolt [1 Gb/s]
Operating System	Fedora 39 (Workstation)

Table 7.2: ThinkPad X1 Nano Gen 2 Specifications

Router

Specification	Details
Model	MikroTik CRS326-24G-2S+RM
Switch Chip Model	98DX3236
CPU	ARMv7 800MHz
Size of RAM	512 MB
Firmware version	6.48.3
10/100/**1000** Ethernet Ports	24

Table 7.3: MikroTik CRS326-24G-2S+RM Specifications

7.1.1 Network topology

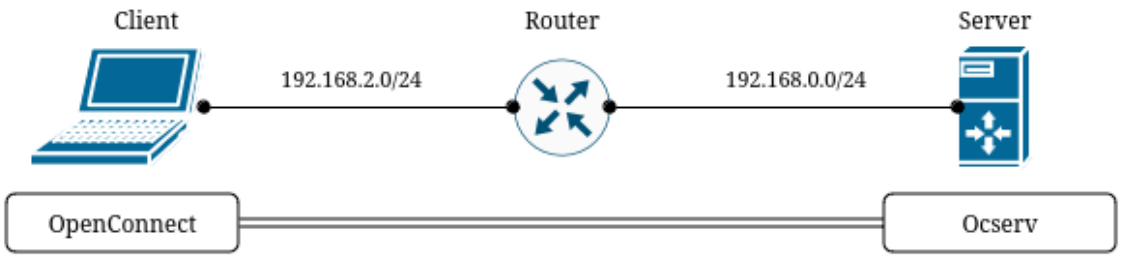


Figure 7.1: Network topology for testing

7.1.2 OpenConnectVPN Benchmarks Using DTLS Protocol

To evaluate the performance of the OpenConnectVPN connections across different DTLS protocols, the iperf3[1] tool was used. iperf3 is a widely recognized network testing tool that measures bandwidth and the quality of a network link.

The benchmarking process lasted for 10 minutes, during which a TCP throughput test was conducted to measure the bandwidth of the connection. TCP was chosen for this test due to its prevalent use in VPN environments, where reliability and data integrity are crucial.

	DTLS1.2	DTLS1.3
total bytes [GB]	1.86	1.52
transmit rate [Mbits/s]	53.3	43.5

Table 7.4: OpenconnectVPN benchmark results

The observed slowness in DTLS1.3, compared to DTLS1.2, is primarily attributed to the implementation rather than the DTLS1.3 protocol itself. Flamegraphs were utilized to identify bottlenecks, which have been addressed, but further refinements may still be necessary.

[1] https://iperf.fr/iperf-doc.php

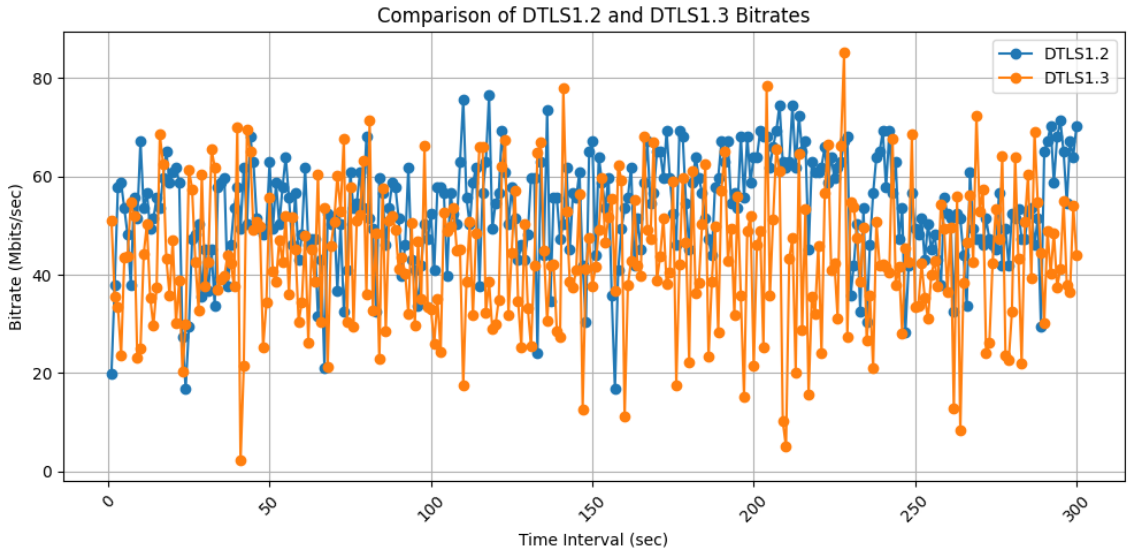


Figure 7.2: OpenconnectVPN iperf3 benchmark DTLS1.2 x DTLS1.3

kDTLS Performance Analysis

The integration of kDTLS support into Openconnect encountered several technical challenges, leading to the decision to exclude it from this project due to the absence of likely performance gains. The DTLS1.3 protocol does not offer substantial performance enhancements compared to its predecessors. Furthermore, the TLS1.3 kTLS implementation, which closely resembles the DTLS1.3 protocol, up until recently underperformed relative to the older TLS1.2 implementation. This underperformance is attributed to the inherent characteristics of the protocol [5]. Additionaly the overhead introduced by the (D)TLS session setup specificaly the provisioning of keys and allocation all the neccesary resources in the kernel has to be taken in to consideration as kTLS module doesn't support key update meaning that it doesn't support chaning the keys on a already established connection, which significantly reduces the lifespan of the session. This is the reason kDTLS is used by default by most libraries [2]

Splice()

Potential enhancements could be realized through the `splice()` system call, which facilitates direct data movement between file descriptors, thus reducing the overhead of data copying between kernel and user space. This operation is often utilized by `pipes`, which are communication channels that allow data to pass from one process to another sequentially. However, a significant limitation arises with the TUN/TAP device used by OpenconnectVPN, which does not support these `splice()` operations [6].

[2] https://pagure.io/fesco/issue/2871#comment-824428

Chapter 8

Future of work

8.1 Related work

8.1.1 ktls-utils

The proof-of-concept utility ktls-util[1] aims to establish a TLS connection on a socket created in the Linux kernel space. As implementing the TLS handshake into the kernel itself is considered by many an inadequate approach as described in Subsection 8.1.1; it will be carried out in the user space, as is the case for the standard kTLS userspace socket; however, this raises the question of how to provide the userspace with access to the kernel socket.

The ktls-util resolves this by launching a separate user-space process, referred to as "user agent", which will be responsible for negotiating and setting the keyring material onto the kernel socket.

First, a connection must be established between the host and the peer, then the user agent creates a socket using the newly introduced address family type `AF_TLSH` and listens for incoming connections from the kernel using the system socket function `listen()`. The kernel then provides a TCP socket to the userspace process through the listening socket via a new call `tls_client_hello(*socket, *done)`; the `*done` stands for a function pointer to a function that is to be executed once the keyring material is negotiated using `gnutls_handshake()`and is set onto the kernel socket using the standard kTLS API. Once this is done, the userspace lets go of the kernel socket and a TLS session on a kernel socket is successfully established.

You can see the process in figure 8.1

In-kernel TLS handshake

At the time of writing this thesis, the Linux kernel crypto is not suitable for TLS handshake, as many required cryptographic functions are obsolete and would require updating and extensive testing.

The TLS handshake constitutes a significant portion of the TLS code, and its introduction will lead to an increase in maintenance. There are also many corner cases that would need to be handled, this would add to code complexity, thus reducing readability.

Another pain point would be the patching of the kernel, userspace library is much easier to fix and redistribute/update.

[1] https://github.com/oracle/ktls-utils

There is also demand for socket in kernel space to support the (D)TLS. This problem alongside a solution was presented at 2022 Linux Storage, Filesystem, Memory-management and BPF Summit by Chuck Lever and Hannes Reinecke [3]. The software solution for the problem is known as the ktls-util[2], which creates a listening daemon on in the userspace waiting for a handshake request from the kernel space. the request contains a reference to the socket as well as a pointer to a callback function done which is to be executed after the handshake was complete, and the socket is free from userspace. [3]

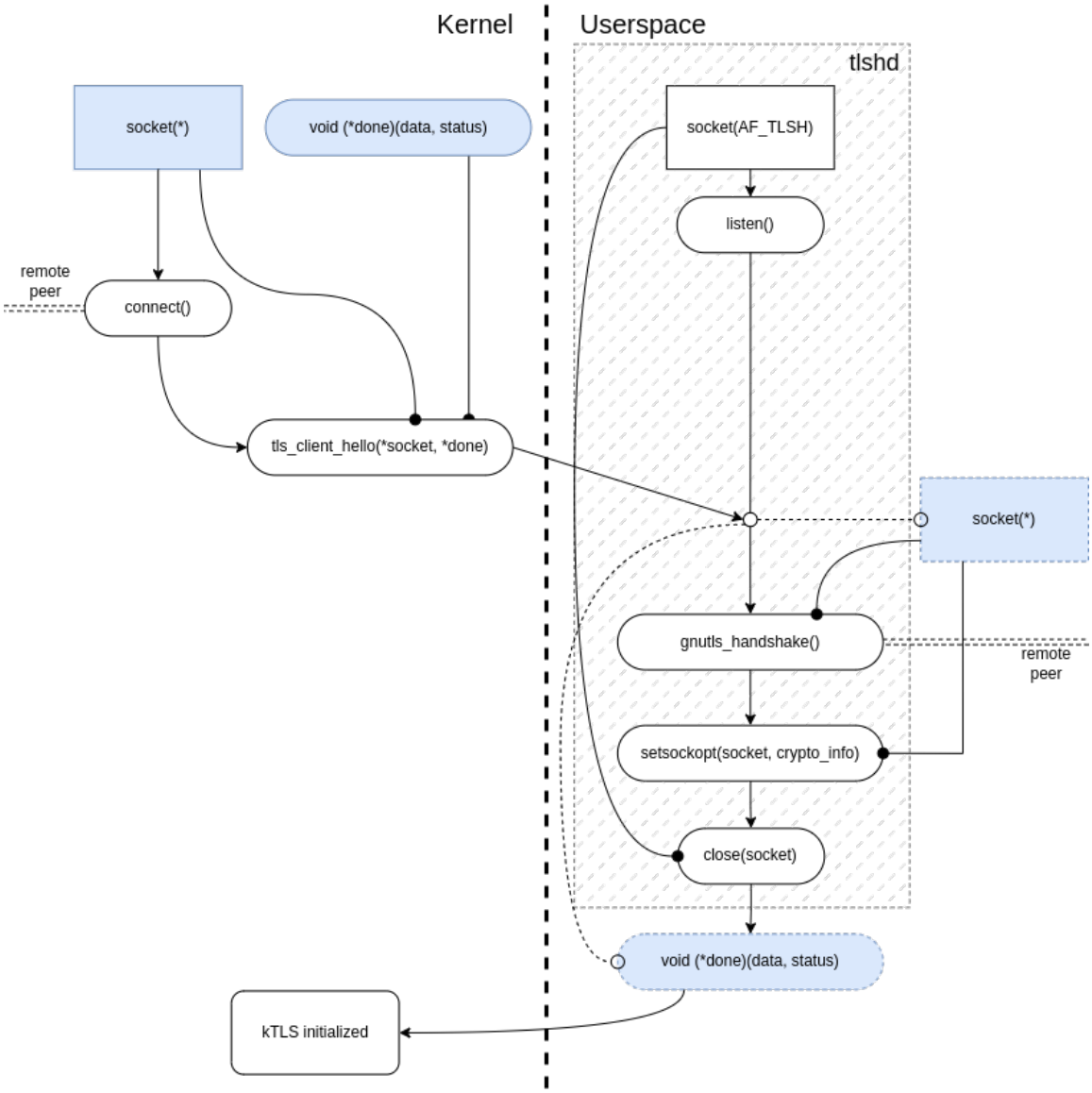


Figure 8.1: ktls-util setup diagram

[2] https://github.com/lxin/tls_hs/
[3] https://lwn.net/Articles/896746/

8.1.2 Berkeley Packet Filter

Linux kernel version 4.20 received patches to support the combination of two until then mutually exclusive features, namely kTLS and BFP (Berkeley Packet Filter). This allows packet inspection and filtering ahead of the data encryption, thus saving significant resources as opposed to userspace TLS implementations where the data is encrypted and then sent to the kernel for BPF processing.

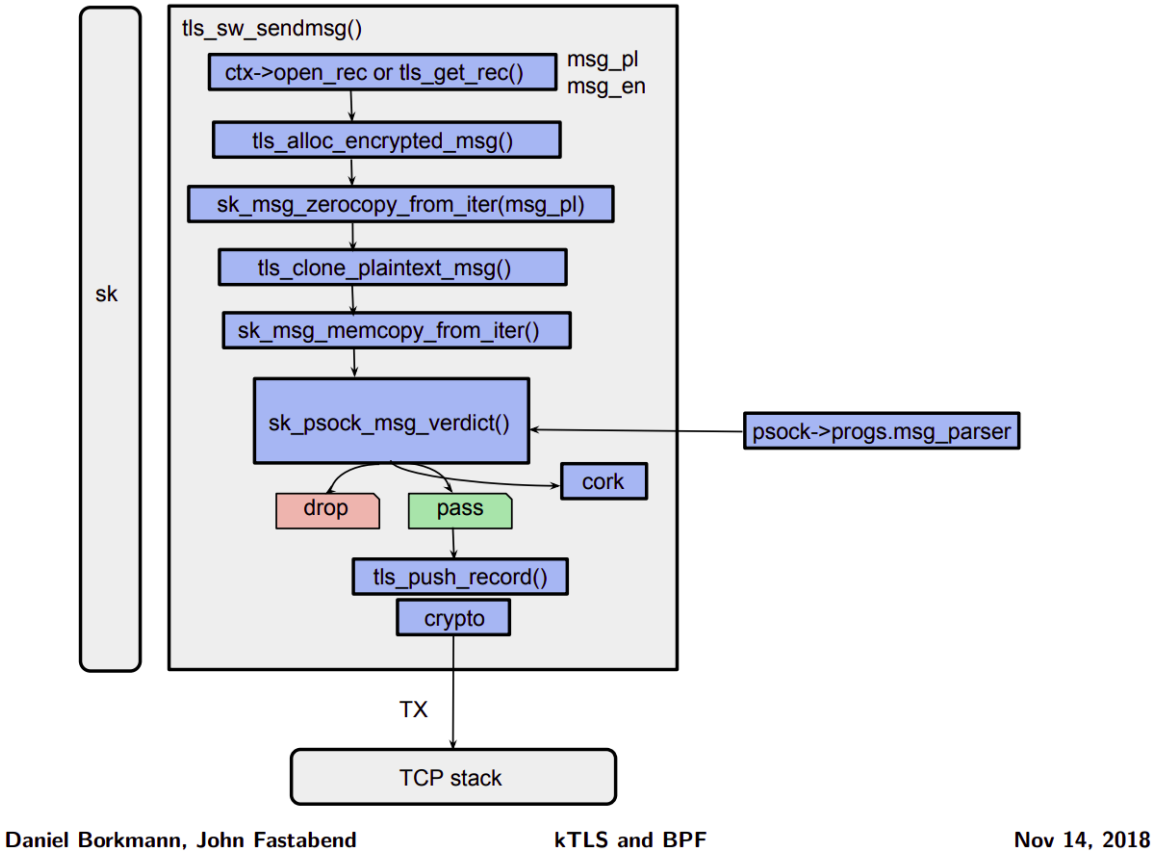


Figure 8.2: kTLS with BPF

8.1.3 Kernel Packet filtering

The combination of Kernel TLS socket 8.1.1 and the BPF 8.1.2 would allow for a VPN DTLS connection running purely in kernel with the possibility to append the VPN specific data to the packets directly in the kernel and would also allow for a deep packet inspection [2] i.e., application layer filtering, so the packet wouldn't need to enter userspace for the inspection. This approach bears resemblance to WireGuard VPN[4], which employs a dedicated Linux kernel module for similar functionalities. However, our proposed method would leverage BPF to link a userspace program directly to the kernel, specifically interfacing with the k(D)TLS module, thereby obviating the need for a separate kernel module.

[4] https://www.wireguard.com/

Chapter 9

Future

The GnuTLS implementation is set to be upstreamed following its revision. The openconnect VPN DTLS1.3 support will follow suit if a performance improvement is accomplished by further optimizing the GnuTLS implementation. Meanwhile, it would be beneficial to identify a more suitable use case for kDTLS than OpenconnectVPN. As streaming services transition to the QUIC protocol, the Internet of Things (IoT) could be a promising candidate. This potential lies not in performance enhancement, but in reducing power consumption. Richard W.M. Jones conducted an unpublished test on kTLS using nbdkit[1], a toolkit that allows users to create custom Network Block Device (NBD) servers. This test revealed that kernel TLS improves power efficiency by enabling sleep mode when the server is inactive. Other improvement of DTLS1.3 compared to its predecessor, increase the potential use case of the DTLS1.3 in the world of IOT.

9.1 Upstreaming the patches

Although functioning as expected, the patches lack many features required for smooth operation. Additionally, The patches did not undergo the comprehensive review process required to meet established security standards. see. the outline of the specific deficiencies and additional requirements for the GnuTLS and Kernel patches below:

9.1.1 GnuTLS

The GnuTLS patch falls short of full compliance with the established protocol definitions, omitting some components deemed irrelevant for the objectives of this project:

- Connection ID is not implemented.
- Comprehensive interoperability CI tests with at least one other library implementing DTLS1.3 are essential to ensure compatibility across different systems.

9.1.2 kDTLS

The primary objective of the Kernel patch is to assess the feasibility of protocol implementation within the kernel environment to aid in future development, so the following items have to be resolved before targeting upstream:

[1] https://libguestfs.org/nbdkit.1.html

- `recvmsg()`5.3.3 currently doesn't support the dynamic sequence number length2.3.1, meaning the 16bit length is expected, and the decryption fails otherwise.
- Connection ID is not implemented.
- Currently, only zero-copy for `recvmsg()`5.3.3 is supported, which might result in failure on some systems or in specific scenarios.
- The renegotiation of keys is not supported (this also applies for kTLS) which make kTLS unusable for applications where either a long-lasting connection is expected or one that exceeds the number of messages that are considered safe for a given cipher.

Chapter 10

Conclusion

This thesis conducts an in-depth analysis of the DTLS1.3 protocol, emphasizing its essential components for both userspace and Linux kernel implementations.

A significant outcome of this work is the creation of a GnuTLS DTLS1.3 implementation that supplies cryptographic materials to the kernel TLS module. This was done in preparation for the integration of DTLS into the module, enhancing the existing kernel TLS (kTLS) framework. The integration of DTLS into the kTLS module allows userspace applications dependent on datagram-based connections, such as UDP, to potentially achieve enhanced performance and greater power efficiency.

Upon evaluating the potential impact of the kernel module on OpenConnectVPN, the specific use of TUN/TAP devices in OpenConnect presents a limitation. This configuration prohibits bypassing the data transition to userspace, which would otherwise minimize the use of data buffers and the frequency of context switches. However, the Internet of Things (IoT) sector presents a more favorable scenario. IoT systems, already leveraging DTLS1.3 features, stand to gain additional improvements from the kDTLS module, speed, latency, power efficiency and overall device performance.

Chapter 11

Reference

The following is a compilation of invaluable resources that have significantly contributed to the successful completion of this work.

- Sockets in the Linux Kernel by Rami Rosen [10] — In depth learning resource for Linux kernel sockets.
- Addition of BPF to kTLS by Daniel Borkmann and John Fastabend [2].
- TLS 1.3 Rx improvements in Linux 5.20 by Jakub Kicinski [5]
- Virtualization [4] — Tutorial series on how to set up QEMU VM (virtual machine). In the context of this work. it was used for setting up the kernel module development environment.
- Linux kernel module development blog post — This guide was created directly as a result of this work, intended for newcomers to get them running on kernel development with a safe and efficient setup.

Bibliography

[1] Abouchaev, A. *[net-next v4 3/6] net: Add UDP ULP operations, initialization and handling prototype functions.* 8. September 2022. Available at: https://lore.kernel.org/linux-kernel/20220909001238.3965798-4-adel.abushaev@gmail.com/. Email message to netdev@vger.kernel.org.

[2] Borkmann, D. and Fastabend, J. Combining kTLS and BPF for Introspection and Policy Enforcement. *Cilium.io*, 2021. Available at: http://vger.kernel.org/lpc_net2018_talks/ktls_bpf_paper.pdf. Whitepaper.

[3] Corbet, J. *Extending in-kernel TLS support* online. 2022. Available at: https://lwn.net/Articles/896746/. Accessed: 2024-05-06.

[4] Dhar, A.; Prokop, J. and Dirschel, N. *Virtualization — Seting up virtual machines using qemu* online. 2024. Available at: https://developer.fedoraproject.org/tools/virtualization/installing-qemu-on-fedora-linux.html. Accessed: May 7, 2024.

[5] Kicinski, J. *TLS 1.3 Rx improvements in Linux 5.20* online. July 2022. Available at: https://people.kernel.org/kuba/tls-1-3-rx-improvements-in-linux-5-20. Accessed: May 7, 2024.

[6] Pokorný, F. *Linux VPN Performance and Optimization.* 2016. Master's theses. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.vut.cz/en/students/final-thesis/detail/96283. Adviser: Kašpárek, Tomáš; Referee: Michal, Bohumil.

[7] Rescorla, E. *The Transport Layer Security (TLS) Protocol Version 1.3* RFC 8446. RFC Editor, august 2018. Available at: https://doi.org/10.17487/RFC8446.

[8] Rescorla, E.; Tschofenig, H. and Modadugu, N. *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3* RFC 9147. RFC Editor, april 2022. Available at: https://doi.org/10.17487/RFC9147.

[9] Restuccia, G.; Tschofenig, H. and Baccelli, E. Low-Power IoT Communication Security: On the Performance of DTLS and TLS 1.3. *CoRR*, 2020, abs/2011.12035. Available at: https://arxiv.org/abs/2011.12035.

[10] Rosen, R. Sockets in the kernel. In:. Haifa Linux Club, August 2009. Linux Kernel Networking – advanced topics. Available at: http://www.haifux.org/lectures/217/.

Appendix A

Content of Attached DVD

Patches are either distributed as a one file with suffix `-all` or a series in which they were committed during development in a `git_history` branch.

Folder/File	Content
gnutls-dtls1_3/	Patches for GnuTLS cryptographic library.
kernel-dtls1_3/	Patches for the kernel implementation.
openconnect-dtls1_3/	Patches for openconnect (client) and ocserv (server).
examples/	Code examples used for testing
latex/	Sources for this text
xkrenz00.pdf	This text

Table A.1: Content of the attached CD

Appendix B

Versions of the used software

Name	Version	Upstream Git commit number
Openconnect	9.12	d2025f9d49637065aaa15f506b022c45765cf6b7
Ocserv	1.2.5	72b8e19cac44bf1ca0246791967cdc6a012d6d55
GnuTLS	3.8.3	a3c19bcaf679ed7e0fcb71edd89387b889be533a
Wolfssl	5.6.3	c23559a91c55902deefe6f1a5b72c623250b96c9
Fedora Linux kernel-ark	6.7.0	9bacdd8996c77c42ca004440be610692275ff9d0
Iperf3	3.16	

Table B.1: Versions of the used software