Multipath transport protocol offloading

Rebecka Alfredsson beckaalfredsson@hotmail.com

Faculty of Health, Science and Technology

Master thesis in Computer Science

Second Cycle, 30 hp (ECTS)

Supervisor: Prof. Dr. Andreas Kassler, andreas.kassler@kau.se

Examiner: Dr. Per Hurtig, per.hurtig@kau.se

Karlstad, 2022-06-20

Abstract

Recently, we have seen an evolution of programmable network devices, where it is possible to customize packet processing inside the data plane at an unprecedented level. This is in contrast to traditional approaches, where networking device functionality is fixed and defined by the ASIC and customers need to wait possibly years before the vendors release new versions that add features required by customers. The vendors in the industry have adapted and the focus has shifted to offering new types of network devices, such as the SmartNIC, IPU, and DPU. Another major paradigm shift in the networking area is the shift towards protocols that encrypt parts of headers and contents of packets such as QUIC. Also, many devices such as smart phones have support for multiple access networks, which requires efficient multipath protocols to leverage the capabilities of multiple networks at the same time.

However, when using protocols inside the network that requires encryption such as QUIC or multipath QUIC, packet processing operations for the en/decryption process are very resource intensive. Consequently, network vendors and operators are in need to accelerate and offload crypto operations to dedicated hardware in order to free CPU cycles for business critical operations. Therefore, the aim of this study is to investigate how multipath QUIC can be offloaded or hardware accelerated in order to reduce the CPU utilization on the server.

Our contributions are an evaluation of frameworks, programming languages and hardware devices in terms of crypto offloading functionality. Two packet processing offloading prototypes were designed using the DPDK framework and the programming language P4. The design using DPDK was implemented and evaluated on a BlueField 2 DPU. The offloading prototype handles a major part of the packet processing and the crypto operations in order to reduce the load of the user application running on the host. A evaluation show that the throughput when using larger keys are only slightly decreased. The evaluation gives important insights in the need of crypto engines and/or CPUs with high performance when offloading.

Keywords

QUIC, multipath, hardware offloading, DPDK, crypto, DPU

Sammanfattning

Nyligen har vi sett en utveckling av programmerbara nätverkskort, där det är möjligt att anpassa paketbehandling inuti dataplanet på en nivå aldrig tidigare skådad. Detta står i kontrast till traditionella metoder, där funktionaliteten i nätverkskort är definierad av ASIC och kunderna måste vänta eventuellt i år innan leverantörerna släpper nya versioner som lägger till funktioner som krävs. Leverantörerna i branschen har anpassat sig och fokus har flyttats till att erbjuda nya typer av nätverkskort, såsom SmartNIC, IPU och DPU. Ett annat stort paradigmskifte i nätverksområdet är skiftet mot protokoll som krypterar delar av packetinnehållet i paket som QUIC. Många enheter som smarta telefoner har också stöd för flera nätverk, vilket kräver effektiva flervägsprotokoll för att utnyttja funktionerna i flera nätverk samtidigt.

Men när protokoll i nätverket som kräver kryptering som QUIC eller multipath QUIC används är paketbearbetningen för kryptera och avkryptera mycket resurskrävande. Nätverksleverantörer och operatörer är i behov av att påskynda och avlasta kryptooperationer till dedikerad hårdvara för att frigöra CPU-cykler för andra kritiska operationer. Syftet med denna studie är därför att undersöka hur multipath QUIC kan avlastas eller accelereras med hårdvara för att minska CPU-användningen på servern.

Våra bidrag är en utvärdering av ramverk, programmeringsspråk och hårdvaruenheter när det gäller kryptoavlastningsfunktionalitet. Två prototyper för att avlasta paketbearbetning designades med DPDK-ramverket och programmeringsspråket P4. Designen med DPDK implementerades och utvärderades på en BlueField 2 DPU. Den avlastande prototypen hanterar en större delen av paketbehandlingen och kryptooperationerna för att minska belastningen av användarprogrammet för CPUn. En utvärdering visar att det dataflöde som använder större krypteringsnycklar bara har något minskad flödeshastighet. Användning av avlastningsprogrammet ger minskad flödeshastighet jämfört med obearbetade paket. Utvärderingen ger viktiga insikter om behovet av hårdvarustöd för kryptooperationer och/eller processorer med hög kapacitet vid avlastning.

Nyckelord

QUIC, multipath, avlastning med hårvara, DPDK, krypto, DPU

Acknowledgements

I want to thank my supervisors Michael Eriksson and Zaheduzzaman Sarker at Ericsson Research for their support, motivation, curiosity, and constant engagement.

I would also like to thank Leif Johansson and Martin Julien at Ericsson for providing me with hardware equipment and knowledge necessary for this project.

Lastly, I would also like to thank my supervisor at Karlstad University Prof. Andreas Kassler for his support, valuable insights, and for introducing me to the computer networking research.

Acronyms

AEAD Authenticated Encryption with Associated Data
AES-GCM Advanced Encryption Standard - Galois Counter Mode
AES-CCM Advanced Encryption Standard - CBC counter mode
AES-NI Advanced Encryption Standard New Instructions
ASIC Application-Specific Integrated Circuit
CPU Central Processing Unit
DCCP Datagram Congestion Control Protocol
DPDK Data Plane Development Kit
DPU Data Processing Unit
DUT Device Under Test
EAL Environment Abstraction Layer
FPGA Field Programmable Gate Array
GPU Graphics Processing Unit
HDL Hardware Description Language
HTTPS Hypertext Transfer Protocol Secure
IETF Internet Engineering Task Force
IPDK Infrastructure Programmer Development Kit
IPSec Internet Protocol Security
IPU Infrastructure Processing Unit
IV Initialization Vector
MAC Media Access Control
MPTCP Multipath Transmission control protocol
NIC Network Interface Card
OS Operating System
P4 Programming Protocol-independent Packet Processors
PCIe Peripheral Component Interconnect Express
PMD Poll Mode Driver
RPC Remote Procedure Call
SDK Software Development Kit
SoC System on a Chip
SPDK Storage Performance Development Kit
TCP Transmission control protocol
TLS Transport Layer Security
UDP User Datagram Protocol

Contents

Chapter 1

Introduction

In this chapter an overview is given of the evolution of both transport layer protocols as well as hardware devices for networking. This evolution which has led to an increased interest in offloading and hardware acceleration is described. The problem description, and goal of the thesis is then discussed followed by the ethical concerns and impacts on sustainability. The methodology, stakeholders, delimitation and finally the outline of the thesis is presented.

1.1 Overview

For a long time, Transmission control protocol (TCP) has been the most used transport layer protocol on the Internet [24]. TCP offers features such as reliability, congestion control, and also security together with the cryptographic protocol Transport Layer Security (TLS). Since the beginning of Internet, computer advancement has increased drastically. The amount of different network devices and the computing power has evolved. As this evolution has occurred, TCP has still been the de facto standard transport layer protocol since it was first developed in 1974.

In 2012 [1] the development started of the transport layer protocol QUIC, and was later submitted to Internet Engineering Task Force (IETF) working group in 2016 and got standardized in RFC9000 [22]. QUIC offers similar features as TCP with TLS, but with different implementations, to better suit today's Internet and be more adaptable to changes. The fast adaptability is one of the key goals of QUIC, which has led to QUIC being implemented in the user-space, compared to many other transport protocols, such as TCP, which are implemented in the Operating System (OS) kernel. By implementing protocols in the OS, changes often take long time since users could be many years behind in OS upgrades. QUIC uses Frames, further described in section 2.1.1, which allows people and companies the possibility to extend QUIC according to their own requirements. Examples of QUIC extension frames are datagram frames [38], used for sending and receiving unreliable datagrams, and ACK_MP frames [31], used in multipath QUIC for sending ACK packets when multiple packet number spaces

are used.

In the beginning of the Internet, devices were communicating to each other by being connected with a fixed network. These devices also often only used one network interface. Today, there is a significant shift towards utilizing the resources of multiple paths and access networks available between two communicating peers. Many devices are multi-homed, mobile phones can use Wi-Fi and mobile network interfaces such as 3G, 4G, and 5G. However, many of the common transport layer protocols still only utilize a single network interface. Multipath Transmission control protocol (MPTCP) got standardized in RFC6824 [15] and quickly got adopted by Apple's Iphone voice recognition application Siri, Apple Maps and Apple Music [4]. The use of MPTCP increased the performance and seamless handover between the Wi-Fi and cellular network. There are also ongoing work to extend the Datagram Congestion Control Protocol (DCCP) protocol into multipath DCCP [2], to support more latency sensitive protocols that do not require reliable communication. The work of extending the protocol QUIC with multipath capabilities has already begun [31].

Hardware used for networking have also been evolving continuously, since the beginning of the Internet. For example, TCP accelerators are used to make TCP better suit today's Internet where speed is of high importance [37]. However, many of these devices are traditionally fixed-function, which means that the manufacturers decides which functionality the device should support. If protocols on the Internet are modified or new protocols are being deployed, these fixed-function might need to be updated or replaced [5]. However, the need to buy devices to adapt to changes are expensive. A solution to this problem is the ability to program network devices. An evolution has occurred from fixed function devices programmed by the vendors, devices that can be programmed by developers such as Smart Network Interface Card (NIC), Data Processing Unit (DPU) and Infrastructure Processing Unit (IPU). With these new programmable devices, the user can implement specific desired network behaviours directly inside the data plane of the network. Such initiatives have their origin from data center operators, where programmable networking equipment can be used to adapt their operation to suit its specific needs.

A new use case with SmartNICs, DPUs, and IPUs is to implement functionality in these devices in order to offload packet processing tasks from the Central Processing Unit (CPU). This method is called hardware offloading and the specific packet processing functionality is executed in these hardware devices directly instead of on the CPU. Some examples of hardware offloading is using graphics cards for tasks such as gaming, machine learning, and cryptocurrency mining [29]. The load of the CPU can also be decreased by using accelerating software or hardware. Software acceleration, such as Advanced Encryption Standard New Instructions (AES-NI), implements a special instructions set for x86 processors which are used to increase performance. By using AES-NI the implementation using AES can improve the speed by a factor of 3 to 10 [40]. Hardware acceleration can be integrated in multiple ways on the device, for example into the System on a Chip (SoC) as an separate processor or core, in a coprocessor

on the circuit board, or on a chip connected to the main board by the Peripheral Component Interconnect Express (PCIe) bus [7]. Crypto libraries, such as gnutls or openSSL, can be used for accessing the cryptographic hardware accelerator in user space.

1.2 Problem Description

Since QUIC aims to be a competitor to TCP and become a commonly used transport layer protocol, the disadvantages of the protocol need to be taken into account and managed. One of the major disadvantages that affects QUIC is the heavy CPU utilization that occurs during crypto operation which is inherited from the TLS protocol. Measurements [46] of different QUIC implementations have shown that the crypto operations can be responsible for up to 40 percent of the CPU utilization per connection. The Authenticated Encryption with Associated Data (AEAD) encryption and decryption stand for 75-80 percent of the crypto operations, which is stateless functions. The hypothesis is that this functionality could benefit from being offloaded from the CPU to the NIC data plane. The measurements also showed that packet I/O can be responsible for 40 percent of the CPU usage. By offloading most of the packet processing to the NIC or use kernel-bypass, the CPU usage for packet processing can be decreased and CPU cycles that are freed due to the offloading could be re-used for other tasks on the server. Such offloading should be beneficial for both QUIC and its multipath variant multipath QUIC.

The main question this thesis aims to answer is:

How can different functions of multipath QUIC packet processing be offloaded from the CPU or accelerated with hardware?

1.3 Thesis Goals

In order to answer the above research question, this thesis makes several contributions in order to achieve the following goals. The first goal of this degree project is to investigate and scout what frameworks, programming languages and hardware are available for offloading and/or accelerating multipath QUIC packet processing operations. Since the programmability of network devices is a fairly new concept, the maturity of the tools and devices also need to be investigated before being used.

The second goal for the degree project is to select a framework or programming language and a hardware device to implement a prototype for offloading crypto and parts of the packet processing of multipath QUIC. Depending on the tool and device selected, hardware acceleration might be supported.

The final goal is to develop a testbed and measurement setup that can be used to evaluate the performance that can be achieved when offloading packet processing functionality.

1.4 Ethics and Sustainability

This project will not contain any ethical issues. No individual data including sensitive personal data are collected and no trade secrets are discussed.

By offloading CPU heavy tasks, such as the encryption of QUIC, to dedicated hardware instead of the CPU a result can be that the energy consumption is reduced which would aid sustainability. Hardware offloading also frees the processor which then could be used for other business critical tasks, that would otherwise be scheduled and performed later, which also results in better energy efficiency.

1.5 Methodology

First, this degree project will start by an investigation phase, where the goal is to scout and find which different kind of frameworks or programming languages that support implementing AEAD crypto offload or acceleration for multipath QUIC. The hardware devices that support this will also need to be investigated. This step will require much reading and contacting people, both within and outside the organization for information that is unavailable online. Choosing a feasible framework or programming language for this project is more important than hardware, which is the reason why the hardware decision is of lower priority. In order to make these decisions, a comparison of the maturity and features has to be done.

The final step is to generate design concepts of the feasible solutions from the previous step. One of these solutions is selected and implemented with the chosen hardware. The prototype is lastly evaluated where throughput is first investigated using different packet and key sizes, and then the latency at different sending rates. Depending on which hardware becomes accessible, a benchmark will be made to measure the throughput of different crypto engines. If other benchmarks related to the project are available, they will be evaluated.

1.6 Stakeholders

This project is hosted by Ericsson Research, which is a department of the multinational company Ericsson. Ericsson operates within telecommunication networks, cloud software and services, and emerging businesses [14]. This degree project is beneficial for Ericsson Research, due to multiple reasons. This project enlightens a use case with the chosen framework or programming language and hardware device, as well as multipath QUIC which is still under development and has a working-group at Ericsson. It is also beneficial for Ericsson due to the results the investigation phase provides. This project is useful to gain information of the current states of the frameworks investigated since they are under development. This project provides information of the maturity of the frameworks and what features are supported at the time of drafting this thesis.

1.7 Delimitations

In this degree project, parts of the packet processing using multipath QUIC will be implemented in the prototype. The packets will be sent only in the 1-RTT format, which exclude some QUIC packet types. In these packets, for example the Initial and Handshake packets is where parts of the crypto is performed, such as the TLS 1.3 handshake and key derivation. The evaluation of the prototype will focus on AEAD_AES_128_GCM and AEAD_AES_256_GCM for the AEAD algorithms, and not AEAD_CHACHA20_POLY1305 or AEAD_AES_128_CCM due to AES_GCM being the most commonly used crypto algorithms [17]. The evaluations of the prototype will be measured in single experiments, due to time constraints. Therefore, no average or standard deviation will be included in the evaluation, which might have resulted in more accurate measurements.

1.8 Outline

This thesis is structured as follows. In Chapter 2, the necessary background needed to understand multipath QUIC is provided. The thesis continues with presenting different frameworks, programming languages and hardware devices which could be used for hardware offloading and/or acceleration. Lastly in Chapter 2, the related work is presented. In Chapter 3, the design selection is discussed followed by a design of the offloading application prototype using Data Plane Development Kit (DPDK), as well as for an alternative Programming Protocol-independent Packet Processors (P4) design is given. In Chapter 4, implementation details are presented for the prototype design using DPDK. In Chapter 5, the test bed is illustrated followed by the evaluation of the prototype and benchmarks using DPDK crypto device drivers and OpenSSL. Finally, in chapter 6, concludes the thesis and describes future work.

Chapter 2

Background and Related Work

In this chapter, the background necessary for understanding this project is presented together with previous related work. Section 2.1 presents the transport layer protocol QUIC and the multipath extension multipath QUIC, which is used in this project. The frameworks DPDK, Infrastructure Programmer Development Kit (IPDK), and DOCA, and programming languages P4 and Hardware Description Language (HDL) are then presented in section 2.2 followed by the hardware devices SmartNIC, IPU, DPU, and Field Programmable Gate Array (FPGA) in section 2.3. Section 2.4 presents different previous work this thesis is based upon.

2.1 QUIC

In this section, the transport layer protocol QUIC is presented followed by a more detailed description of the encryption and multipath capabilities.

2.1.1 QUIC Overview

QUIC [1] is a transport layer network protocol, which first was developed by Google in 2012 by the name gQUIC and with the main goal to improve Hypertext Transfer Protocol Secure (HTTPS) in Chromium compared to using TCP with TLS. QUIC is an alternative protocol to TCP with TLS. IETF started working on QUIC and standardized it in May 2021 in RFC9000 [22].

QUIC packets have different formats depending on the state of the communications. QUIC packets are divided into those having long headers (Version negotiation, Inital, 0-RTT, Handshake, and Retry packets) and short header (1-RTT packets). In this degree project only 1-RTT packets will be used. A generic 1-RTT packet structure can be seen in figure 2.1.1, where the size in bytes for each field can be seen in the parentheses. The flags, destination connection ID, and packet number fields are considered the QUIC header. A 1-RTT packet contains one or more frames, which is sent in the payload of the packet. The frames are used to send different types of data, and uses specific

types depending on the purpose. These types can for example be Stream, Ack, or Crypto. At the handshake of a connection between two peers, transport parameters are exchanged. These parameters are used for setting the rules for the communication, for example the initial maximum data, maximum acknowledgment delay, and initial maximum unidirectional streams.

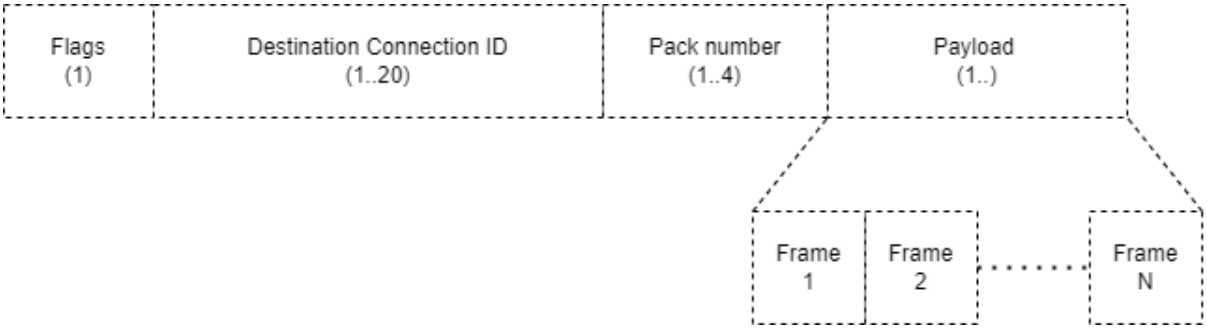

Figure 2.1.1: Generic 1-RTT packet format

QUIC [22] is encapsulated in User Datagram Protocol (UDP) datagrams and the default UDP destination port used by the server is port 443, which is assigned to HTTPS traffic. One design goal of QUIC is the fast connection setup. If two peers have communicated before, 0-RTT connection setup could be used. The handshake and data are then sent together. If the peers do not send 0-RTT packets, the data is sent after the handshake in 1-RTT packets.

QUIC supports connection migration. An end point can change the IP address or port number and the connection will survive. Multi-homed devices are allowed to utilize different network interfaces. Connection migration is possible with QUIC since the protocol is connection oriented and use connection IDs to keep track of connections instead of 5-tuple (of procotol, source address, source port, destination address, destination port).

The packet number in QUIC packets are reduced and encoded to the size of 8 to 32 bit. The full packet number is an integer which can have a value within the range of 0 to $2^{62} - 1$. This requires the sender of a connection to encode the value and the receiver to decode. The full packet number is needed for building the nonce for the AEAD algorithm, further described in section 2.1.2.

User applications that rely on QUIC as a transport layer protocol are structured as in figure 2.1.2. The user application consists of the application code and a QUIC stack. The QUIC stack is responsible for encapsulating and managing the user application data that will be sent towards the host's peer of the connection. The QUIC stack is also responsible for packet reordering, scheduling, ACK packet processing etc. The QUIC stack also derives the crypto keys and handles the encryption and decryption.

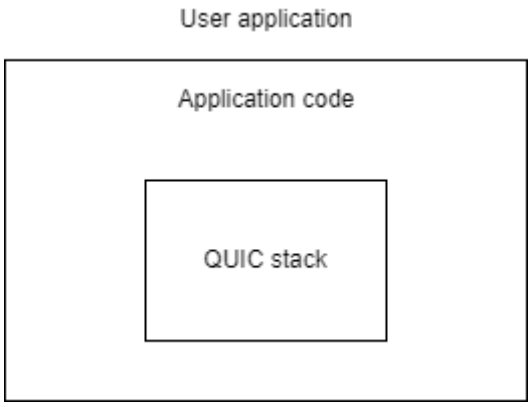

Figure 2.1.2: The structure of an application using QUIC

2.1.2 QUIC Encryption

QUIC version 1 uses TLS 1.3 for security [41]. The communication between two end points start with a TLS handshake using transport parameters and crypto frames to agree on which algorithms to use and to derive the keys. The packets are protected in two steps, first the packet protection is applied for the payload and then the header protection for specific header fields. In TLS 1.3, five different cipher suits are defined [39] and QUIC can use AES_GCM_128, AES_GCM_256, AES_CCM_128, and CHACHA20_POLY1305 for packet protection, but not AEAD_AES_128_CCM_8_SHA256. All of these algorithms encrypt the payload and produces an authentication tag of 128 bit which is used to ensure the packet has not been tampered with or modified. The header is protected by either AES_ECB if AES_GCM or AES_CCM are used for packet protection, or raw CHACHA20 if CHACHA20_POLY1305 is used. The key size used for the packet protection algorithm matches the key size used for header protection. When encrypting the header, only the reserved bits, key phase, and packet number length fields are masked. The structure of 1-RTT packets and which fields are masked can be seen in figure 2.1.3. The value in the parenthesis represent the size in bits of the field.

```
1-RTT Packet {
  Header Form (1) = 0,
  Fixed Bit (1) = 1,
  Spin Bit (1),
  Reserved Bits (2),                  //masked
  Key Phase (1),                      //masked
  Packet Number Length (2),           //masked
  Destination Connection ID (0..160),
  Packet Number (8..32),              //masked
}
```

Figure 2.1.3: 1-RTT QUIC header with masked fields marked

The packet protection key, nonce, and header protection key have static initial values

that are computed with the HKDF-Expand-Label function from TLS in the Initial packets. QUIC has different encryption levels (Initial, Early data (0-RTT), Handshake, and Application data (1-RTT)). For each level different keys are used. The keys are also replaced after a specific number of packets have been sent, to ensure security. An overview of QUIC encryption pipeline can be seen in figure 2.1.4.

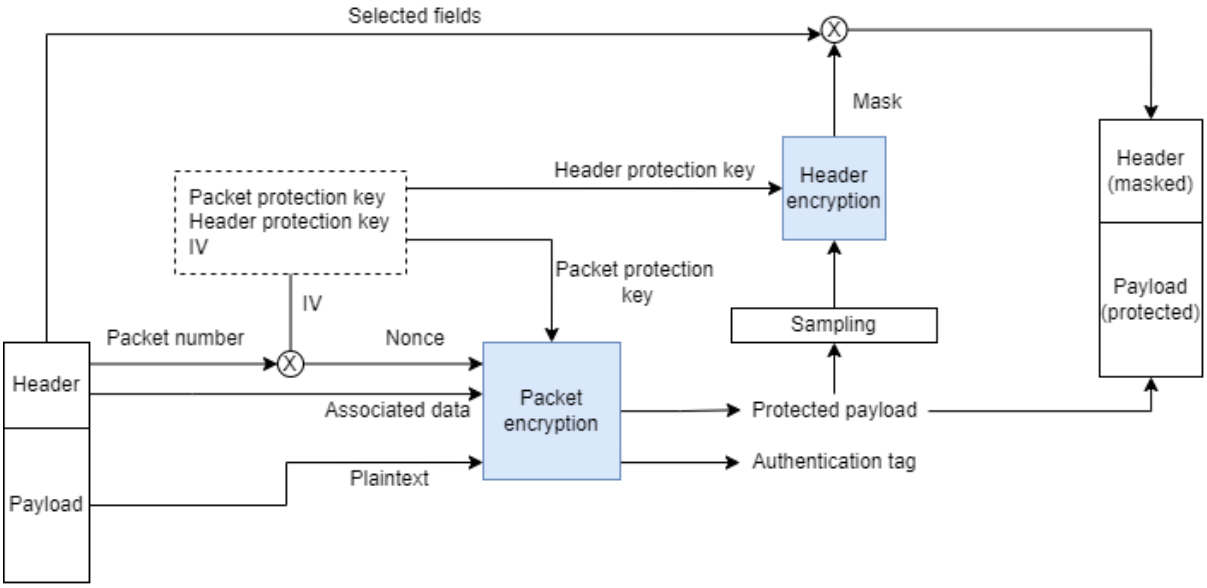

Figure 2.1.4: QUIC encryption overview

When encrypting a QUIC packet the following steps are taken:

1. The full packet number encoded using 62 bit is read in network byte order and left padded with zeros to match the size of the Initialization Vector (IV). The modified packet number and the IV, are calculated with exclusive OR to form the AEAD nonce. This step differs from how multipath QUIC builds the nonce, later described in 2.1.3
2. The nonce, plaintext (the payload), the QUIC header which is used as associated data, and a packet protection key is input to the AEAD algorithm. The output is the protected payload and an authentication tag. AES_GCM, AES_CCM, and CHACHA20_POLY1305 produces a 128 bit authentication tag.
3. The protected payload replaces the plaintext payload of the packet and is sampled into a specific number of bytes. The sample offset starts 32 bit after the start of the packet number field. AES_ECB and CHACHA20 samples 128 bit from the ciphertext. When using CHACHA20, the first 4 bytes are used as a counter and the 12 remaining bytes are used as a nonce.
4. The sampling and the QUIC header protection key is used as input to AES_ECB and CHACHA20. CHACHA20 also uses these as input together with the counter, nonce, and 32 zero bit. The output is a five byte mask.
5. The mask and selected header fields (seen in figure 2.1.3) are calculated with exclusive OR which is used as the resulting protected header.

2.1.3 Multipath QUIC

Multipath-QUIC is an extension of the QUIC protocol. Multipath QUIC has not been standardized yet and is described in an active Internet-draft [31]. The investigation and ideas of extending QUIC with multipath capabilities has been an ongoing work since 2018 [43]. The connection overview can be seen in figure 2.1.5

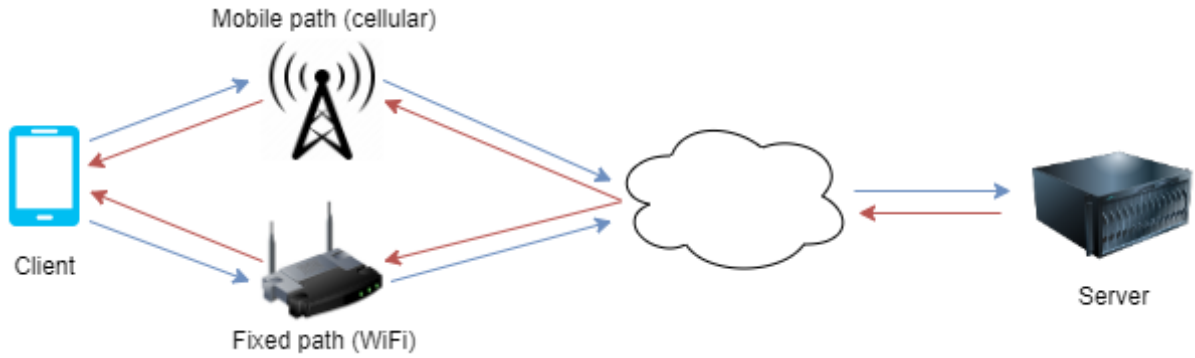

Figure 2.1.5: Connection using multipath QUIC

QUIC's feature connection migration makes multipath capabilities easier to implement. Since QUIC aims for being a flexible protocol, the QUIC extension Frames allows for changes such as multipath capabilities to be implemented. Multipath QUIC extends QUIC by the frames PATH_ABANDON and ACK_MP, which are only sent in 1-RTT packets. There is a debate regarding if multipath QUIC should be using multiple packet number spaces, i.e. one packet number space for each path, or a single packet number space for all paths [8]. In the current Internet draft [31] the suggested solution is to use transport parameters, that the peers can use to agree on which option to use. If the option 0x2 is used, multiple packet number spaces are supported, this will impact how the nonce is built during the AEAD encryption and decryption. A unique nonce is normally calculated by combining the padded packet number and IV with exclusive or. By communicating with multiple packet number spaces, the same packet number is sent in different paths. This is not acceptable since the nonce must be unique, and changes are required in the creation of the AEAD nonce. The full decoded packet number, two zero bit, and a path identifier, are instead combined with exclusive or. The packet number and the path ID are reconstructed in network byte order. The path-and-packet number combination is then combined with the IV using exclusive or. In this project, the only difference between QUIC and multipath QUIC that needs to be considered is the nonce creation. The additional frame types in multipath QUIC does not need to be taken into account.

$$\text{path-and-packet-number} = \text{path identifier} + 00 + \text{packet number}$$
$$\text{nonce} = \text{path-and-packet-number} \oplus \text{IV}$$

2.2 Frameworks and Programming Languages

Multiple vendors in the industry are developing different ecosystems and frameworks to provide services and increase flexibility for the developers. Software Development Kit (SDK)s [42] are programs and software tools, such as libraries, documentation, and guides, which can be used by developers when implementing applications. Multiple vendors offer SDKs for their network cards and silicon. Two larger ecosystems today are DOCA and IPDK. In order to limit the scope of the investigation of which frameworks, SDKs, or programming languages that can be used, the focus is on large vendors in the industry and the currently available resources for this project.

2.2.1 Data Plane Development Kit (DPDK)

DPDK [10] is an open-source framework which offers a set of libraries and drivers for specific environments and architectures. DPDK offers a programming framework for x86, ARM, and PowerPC processors. With DPDK, the application is able to get direct access to the packets received at the Ethernet port of the NIC by using Poll Mode Driver (PMD). In DPDK, packets are completely bypassing the kernel stack and placed in receive (rx) and transmit (tx) queues in the user-space. This avoids context switching or intermediate copy operations needed when using kernel space functionality. However, functionality available in the kernel such as TCP congestion control functionality must be reimplemented when using DPDK for packet processing. DPDK packet pipeline is illustrated in figure 2.2.1. In traditional packet processing, the packet is received at the NIC, and for each packet being processed by the kernel space a interrupt is thrown in order to make the CPU context switch and handle the request [30].

One of the key components of DPDK is the Environment Abstraction Layer (EAL) [12] which is a generic interface to gain access to low-level resources, such as hardware and memory space. One of the main functionalities of the EAL initialization is to launch logical cores to run the application. The initialization involves function calls to the pthread library, which is used to create multithreaded programming. Logical cores in DPDK are used to poll packets from the rx queues. Each core can poll from many queues, but it is not possible for multiple cores to poll from the same queue, since it could result in race conditions.

Another key component of DPDK is the mbuf library [11]. This library handles the allocation of memory for message buffers which are used to store the data. The data can be of different types, such as network packets, control data, or events. The mbuf library contains functionality to manipulate the data, i.e. the packet, such as appending data, removing data, and getting pointers.

DPDK offers crypto device drivers for a finite number of crypto devices [13]. This set of drivers consists of both software crypto functionality, such as AES-NI and openSSL, and hardware crypto support. These drivers allow the developer to accelerate

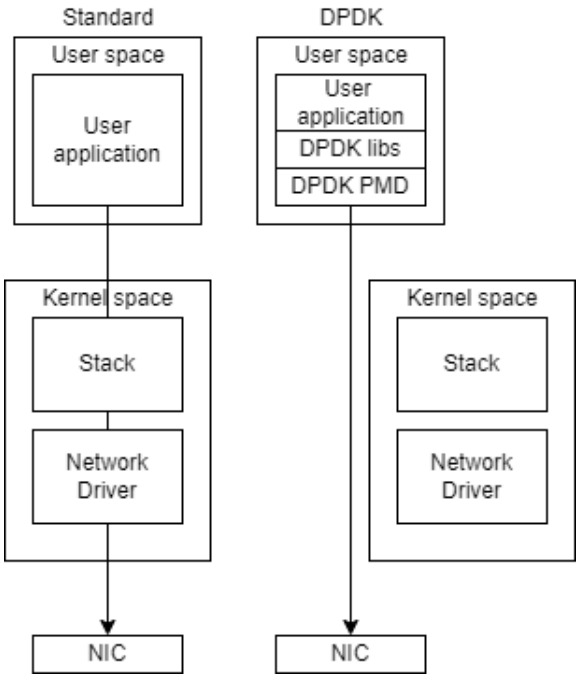

Figure 2.2.1: DPDK using PMD to handle packets received on the NIC and bypass the kernel space

crypto functionality such as cipher, authentication, AEAD, and asymmetric crypto algorithms. When running the EAL initialization, the supported hardware acceleration functionality is discovered.

For this project DPDK version 21.11 is investigated, which is the latest Long Term Stable (LTS) version.

2.2.2 P4

P4 [36] is a programming language used for programming the data plane in network devices such as switches, routers, NICs, and filters. The idea behind P4 was first published in 2014 and is now the de facto standard programming language for specifying how these devices should in real-time process the packets and how to achieve desired behaviours in the data plane. It is domain specific and compiles to a specific architecture of the target. The target is either hardware-based, such as FPGA, or programmable Application-Specific Integrated Circuit (ASIC)s, or software-based if a P4 program is running on a x86 processor. Since P4 is used to program many different devices, some functionality used in imperative programming languages cannot be used because it is not supported by some targets. These functionalities include loops, recursion, the arithmetic operators division and modulo, and dynamic memory allocation [30].

A typical P4 program consists of three parts; a parser, a match-action pipeline, and a deparser. This is illustrated in figure 2.2.2. The program begins with defining a set of headers. At the parser, the predefined headers are extracted. The parser is a

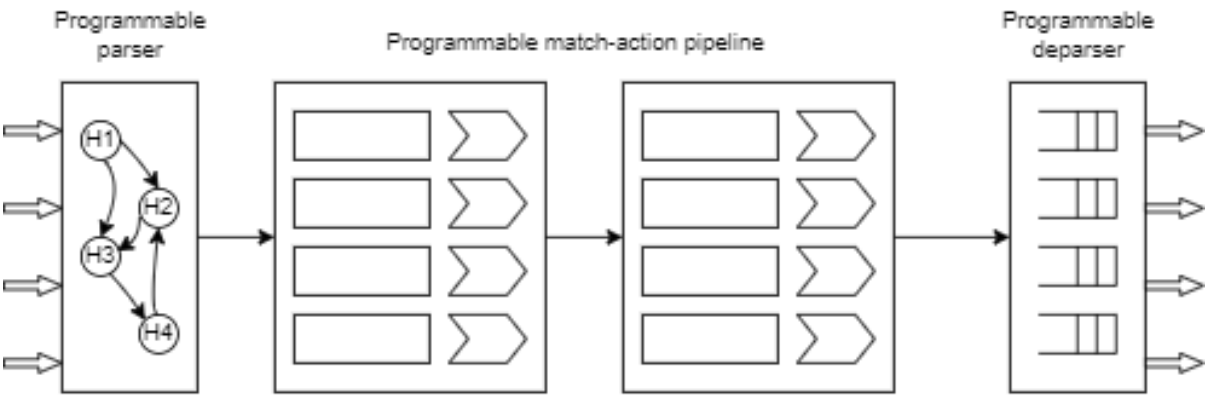

Figure 2.2.2: A typical P4 program

finite state machine and different states can be reached depending on the values of the header fields. The match-action part of the program is where the main processing is done on packet headers or packet meta data. The match-action part is divided into two parts; the ingress and the egress block, which is often used to perform different tasks. A table in P4 often consists of a matching key value and parameter values that are returned to the following action if a match has been triggered. An action in P4 is similar to a function in imperative programming languages, such as C. At the end of a P4 program is a deparser. The functionality of a deparser is to put the headers together in the specified order specified by the programmer and attach the payload at the end before the packet is serialized on the wire towards the next hop.

Externs in a P4 program are additional functions with can be used to process the packets beyond the functionality that P4 offers. When the external functions have been imported and declared, they can be used in the P4 code as regular functions. Externs can be written in for example C or Micro-C, depending on the target capabilities. There are different versions of the language with slightly different semantics and P4 version 16 is investigated for this project.

T4P4S [44] is a open-source multi-target compiler created for P4 programs to utilize a DPDK environment. The T4P4S compiler generates platform independent C code from P4 descriptions which are using Networking Hardware Abstraction Layer (NetHAL) in order to support multiple targets. By using a compiler such as T4P4S, functionality that is not supported by P4 can be added.

2.2.3 Infrastructure Programmer Development Kit (IPDK)

IPDK [20] is an open-source framework containing drivers and APIs. IPDK offers infrastructure offload for the CPU, DPU, IPU and switch. It contains different tools such as Storage Performance Development Kit (SPDK), DPDK and P4. Some use cases are Infrastructure as a service (IaaS), such as network, storage, crypto, Platform as a service (PaaS), and inline acceleration. It is target agnostic, i.e. the content of IPDK is independent of each device's functionality, the device's SDK, drivers, and compiler backend. IPDK is used as an aggregation point with scripts and containers

to centralize open source projects into a single framework. It offers two standard interfaces: an infrastructure application interface and a target abstraction interface. The infrastructure application interface is Remote Procedure Call (RPC)-based, which means procedures can be called directly while existing in a remote program and using separate memory address spaces [32]. The controlling application can be executed locally, remotely, or both. The RPCs offered by IPDK are P4Runtime, OpenConfig, Redfish / REST API, SPDK Storage Protocol, Managed Kubernetes, and Envoy xDS. The target abstraction interface is driver, library, and capabilities based. IPDK has not published any releases yet and therefore the current state available in early 2022 will be investigated.

2.2.4 DOCA

DOCA [28] is a SDK used for NVIDIA's DPU BlueField and contains drivers and a runtime and development environment within the BlueField DPU OS. By using the device drivers, the developer is provided with an interface for the hardware device. The DOCA SDK uses frameworks and APIs such as DPDK, SPDK, P4, and Linux Netlink with NVIDIA acceleration packages to achieve offload, acceleration, and isolation of workloads. Applications used for networking, security, storage, High Performance Computing (HPC), AI, telco, and media can leverage from DOCA. DOCA applications can be executed on the x86 host or on the Arm cores of the DPU. When executed on the x86 host, DPU acceleration functionalities becomes available through DOCA library calls. P4 will be a component of DOCA but is currently not available. The framework is currently in version 1.3 and was first released in April 2021. The SDK is still in early access with more functionality to come in the future. For this project DOCA version 1.3 is investigated.

2.2.5 Hardware Description Language

HDL is a textual language used to program electronic and digital logic circuits, often on FPGA cards, with words and symbols which translates into configuration data and are loaded onto the FPGA. The fundamental difference between HDL and other programming languages, such as C or Java, is that regular programming languages use sequential order while HDL uses parallel operations. Multiple parts of the hardware can operate concurrently [25]. VHDL and Verilog are the most used HDLs, where VHDL is most common since it allows designs for all types of circuits. The output of HDL is not an executable file, but a gate map which is used to examine the operations of the desired circuit [33].

2.3 Programmable Networking Hardware

Many different types of programmable hardware devices are available on the market. These hardware devices have different sets of functionality and are used for varying

purposes.

2.3.1 SmartNIC

A SmartNIC is a PCIe network device with Ethernet ports which enables connectivity to an end host, such as a server or computer. It also provides some measure of programmability, which allows for data processing used in for example accelerating data center networking, security and storage. The definition of a SmartNIC varies depending on the vendor. A SmartNIC can be based on an ASIC, FPGA, or SoC. The ASIC based SmartNIC has often a great price-performance value, but lacks flexibility due to predefined capabilities. The FPGA based SmartNIC is often expensive and extremely difficult to program, but offers flexibility since it is highly programmable. The SoC based SmartNIC offers high programmability and flexibility due to having its own onboard processor [3].

2.3.2 DPU

A DPU [9], also called a SoC based SmartNIC, is a programmable processor which can be used by itself but is often embedded into a SmartNIC card. It can be thought of as a third member of the computing devices, among with the CPU and the Graphics Processing Unit (GPU). The DPU often includes a software-programmable multi-core CPU, a network interface, and programmable acceleration engines. Some features included in many DPUs are data packet parsing, matching and manipulation, GPUDirect accelerators, TCP acceleration, and crypto acceleration.

2.3.3 IPU

An IPU offers similar functionalities as the DPU but is developed by Intel and are ASIC and FPGA based [27]. It is a programmable processor and extends SmartNIC capabilities and targets cloud and communication service providers. It is used to accelerate applications using micro services, storage and network virtualization. It is used to offload the CPU cores and shift functionality to the IPU instead [26].

2.3.4 FPGA

FPGA is a hardware device with integrated circuits, and contains combinational components, such as logic gates, multiplexers, and sequential components, also called flip-flops [18]. With enough resources, almost any digital circuit can be built. A FPGA consists of logic blocks (lookup tables), I/O blocks, and programmable interconnections. With these tools, different functionality can be implemented and offloaded, such as the encryption of QUIC. A FPGA is programmed by using HDL, see section 2.2.5, which is increasing the complexity of using these devices compared to other devices such as ASIC cards. FPGAs are used for different purposes, such as aerospace, defense, data centers, industrial, medical, video and image processing and

wireless communication. A FPGA can be configured to perform simple gate logic to extremely complex functionality [45].

2.4 Related Work

There is a lack of studies done on offloading multipath QUIC packet processing. There are several studies on offloading, specifically Internet Protocol Security (IPSec). In this section, some of them are discussed and how they relate to this degree project.

2.4.1 QUIC Offloading

In [46] different QUIC protocol implementations (before QUIC was standardized) was analyzed. The QUIC versions used in this project were Quant, Quicly, Picoquic, and Mvfst. All of these versions complied with the latest IETF QUIC draft at the time. The goal of the project was to investigate which parts of the different QUIC implementations required more CPU usage and would therefore benefit from NIC offloading. For all QUIC versions crypto, connection setup and teardown, ACK and packet processing, packet I/O, and header formatting was investigated. The results showed that crypto and packet I/O utilized the CPU for almost all of the implementations. Crypto required 10 to 43 percent of the CPU time on the server side, while packet I/O was responsible for 35 to 65 percent. ACK and packet reordering caused a high CPU usage in some versions, while connection setup/teardown and header formatting required less than 5 percent of the CPU time for all implementations. By analyzing the different QUIC implementations the main findings were that data copy between user space and kernel space requires high CPU usage, and kernel-bypass techniques should be used. If kernel-bypass techniques are used, the crypto operations becomes the next CPU demanding operation. In the proposed architecture for offloading QUIC, AEAD and packet reordering should be moved into hardware, while the control operations, such as the TLS handshake should remain on the CPU to keep expensive stateful processing. This study contribute with a design and measurements. These measurements are used in this project to determine what functionality should be offloaded with the prototype.

In 2020, Intel presented a paper where QUIC was accelerated via hardware offloads. In this study, Intel used a socket interface, in order to offload QUIC similar as other transport protocols that are implemented in kernel space can benefit from. The contribution by this paper was a interface, which implements a new upper layer socket protocol. By using this protocol, QUIC is able to use socket options to enable hardware offload and manage security association (SA). En/decryption (AES-GCM) and UDP segmentation was hardware offloaded. By using the interface and new upper layer socket protocol to achieve crypto offloading on the Chromium stack, the CPU utilization was decreased by 13 percent and the throughput increased by 13 percent when using a 100MB files compared to software implementations using dedicated AES instructions. The CPU utilization was reduced 16 percent and throughput increased 32

percent when using this setup with 50MB files for crypto and UDP offload. This study enlightens the benefits of offloading QUIC functionality with hardware. A challenge is to decide how the crypto and segmentation parameters should be transferred to the hardware, and in this paper the presented interface will be responsible for these tasks [23]. In our project, the socket interface will not be used. The prototype will be running in the user space with kernel-bypassing techniques. This degree project faces the same issue with transferring crypto parameters to the offloading hardware. In our work, signaling packets are design and implemented in order to transfer this information to the prototype.

2.4.2 IPSec offloading

Multiple earlier studies have been conducted on IPSec offloading and/or acceleration. IPSec is a security protocol which also requires the heavy crypto operation AEAD, similar to TLS which QUIC uses. In [16] P4-IPSec was offloaded on both the software switch BMv2 and a Edgecore Wedge 100BF-32X Tofino-based hardware switch without a crypto unit using P4. The NetFPGA SUME platform was also used but due to limitations in the P4-NetFPGA environment, which the NetFPGA SUME reference switch uses, an implementation could not be provided. P4 on this architecture does not support parsing variable-length header fields or data exchange between the P4 pipeline and externs. When using the Edgecore Wedge Tofino-based switch, two implementations were presented. In one of them, the P4 extern functions was replaced with IPSec kernel functions of the Linux OS and a IPSec crypto manager program running in the user space. In the other implementation, the same kernel and user space functionality was used but with the P4 program forwarding the IPSec flows based on rules in the match-action tables.

In our work, design of the prototype using P4 is presented, which is similar to the P4-IPSec design. The key difference between these designs is that QUIC will be processed and not IPSec. The packet processing also differs, since the packets sent and received in P4-IPSec are IP packets. In the prototype presented in this thesis, the goal is to offload functionality from the user application's QUIC stack running on the host.

Chapter 3

Design of offloading application

In this chapter, the framework and hardware decision is first discussed in section 3.1. In section 3.2, a high level design overview of the offloading application and the packet designs are presented. In section 3.3, a design using DPDK is presented followed by a design using P4 in section 3.4. The P4 pipeline presented in this thesis is only a reference design and will not be implemented due to time constraints.

3.1 Framework and hardware selection

For this project different frameworks and hardware devices can be used to achieve the goal of implementing an offloading application for multipath QUIC. In this section, the frameworks, programming languages and hardware devices from section 2.2 and 2.3 are investigated to determine which solution is more feasible to implement. First, the framework or programming language is selected, and then a decision is made on which hardware device to use which supports the chosen framework.

3.1.1 Frameworks

First, IPDK was investigated, since it is a new framework this degree project would provide as an example of a use case to the community and to Ericsson. The investigation proved quickly that IPDK was not mature enough to be used for this project. No first stable release has been submitted yet and the Github repository is frequently updated. In the Slack channel[1] for IPDK, which consists of the community and developers, daily requests and discussions are made on changes in the framework. During the open programmable infrastructure event [21], Intel presented the roadmap for IPDK which showed security functionality being planned for in the future. Due to the lack of functionality supported and no stable release has been submitted, IPDK was not selected.

[1] https://ipdkworkspace.slack.com/archives/C02D9SPPFH8

The framework DOCA was then investigated. According to NVIDIA's blog, DOCA will have support for P4 and TLS encryption, but in the current state it is not supported. DOCA is less than a year old, and the only crypto operation supported at the time is key generation. After a discussion with developers at NVIDIA it was clear that the framework is not ready for supporting a QUIC encryption hardware offloading application. There are plans for future NICs to support QUIC implementation on hardware. DOCA was not selected since the crypto functionality needed is not supported, yet.

The DPDK framework investigation resulted in DPDK was considered a feasible framework to use for implementing the QUIC offloading application. It is much more mature and offers stable release in comparison to the other frameworks. DPDK supports all AEAD algorithms QUIC uses, through different crypto device drivers. The limitation of DPDK offloading and acceleration capabilities depends on the hardware device used. The investigation resulted in a offloading prototype using DPDK was designed and later implemented.

P4 does not support the crypto operations needed, resulting in the need of external functions. The DPDK compiler T4P4S can be extended with external functions, to support crypto functionality. By using the software switch bmv2, software libraries that support AEAD such as openSSL, could be used. A offloading prototype using P4 was designed but was not implemented since a large part of the implementation would have to rely on external functions, for example through extending the T4P4S compiler. The offloading application design using P4 (see section 3.4.2) is similar to the design using DPDK (see section 3.3). Since the design using P4 might have to rely on multiple functionality provided by DPDK, the decision to use native DPDK was made.

3.1.2 DPDK crypto offload drivers

In table 3.1.1 the different crypto drivers that supports AEAD in DPDK can be seen. QUIC uses the cipher suits AES_GCM_128, AES_GCM_256, AES_CCM_128, and CHACHA20_POLY1305. In order to determine which driver to use the hardware had to be chosen. NVIDIA's BlueField 2 DPU is the hardware device selected for this project, later motivated in section 3.1.3.

Out of the listed drivers in table 3.1.1, the OpenSSL Crypto PMD was selected. This decision was made due to issues getting access to hardware devices with crypto engines that is supported by DPDK's crypto drivers. By using a software crypto driver, the number of the hardware devices which can be used for running the QUIC offloading application is increased. The crypto device is an input parameter to the program, which enables high portability. When running on a hardware with a crypto engine supported by DPDK, this input parameter is changed without modifying the code.

Table 3.1.1: Crypto devices in DPDK that support AEAD algorithms used in QUIC

	aesni-gcm	aesni-mb	bcmfs	caam-jr	ccp	chacha20-poly1305	cn10k	cn9k	dpaa-sec	dpaa-sec2	mvsam	nitrox	octeontx	openssl	qat
AES_GCM_128	Y	Y	Y	Y	Y		Y	Y	Y	Y	Y	Y	Y	Y	Y
AES_GCM_256	Y	Y	Y	Y	Y		Y	Y	Y	Y	Y	Y	Y	Y	Y
AES_CCM_128		Y	Y											Y	Y
CHACHA20_POLY1305		Y				Y	Y	Y							Y

3.1.3 Hardware

The hardware devices which could be used for this project were selected after the decision on using DPDK was made. The reason why the framework or programming language was selected before the hardware, was due to the fact that hardware devices are more common to replace since hardware are upgraded at a higher frequency than frameworks or programming languages. The requirement for the chosen hardware device was that it must be able to run the framework or programming language that was previously chosen.

For this project, NVIDIA's BlueField 2 DPU was selected as hardware device. The BlueField 2 DPU has support for hardware accelerating AES_GCM_128 and AES_GCM_256 but the crypto engine is only accessible through the kernel and not by the DPDK framework. Therefore, the prototype will not leverage from hardware accelerated crypto support on the BlueField 2 card but instead use the software library OpenSSL seen in table 3.1.1. Intel's Mount Evans IPU was considered, and Ericsson had access to a card which could have been used for the prototype. Due to the card not being released yet during the time of the project it was considered a major risk of using a card which might result in errors caused by other factors than the prototype application.

A FPGA card was not chosen due to DPDK being the most feasible solution. FPGA cards are much more complex to program than other cards, due to the use of HDL. A SmartNIC could have been chosen if P4 was used, or if the goal was to only run DPDK on the x86 host. Since one goal of this project is to offload the encryption from the CPU, one solution is using DPDK on ARM cores exclusively.

The BlueField 2 card will be used, which offers the three modes separated host mode, embedded function, and restricted mode, which is an extension of embedded function. In the separated mode, the SmartNIC is a standalone device and uses its

Table 3.1.2: Comparison of hardware devices

Hardware devices	Mount Evans [6]	BlueField 2 [34]	Octeon CN10 [35]
Type	IPU	DPU	DPU
Manufacturer	Intel	Nvidia	Marvell
Framework	DPDK, IPDK	DPDK, DOCA	DPDK
Release year	Q2 2022 (est.)	2021	-
Crypto	supported	supported	supported
CPU	Arm (Neoverse N1)	Arm (Cortex-A72)	Arm (Neoverse N2)
Max Frequency	3GHz	2.5GHz	2.5GHz
Cores	16	8	24

ARM processor, without the x86 processors interference. The x86 can use the NIC at the same time as usual. The Media Access Control (MAC) addresses are used for separating if the packet is to be received in the SmartNIC by the ARM processor or by the host using the x86 processor. In the embedded function mode, the packets are sent via the ARM processor to the x86. The card has an embedded switch which routes the packets internally. In this project, the BlueField 2 DPU will be used with the embedded function mode.

3.2 Design overview

In this design, a host and a peer are communicating using multipath QUIC. Multipath QUIC is the transport protocol for the communication, but the packet format is identical with QUIC packets, except the additional frames in the payload. Therefore, the multipath QUIC packets will be referred to as QUIC packets in this thesis, and the nonce will be created according to multipath QUIC and not QUIC.

An overview of the communication flow between a host and a peer, with two offloading applications in-between can be seen in figure 3.2.1. The user applications first sends packets to set up the communication session with each other. In this step, the TLS handshake is performed, parameters are exchanged, and the peers know what destination connection ID should be used to address each other. This process is the same as how QUIC normally sets up a connection. The offloading applications do not interfere in this stage, but simply forward the packets. When this is done, they have all the information needed to start communicating with packets containing user application data, i.e. 1-RTT packets. Before sending 1-RTT packets, the user applications share information from the communication setup with the offloading applications. This information consist of for example the crypto keys the user applications have agreed to use, path IDs, and destination connected IDs for its peers. The details of the signaling packets are described in section 3.2.1. The purpose of these signaling packets is for the offloading application to know how to process the 1-RTT QUIC packets. After sending this signaling information, the user applications shift to using 1-RTT QUIC packets. When these packets are forwarded from the QUIC stack on the user application to the offloading application, they consist of an Ethernet header (to

route the packet), metadata, a QUIC header, and the payload from the user application for its peer. The offloading application encapsulates this packet into an IP packet by adding IPv4 and UDP headers, encrypts the QUIC header and payload, and sends it on the wire towards the peer. The offloading application at the peer decrypts the QUIC header and payload, decapsulates the packet into the same format sent by the original host, and sends it to the user application on the peer.

By having this design, the crypto is offloaded from the CPU of the host, which decreases the CPU utilization and the host can perform more critical operations instead. If the card which the offloading application is executing on have a crypto engine, the throughput of the crypto operations might be increased. By sending less overhead per packet between the user application and offloading application, the CPU utilization on the host might be decreased further since less data has to be copied between kernel and user space when a packet is sent or received, unless kernel-bypass techniques are used. This proposed design is beneficial for data centers since the CPU on the servers can spend more time performing business critical tasks.

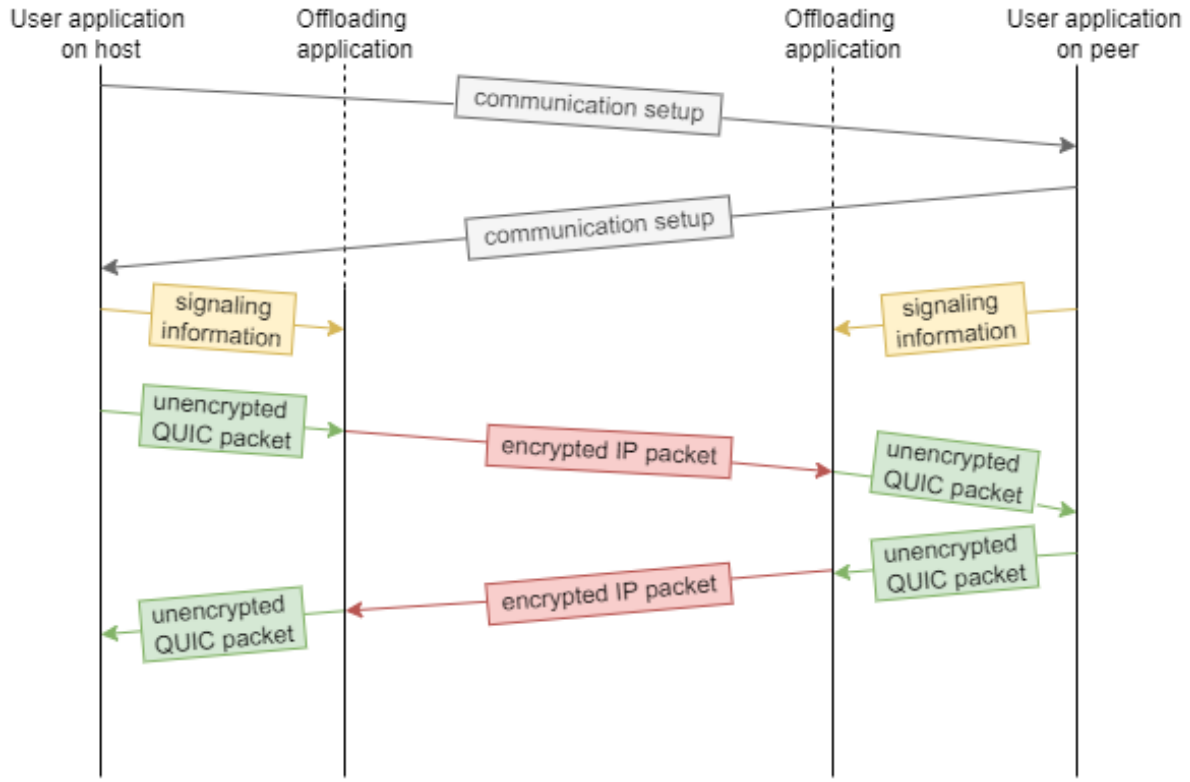

Figure 3.2.1: The communication flow

A more detailed view of the the communication between the user application and the offloading application can be seen in figure 3.2.2. The offloading application is connected to the host using PCIe. A user application is running in the user space of a host. The user application communicates using multipath QUIC and therefore has a QUIC stack. The QUIC stack handles the majority of the QUIC connection functionality. The functionalities performed by the QUIC stack are for example multipath packet scheduling, packet reordering, key generation, and creating of QUIC

packets. It is the QUIC stack's responsibility to craft QUIC packets, i.e. packets with a QUIC header and payload from the user application. The user application must also attach metadata and an Ethernet header addressed to the offloading application in front of the QUIC packet when sending and remove the Ethernet header when receiving a packet.

The user application does not create the IP packets, it will only create a QUIC packet with an Ethernet header to send the data to the offloading application. The QUIC stack operates normally but can choose to send the packet through the application for encryption and encapsulation without having to modify any previous steps in its processing. The QUIC stack on the user application is considered a control plane in this prototype, which means it will control how the data should be processed and forwarded. The offloading application is the data plane, which is where the packets are being processed and forwarded using the rules provided by the control plane.

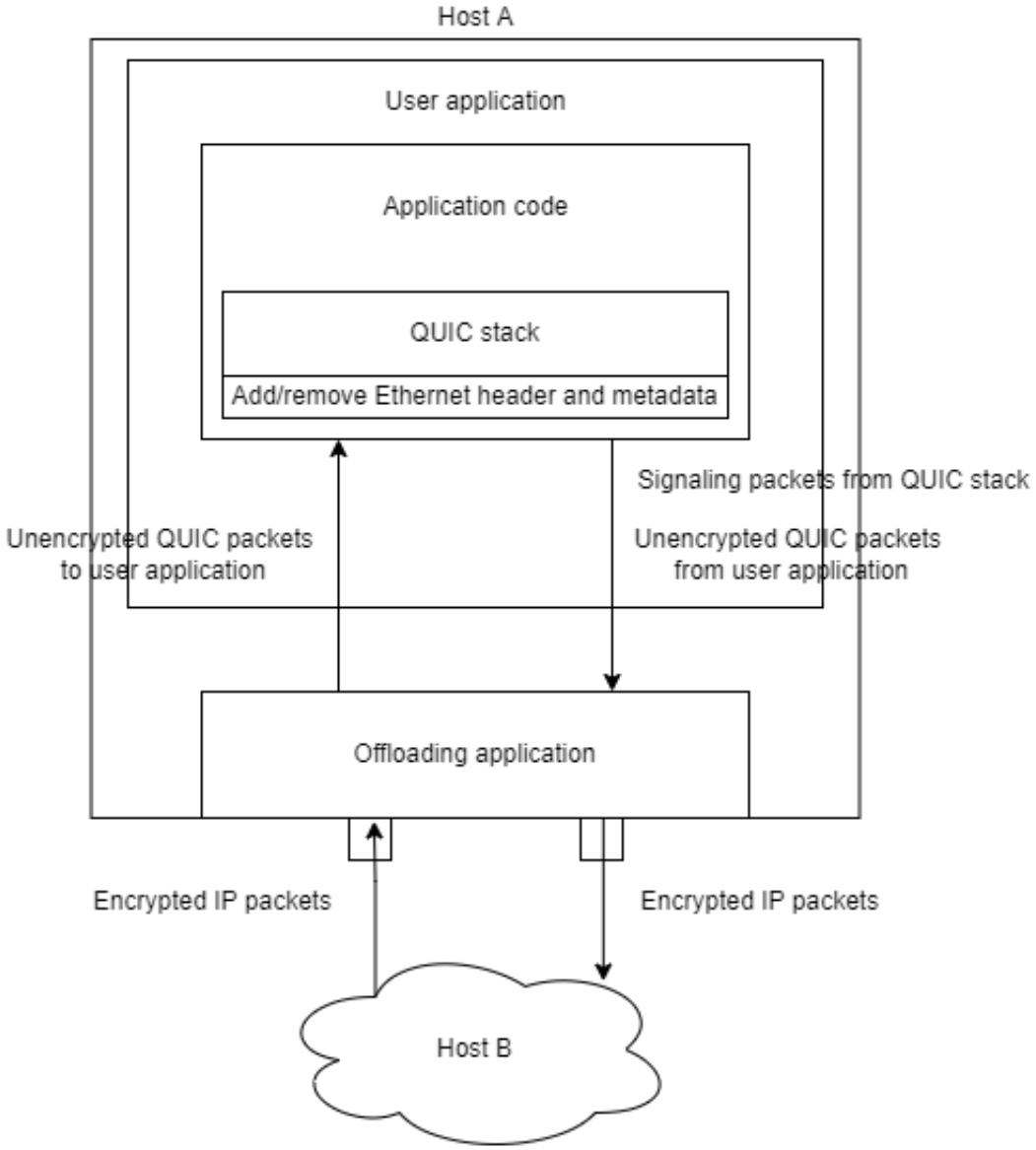

Figure 3.2.2: An overview of the offloading design

3.2.1 Packet format design

In this project four types of packets will be received by the offloading application. One type is packets sent from the peer and the remaining three types are packets sent from the host. Two of the types sent from the host are used for signaling and one for sending user application data. The signaling packets are used to provide the application with information necessary for encryption and decryption. There are two types of signaling packets since the application needs different parameters depending on if the packets are sent from the host and will be encrypted or sent from the peer and will be decrypted. The different types of packets the application can receive can be seen in figure 3.2.3.

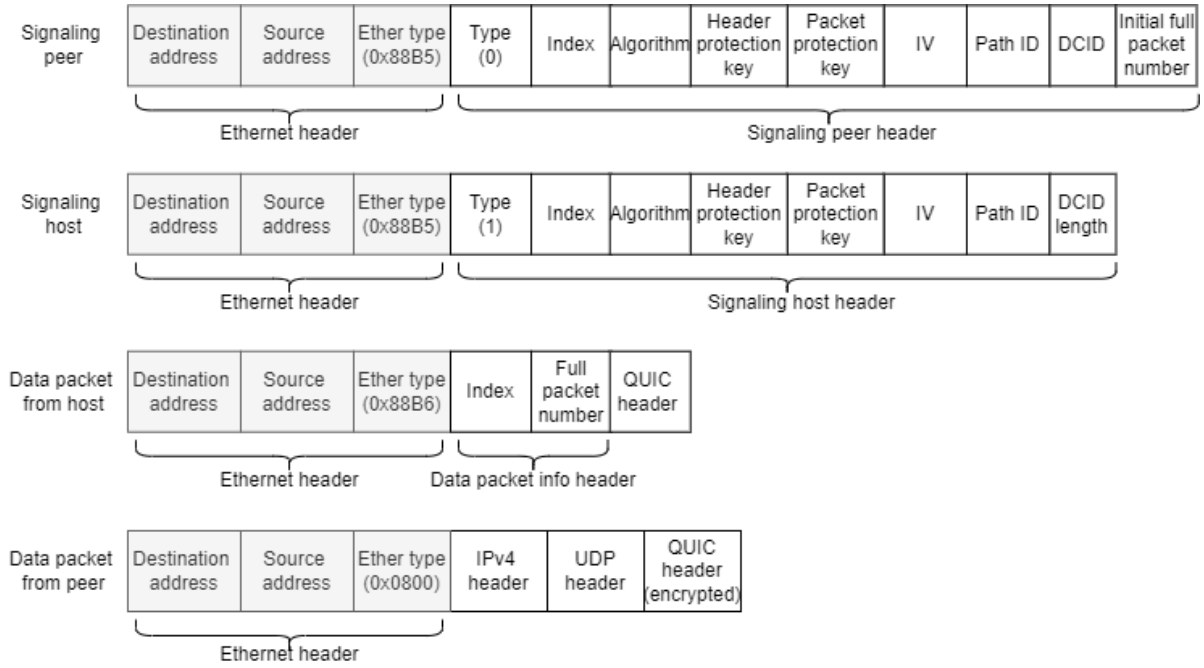

Figure 3.2.3: Four different packet types that can be received by the offloading application

Packet type *Signaling peer* consists of a Ethernet header where the *Ethertype* field is set to 0x88B5 which is a local experimental ethertype. After the Ethernet header, a signaling header follows. The header consists of the following header fields: type, index, algorithm, header protection key, packet protection key, IV, path ID, destination connection ID, and the full packet number. This type of packet is used to provide the application with information for connections when packets are sent from the peer to the host. The information is stored in a table in the application and the packet is discarded.

Packet type *Signaling host* is used to provide the application with information of the connections for packets sent from the host to the peer. The information is stored by the application in a table and the packet is discarded after being read. The packet consists of an Ethernet header with the *Ethertype* field set to 0x88B5, and the signaling host header which consisting of a type, index, algorithm, packet header protection key,

protection key, IV, path ID, and the destination connection ID length.

Packets of type *Data packet from host* is data packets from the user application with a Ethernet header with the *Ethertype* field set to 0x88B6, and a data packet info header which consist of an index and the full packet number. A QUIC header and the user application payload is attached at the end of the packet. These packets are encrypted and sent towards the peer of the connection. The packets are also transformed into IP packets as they need to be able to traverse the network. After being processed by the offloading application, the packets will have an Ethernet, IPv4, UDP, and masked QUIC header followed by the encrypted payload. The Ethernet field *Ethertype* will be set to 0x0800 to represent IPv4. The IPv4 *protocol ID* header field is set to 17 to represent UDP.

The forth packet type *Data packet from peer* is the packets sent from the peer to the host. These packets contain the Ethernet, IPv4, UDP, and masked QUIC header and the encrypted user application payload at the end. The QUIC header and payload is decrypted and the packet is decapsulated by the application. The Ethernet header field *Ethertype* is set to 0x88B6. The IPv4 and UDP header is removed, resulting in a decrypted packet consisting of a Ethernet and QUIC header with payload, similar to the packet *Data packet host* type. The packet is sent to the host.

The offloading application can send two types of packets, seen in figure 3.2.4. The *Data packet to host* packet type is used to send packets to the host. The original *Data packet from peer* packet has been sent from the peer to the offloading application. The offloading application has decrypted and decapsulated the packet and it is sent in this format towards the user application on the host from the offloading application. The other packet type sent from the offloading application is the *Data packet to peer*. In this case the original *Data packet from host* is received by the offloading application, encapsulated, encrypted and sent towards the user application on the peer.

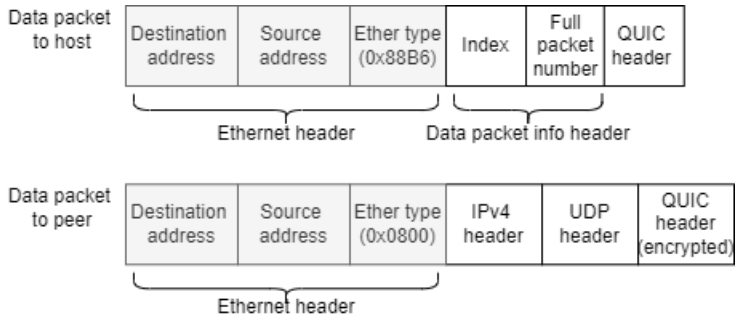

Figure 3.2.4: Two different packet types that can be sent from the offloading application

3.3 Offload application design using DPDK

In this section, the offload application design using DPDK is presented. The data plane processing pipeline is first explained, followed by a detailed explanation of the

encryption where the input and output of the crypto algorithms are described. The tables used by the data plane are explained in the control plane section.

3.3.1 Data plane design

An overview of the offloading application design using DPDK can be seen in figure 3.3.1. Two ports are bound to the application. When a packet is received, the port ID of these ports are used to determine if the packet is sent from the host or the peer. The packets are placed in two separate processing queues depending on which port the packets are received on. The packets that are sent from the user application on the host and will be sent to the peer and the packets that are used for signaling are received on port *port_enc* and the packets that are sent from the peer and will be decrypted are received on the port *port_dec*. Two different threads are used, where each thread is responsible for either the encryption pipeline or the decryption pipeline.

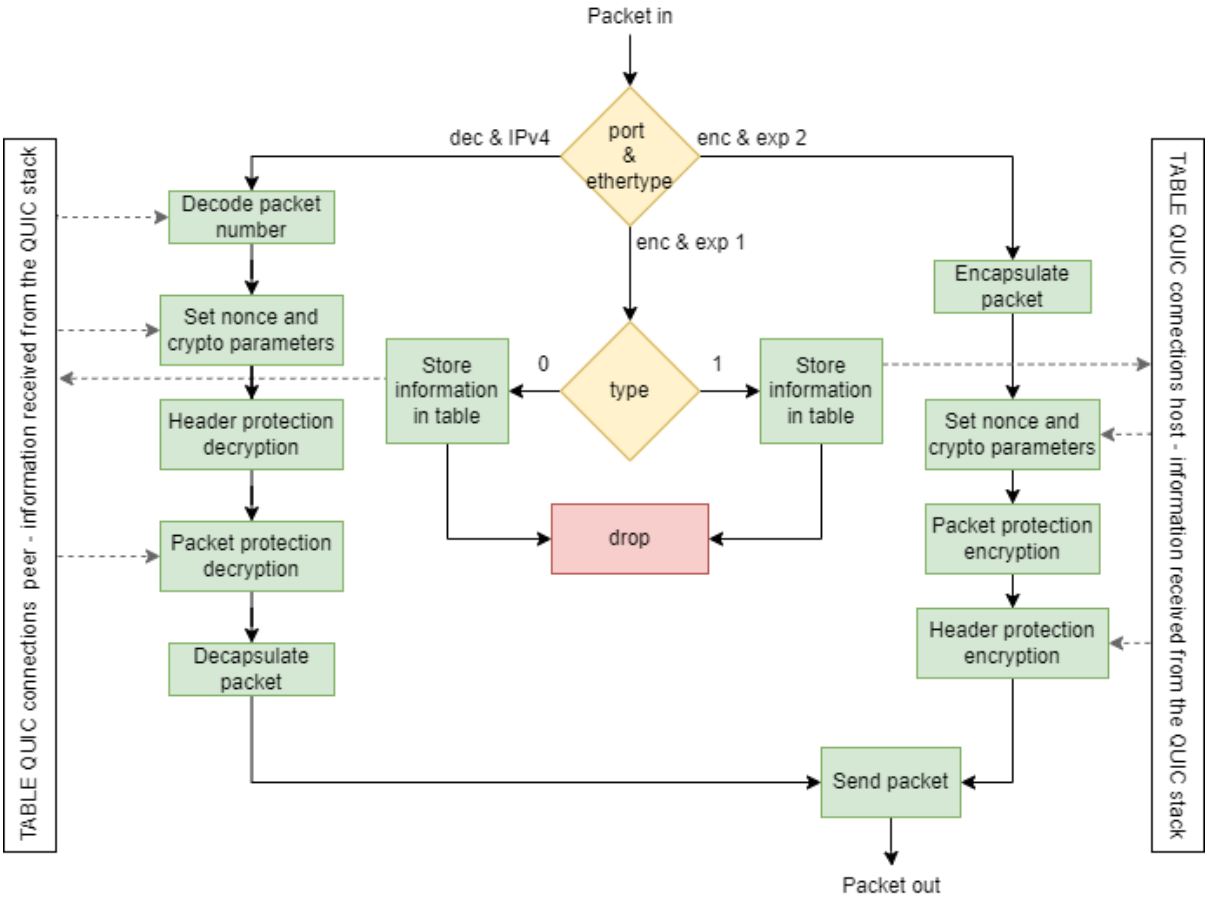

Figure 3.3.1: The data plane processing pipeline

The first step in the data plane is to determine if the packet is received on the *port_dec* and if the packet's Ethernet field *Ethertype* is set to 0x0800. If these two conditions are true the packet might be from the peer and the destination connection ID will be retrieved from the packet. A lookup is made in the hash table *QUIC connections peer* using the destination connection ID as a key. If no match occurs, the packet will be

dropped and the program will continue processing the next packet in the queue. If a match occurs, the information stored in the hash table at that entry is retrieved. The information retrieved is further described in section 3.3.3.

The header protection function is called to decrypt the QUIC header. Depending on the information retrieved from the hash table, different parameters are used for the decryption. The QUIC header can be decrypted using all the supported algorithms, i.e. AES_ECB_128, AES_ECB_256, and CHACHA20. These algorithms will not encrypted or decrypted using the crypto engine in DPDK, since it would not result in a performance gain. The header protection is instead decrypted using the openSSL library directly. The 32 bits output mask from the decryption function is used to unmask the selected QUIC header fields (seen in figure 2.1.3). The first byte unmasks the first byte of the selected QUIC header fields using exclusive or, and the packet number is unmasked by using the remaining 4 bytes of the mask.

After the QUIC header has been decrypted, the preparations for the AEAD payload decryption begins. The parameters set for this operation depends on the information retrieved from the table. Each packet which will be processed by the crypto device in DPDK needs to have a crypto operation attached. The crypto operation sets some parameters for the crypto device, such as the associated data, the length and start of the payload and physical addresses. The crypto device also needs a symmetric crypto transform structure as input for configuration, which is where parameters such as the algorithm, key, nonce length is set. After setting some of these parameters the process of building the nonce begins. The packet number received in the QUIC header is encoded in 0 to 32 bits, which is why the full packet number needs to be decoded since it is used to build the nonce to be used as input to the AEAD algorithm. The sample packet number decoding algorithm found in RFC9000 in Appendix A.3 [22] (which also can be seen in listing 4.4) is used for this. The full packet number is decoded into 62 bits but stored in a 64 bit variable, to create the two 0 bits needed for the nonce. This variable is set in network byte order and the 32 bit path ID, which is also in network byte order, are combined to create the path-and-packet-number variable. This variable is combined with the IV which is stored in the hash table by using exclusive or. The packet along with its crypto operation is queued at the crypto device, and when an arbitrary number of packet are placed at the queue, the packets are sent to the crypto device for processing.

If the packet on the other hand is received on the *port_enc* in the first data plane processing pipeline and the packet's Ethernet field *Ethertype* is set to 0x88B5, the packet is used for signaling from the host to inform of a new connection path. The first byte after the Ethernet header is examined. If the byte has the value 1, it is a signaling packet to indicate information of a new peer have been added and the information used for this peer needs to be stored. If the byte has value 0, it is an indication that the host has a new connection with parameters to store. When *type* is set to 0 the information received is stored in a hash table *QUIC connections peer*. All keys sent in the signaling packets are encoded in 256 bits. If the header protection and packet

protection keys are only 128 bits the rest of the field is padded with zeros. The padding of zeros are moved from the left to the right of the value when stored in the tables of the offloading application. The packet is then dropped from the queue and processing of the next packet begins. If *type* is set to 1 the packet is processed in the same steps but the information is stored in a table which consists of structures instead. The index retrieved from the packet is used to determine which index should be used to store the information in the table QUIC connections host.

A packet received in the *port_enc* in the first step of the pipeline can also have the Ethernet header field *Ethertype* set to 0x88B6. In this case the full packet number and index is retrieved from the packet before manipulating the packet. The Ethernet header and the data packet info are removed and the packet is prepended with a new Ethernet, IPv4 and UDP header. The fields of the headers are populated. The 16 bits index read from the original packet is placed in the two first bytes of the packet, in order to access information needed for header protection after the packet protection is applied. The nonce is created using the same process as described above for the packets from the peer. In this case the full packet number is received from the host in the original packet it received and does not need to be calculated. The packet is placed in the queue at the crypto device along with its attached crypto operation and when the queue reaches an arbitrary number of packets they are sent to the crypto device for processing.

The user application packets from the host and from the peer are dequeued after being processed by the crypto device and the payload have been encrypted or decrypted. The dequeued packets are placed in a new queue for some further processing before being sent to a tx queue and transmitted either to the host or peer. Each packet is investigated to examine if the crypto operation returned a successfully value, otherwise the packet is dropped and the next packet is processed. If for example a decrypted packet returns authentication failed, the authentication tag at the end of the packet does not match the one created during the decryption process and the packets might have been tampered with or modified.

If the packet was originally received on *port_enc* and have successfully been processed by the crypto device, the header protection is applied. The index placed in the first two bytes of the packet is used to look up information for the header encryption in the QUIC connections host table. After the header has been encrypted, the index bytes are removed from the packet. The checksums and lengths of the protocols UDP and IPv4 need to be calculated before being sent towards the peer, otherwise the packet risks being dropped. At the initialization of the offloading application, a check was made to see if the hardware supports offloading UDP and IPv4 checksum. If these checksum calculations are supported by hardware offloading, this feature will be used after the packet has been processed and is ready to be sent out on the wire.

If the packet one the other hand was originally received on *port_dec*, the IPv4 and UDP headers are removed and metadata is added. The metadata contains the index and full decoded packet number, and is sent to the host to keep the host from computing the

full packet number again. The Ethernet header field *Ethertype* is set to 0x88B6 and the packet which now only contains the Ethernet header, metadata, the unmasked QUIC header and the decrypted payload and is sent on the wire towards the host.

3.3.2 Encryption

Encryption can be performed in two different ways using DPDK. If the encryption is done inline, packets are encrypted as a bump on the wire. This process results in packets being encrypted during a transmit or receive operation. Patterns in packets are used to map to actions that should be performed upon a match. Different patterns and actions can be chained together. For example, if the packet has a specific IP destination address and UDP source port, it can be encrypted and then afterwards sent to a certain port. This patterns-actions method is similar to how match-action tables in P4 are used. The other way to encrypt in DPDK is called lookaside, which is the traditional way to achieve acceleration. By using lookaside in DPDK, packets are queued for processing by a crypto device, and dequeued after the operation is done. In the version 21.11 of DPDK that we are using, there are currently no support for inline encryption, unless the packets are of one of the protocols IPSec, macsec, pdcp, or docsis. Since inline encryption is not possible for this project the design will be using the lookaside method.

The header protection algorithms, i.e. AES_ECB and CHACHA20, will not be encrypted using a crypto device in DPDK. This decision is made because of the inefficiency which would occur when the encryption mode does not require much CPU utilization. To place the packets in a queue for encryption, sent them to the crypto device, process them, and then dequeue them would require more processing than to encrypt or decrypt the packets directly using software crypto libraries such as openSSL.

The AEAD algorithms AES_GCM, AES_CCM and CHACHA20_POLY1305 takes four inputs and produces two outputs. The inputs are the packet protection key, a nonce, the payload/cipher, and associated data which in this case is the QUIC header. The algorithms produces ciphertext or plaintext and an authentication tag as output. The key and the IV (used for building the nonce) for these algorithms can be found in the tables used to store the signaling information. The host also sends which algorithm the connection uses, i.e. AES_GCM128_AES_ECB, AES_GCM256_AES_ECB, AES_CCM128_AES_ECB , or CHACHA20_POLY1305. In DPDK, a crypto operation is attached to each packet which will be processed by the crypto device. The values of the crypto parameters, such as key length, depends on the value of the algorithm stored in the table. If for example the algorithm choice in the table equals AES_GCM128_AES_ECB or AES_CCM128_AES_ECB the key length will be set to 128 bits. The output of the AEAD algorithms are ciphertext or plaintext, and an authentication tag of 128 bits.

The input to the header protection algorithms AES_ECB are the header protection key

and the sample from the ciphertext produced by the packet protection algorithm. The CHACHA20 algorithm requires a counter, nonce, 32 0 bits, and a header protection key as input. The header protection key is retrieved from the tables. The output is 32 bits ciphertext, which are used to mask certain fields of the QUIC header.

Each packet needs to be padded to match the block size for the chosen AEAD algorithm during encryption. For this step, the length of the payload or ciphertext of each packet is investigated. If the length is not a multiple of 128 bits, which is the length of AES_GCM the packet is padded with zeros.

3.3.3 Control plane design

In this project, the signaling information created by the QUIC stack which is sent from the host is considered the control plane information. The QUIC stack derives the information used by the offloading application. The control plane is responsible for giving the data plane the necessary information for the communication to process the packets correctly. The control plane can send the two types of signaling packets described in section 3.2.1 to the application.

The *signaling host* packets, that are used when processing packets received from the host, are stored in a table of structures called *QUIC connections host* and contains an index, the crypto algorithms used, a header protection key, a packet protection key, an IV, a path ID, and the destination connection ID length. Since the destination connection ID length of the peer can be 0 to 160 bits the length is stored in the table. The table uses the index as key, since it can easily be sent from the host attached to each packet and is of known length.

The control plane information for processing *signaling peer* packets received from the host is stored in a hash table called *QUIC connections peer* and contains an index, the crypto algorithms used, a header protection key, a packet protection key, an IV, a path ID, the destination connection ID, and the full packet number. The destination connection ID length for the host is always set to 64 bits. The full packet number usually starts at 0 for a new connection, but this design allows the peers to start the full packet number at an arbitrary number. The hash table uses the destination connection ID as a keys to identify entries, since this information is a part of each user application packet sent from the peer.

Figure 3.3.2 illustrates the two tables used for each direction.

The tables consists of the following fields:

- **Index** - Used to identify the connection
- **Algorithm** - The algorithms used for encrypting or decrypting the payload and masking the QUIC header. Can be set to AES_GCM128_AES_ECB, AES_GCM256_AES_ECB, AES_CCM128_AES_ECB, or CHACHA20_POLY1305.
- **Header protection key** - The key used to encrypt or decrypt the QUIC header.

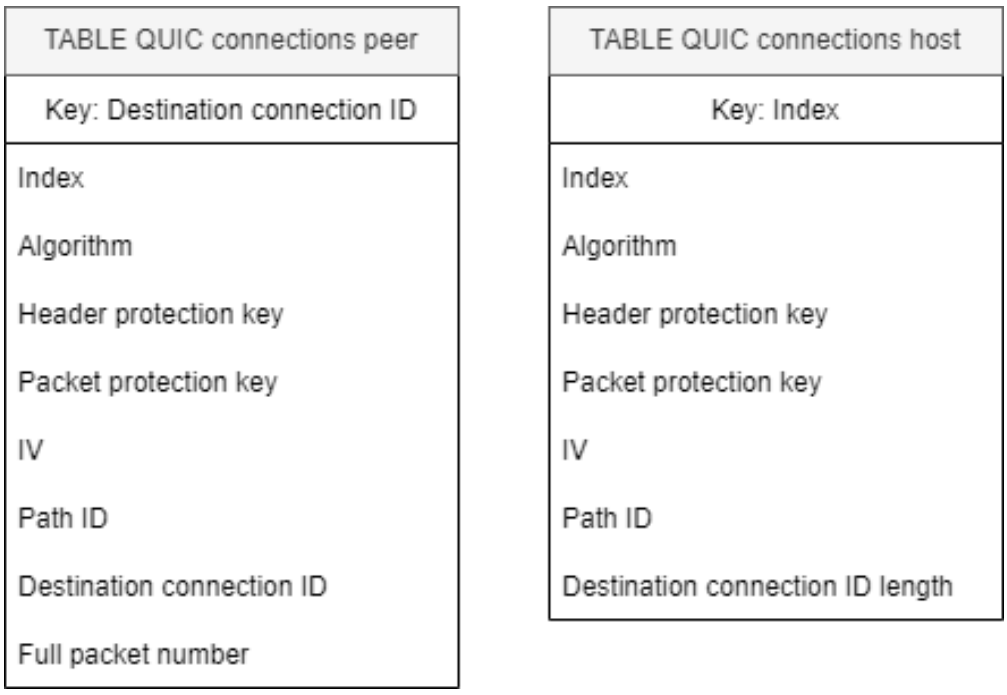

Figure 3.3.2: Tables used by the application for storing information

- **Packet protection key** - The key used to encrypt or decrypt the payload with the AEAD algorithm
- **IV** - Used for calculating the nonce for the AEAD algorithm by combining the IV and the reconstructed path-and-packet-number with exclusive or.
- **Path ID** - An ID used for a path. Needed for calculating the nonce for the AEAD algorithm.
- **Destination connection ID** - The connection ID for the host.
- **Destination Connection ID length** - The length of the connection ID for the peer.
- **Full packet number** - The full packed number decoded. Used for creating the nonce for the AEAD algorithm.

The header protection and packet protection keys are in both cases sent by the hosts as the maximum key lengths, i.e. 256 bits. If the keys are 128 bits the rest of the field will be padded with zeros. When the packets are received by the application, the algorithms used for the connection is checked. If the algorithms use 128 bits keys, the left-padded zeros are ignored.

The *destination connection ID* field in the QUIC header is of variable length, and can be 0 to 160 bits long. In order to process the field at the right byte, the connection ID length must be known. Each host using QUIC, decides which ID should be used for itself. The host will decide which value the peer will set in the *destination connection ID* field of the packets it sends and the other way around. The ID can then be of assumed to be of a specific static length, i.e. always 64 bits long when packets are received by the

offloading application from the peer. When packets will be transmitted from the host, the peer has chosen the *destination connection ID* which the host will use. In order to determine the length the information must be received by the QUIC stack as signaling information in the signaling host packet.

3.4 Offload application design using P4

A general P4 reference design concept is generated, but not implemented in this project. Some functionality, such as the encryption and decryption, are performed by externs which are depending on factors, such as the P4 architecture and compiler, and are therefore not described. To implement this solution, the T4P4S and MACSAD compilers or the P4-DPDK-target and bmv2 software switches can be used since they are open-source and allow developers to add functionality. The P4 code can then utilize functionality or environments for crypto operations. The major challenge when implementing the P4 pipeline is extending the mentioned options with external functions using the DPDK library. To implement this, the options need to be investigated to identify how these external functions can be added. In this section, a P4 parser is first presented, as this is the first step in a P4 program. The match-action data plane pipeline is then described, followed by the design of the control plane.

3.4.1 P4 parser design

The design of the parser can be seen in figure 3.4.1. The parser starts by receiving an incoming packet. The Ethernet header is first extracted, and the field *Ethertype* in the header is used to check if the following protocol is IPv4 (0x0800), local experimental 1 (0x88B5), or local experimental 2 (0x88B6). If *Ethertype* is set to the IPv4 type, a user application data packet from the peer is received, or if it is set to local experimental 2, the packet is a user application data packet from the host. If *Ethertype* is local experimental 1, a signaling packet is received.

If the packet contains an IPv4 header, the IPv4 header is extracted. A check is made to examine if the *protocol ID* field is set to UDP, i.e. 17. In that case, the UDP header is extracted. The QUIC header (except the packet number) is extracted. The *packet number length* field in the QUIC header is examined since it can have the values 0, 1, 2, or 3. It is used to determine how many bytes will be extracted as the packet number field. The parser moves to an accept state and the match-action pipeline starts.

If the packet on the other hand is a user application data packet from the host a data packet information header is extracted. The QUIC header is extracted followed by the packet number depending on the *packet number length* field in the QUIC header. The parser continues to an accept state.

If the packet is a signaling packet the P4 lookahead function will be used. This function is used to examine the following bits without extracting them. The type of the packet is determined. If the byte has the value 0 the QUIC connection peer header is extracted,

and if it has the value 1 the QUIC connection host is extracted. The parser moves to an accept state.

The payload is often not parsed or used in a P4 program, but reattached to the headers at the end of the process, before being transmitted. In this application the payload is needed for the encryption. It will not be extracted by the parser but added later by an external function.

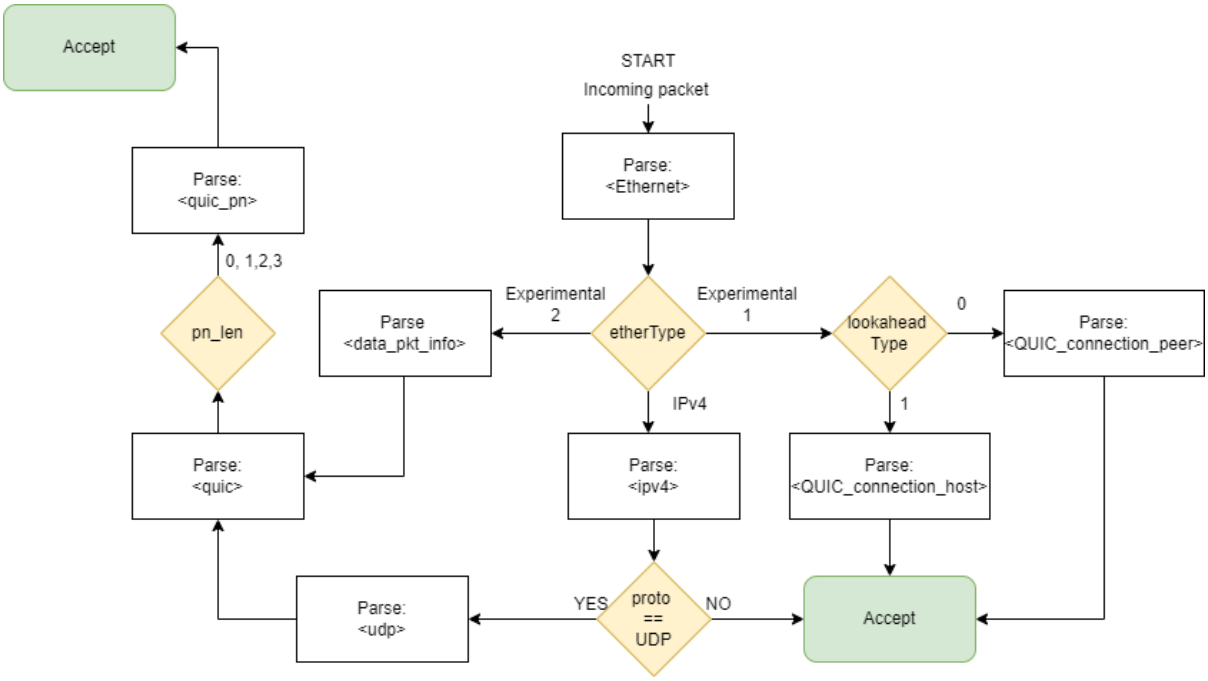

Figure 3.4.1: The parser design

3.4.2 P4 match-action pipeline design

Since P4 does not support encryption, decryption, or message tag authentication, these features have to be implemented using P4 externs. A P4 program can be compiled using a compiler which supports such functionality. The data plane pipeline will be similar to the offloading design using DPDK seen in figure 3.3.1.

The packet are received in the ingress control block after being parsed. In this control block, a check is made to determine which direction the packet was sent from, i.e. which port received the packet. The Ethernet header's *Ethertype* field is also examined to determine how to further process the packet.

When *Ethertype* is set to local experimental ethertype 1 the extracted header needs to be determined. P4 has functionality to test validity bits, which is used to check whether the headers have been extracted or not by the parser. In this step, a check is made to see if the header QUIC connections peer or QUIC connections host is valid. The information sent in these packets (described in section 3.2.1) are stored in table enc_table for the QUIC connection host header and dec_table for the QUIC connection

peer header, described in 3.4.3. After storing the information, the packets are dropped and the processing of the next packet starts.

If *Ethertype* is set to local experimental ethertype 2 the packets will be reconstructed to an IP packet, encrypted, and sent to the peer. A IPv4 and UDP header is populated and set valid. The IPv4 *protocol ID* is set to UDP. The enc_table is applied to the packet using the index as a key. If a match occurs the specified encryption action is triggered, and if not the packet is dropped. Since P4 does not support encryption or decryption, these functionalities are implemented using externs. After the packet has been encrypted the l3_fwd_table (later described in 3.4.3) is applied. The key to this table is the IPv4 destination address and the action called when a hit occurs sets the MAC destination address and the egress port. If no match is found, the packet is dropped. The data_pkt_info header is set invalid. The Ethernet *Ethertype* header field is set to IPv4. The *Total Length* field in the IPV4 header and the *Length* field in the UDP header is updated with the correct lengths. The UDP and IPv4 *Checksum* fields are calculated and set. The packet is sent towards the peer.

If *Ethertype* is IPv4, the packet is sent from the peer and the QUIC header and payload will be decrypted, decapsulated, and sent to the host. The dec_table is applied with the destination connection ID as key. The extern action used for decryption is called when a hit occurs and the packet is dropped if not. After decryption, the l3_fwd_table is applied. The new MAC destination address and the egress port is set. The IPv4 and UDP headers are set invalid. The Ethernet *Ethertype* is set to local experimental ethertype 2. The data_pkt_info header is set valid to send metadata back to the host. The metadata contains the index and full packet number. The packet is sent to the host.

No processing is done at the egress. The final user defined packet is created in the deparser. The following header order is applied: Ethernet, IPv4, UDP, data_pkt_info, QUIC, and QUIC_pn. The headers are only a part of the final packet if they are set valid.

3.4.3 P4 design control plane

Three tables are used to manage the packet processing: enc_table, dec_table and l3_fwd_table. The enc_table is responsible for providing the necessary parameters which will be used by the encryption action. The table can be seen in figure 3.4.2 In this table, the index is used as key values for matching. Each active index is mapped to a specific action, which can be AES_GCM128_AES_ECB, AES_GCM256_AES_ECB, AES_CCM128_AES_ECB, CHACHA20_POLY1305, or drop. Each of these actions except drop receive a number of parameters depending on which encryption algorithms will be used. The encryption action receive keys, an IV, a path ID and the destination connection length. If no match occurs the drop action will be called, which will discard the packet.

If the packets are sent from the peer towards the host, the QUIC header and payload

ENC TABLE	
Match key: Index	
Action	Parameters
AES_GCM128_AES_ECB	Header protection key Packet protection key IV Path ID Destination Connection ID length
AES_GCM256_AES_ECB	Header protection key Packet protection key IV Path ID Destination Connection ID length
AES_CCM128_AES_ECB	Header protection key Packet protection key IV Path ID Destination Connection ID length
CHACHA20_POLY1305	Header protection key Packet protection key IV Path ID Destination Connection ID length
DROP	-

DEC TABLE	
Match key: Destination connection ID	
Action	Parameters
AES_GCM128_AES_ECB	Index Header protection key Packet protection key IV Path ID Full packet number
AES_GCM256_AES_ECB	Index Header protection key Packet protection key IV Path ID Full packet number
AES_CCM128_AES_ECB	Index Header protection key Packet protection key IV Path ID Full packet number
CHACHA20_POLY1305	Index Header protection key Packet protection key IV Path ID Full packet number
DROP	-

Figure 3.4.2: The tables enc_table and dec_table used by the P4 application

will be decrypted. Table dec_table is used for storing the necessary information used for decryption and calling the correct decryption action. This table is similar to enc_table, but uses the destination connection ID as a key for matching. The parameters that the actions can receive are crypto keys, an IV, the path ID and the full packet number. The drop action is called when the destination connection ID gets no match.

The final table used in this design is l3_fwd_table, which is used to route which port the packet should be sent to, i.e. the egress port, and which MAC destination address should be used for the next hop. The matching key for this table is the IPv4 destination address and the actions that can be called are forward and drop. The table can be seen in figure 3.4.3.

L3-FWD TABLE	
Match key: IPv4 destination address	
Action	Parameters
forward	Ethernet destination address Egress port
DROP	-

Figure 3.4.3: The forwarding table used by the P4 application

Chapter 4

Implementation

In this chapter, the implementation of the QUIC offloading design using DPDK from chapter 3 is described. The program is first initialized by the master logical core, where ports, queues, and crypto devices are created. The data processing pipeline is then presented followed by the control plane.

4.1 Initialization

Two different logical cores are used to run the application. On main logical core is used to set up necessary processes for the application. These processes are port, queue, crypto device, and core initialization. In every program utilizing the DPDK framework, the function *rte_eal_init()* is called. This function initialize the EAL with the given parameters. In this program the input parameters to this function are ”-l 0-1 -n 4 –vdev ”crypto_openssl” which means EAL should be initialized with logical cores 0 and 1, a quad channel memory architecture should be used, and a virtual crypto device using openssl should be created. The main logical core then continues creating a mbuf pool and crypto op pool. The mbuf pool is created by using the function *rte_pktmbuf_pool_create()*, which creates and initialize a packet message buffer memory pool used to store the packets that the offloading application will receive. The *rte_crypto_op_pool_create()* function creates a crypto operation pool which is used to store a buffer of crypto operations which the crypto device will receive. Each crypto operation contains information about the crypto operation the packet will be processed by.

```
 /* create the mbuf pool */
pktmbuf_pool = rte_pktmbuf_pool_create("mbuf_pool", NB_MBUF, 512,
                        RTE_ALIGN(sizeof(struct rte_crypto_op),
                        RTE_CACHE_LINE_SIZE),RTE_MBUF_DEFAULT_BUF_SIZE,
                        rte_socket_id());
if (pktmbuf_pool == NULL)
    rte_exit(EXIT_FAILURE, "Cannot create mbuf pool\n");

/* create crypto op pool */
```

```
crypto_op_pool = rte_crypto_op_pool_create("crypto_op_pool",
                        RTE_CRYPTO_OP_TYPE_SYMMETRIC, NB_MBUF, 128,
                        MAXIMUM_IV_LENGTH, rte_socket_id());
if (crypto_op_pool == NULL)
    rte_exit(EXIT_FAILURE, "Cannot create crypto op pool\n");
```

Listing 4.1: Creating the mbuf pool and crypto op pool

After these memory pools have been created and initialized, the Ethernet ports and queues are created and configured. The number of available Ethernet ports are given by the function *rte_eth_dev_count_avail()*, but since only two ports are used by the offloading application the rest of the ports on the DPU will be ignored. The function *rte_eth_dev_info_get()* are called for each port, which returns the Ethernet device's contextual information and fills in parts of the structure *rte_eth_dev_info* to access the controlling driver of the Ethernet device. The ports are then configured with the function *rte_eth_dev_configure()*. This function receives the number of receive queues and the number of transmit queues as parameters, and in this program the number of queues are one of each type for each Ethernet device. The rx queues are setup with the function *rte_eth_rx_queue_setup()* and tx queues with *rte_eth_tx_queue_setup()* for each port. The Ethernet device is finally started with the function *rte_eth_dev_start()* which is then configured with the specified offloading features and the transmit and the receive units of the device are started. The configurations of the ports can be seen in listing 4.2. A check is made to see if fast release of mbufs and IPv4 and UDP checksum calculations are supported. The last step of initializing the ports is to assign the logical cores the responsibility of polling one port each.

```
if (dev_info.tx_offload_capa & RTE_ETH_TX_OFFLOAD_MBUF_FAST_FREE)
    local_port_conf.txmode.offloads |=RTE_ETH_TX_OFFLOAD_MBUF_FAST_FREE;
rte_eth_dev_configure(portid, 1, 1, &local_port_conf);

if(!(dev_info.tx_offload_capa & DEV_TX_OFFLOAD_UDP_CKSUM) ||
 !(dev_info.tx_offload_capa & DEV_TX_OFFLOAD_IPV4_CKSUM))
    rte_panic("offload not supported");
```

Listing 4.2: Configuring offload capabilities for the Ethernet ports

The next step in the main core's responsibility is to set up and configure the crypto device. Two crypto devices will be created. Each crypto device is linked with one of the ports and logical cores pair. The following process is made for both crypto devices. The function *rte_cryptodev_info_get* is called and returns the contextual information of the crypto device in the structure *rte_cryptodev_info*. Two session memory pools are created to hold the header and private data for each session. The functions *rte_mempool_create()* and *rte_cryptodev_sym_session_pool_create()* are used for the creation of memory pools. The crypto device initialization if finalized by calling the function *rte_cryptodev_configure()* to configure the device, *rte_cryptodev_queue_pair_setup()* to map the crypto device to a receive queue pair and *rte_cryptodev_start()* to start the device.

The final step in the main core is to create a *rte_hash_table* to store signaling information used when processing packets from the peer of the connection. This table is created by setting *rte_hash_parameters* and use as this as input to the *rte_hash_create()* functions. The key to the table will be the destination connection ID and the length is set to 32 bits.

Lastly, the function *rte_eal_mp_remote_launch()* is called, which is used to launch one continuously running main function on both logical cores.

4.2 Data plane

Each of the two logical cores 0 and 1 is responsible for polling packets from their assigned ports. The data packets are placed in rx queues and processed. Each cores enqueues the packets at the crypto device. Each core is also responsible for dequeuing the packets from the crypto device, do further processing and finally transmit the packets in the right direction by placing them in a tx queue at the Ethernet device.

Each core is running a main processing function. The function consists of two parts, one where both crypto devices are configured further, and the other of a continually running loop pulling packets from the ports. In the first part, some fields of the structure *rte_crypto_sym_xform* needs to be filled with the correction values for each crypto device, since one port will be encrypting and the other decrypting packets. Certain fields of the structure *rte_crypto_op* are also filled before entering the loop. This decision is made because the processing between receiving and transmitting a packet should be minimized. Fields that are not specific for each packet should be set outside of the continually running loop.

The final step of the configuration of the crypto devices in the main processing function is to call the function *rte_cryptodev_sym_session_create()* which creates a symmetric crypto session header.

The main functionality of the main processing function is to poll packets from the rx queues, process them, place them in tx queues, and transmit the packets. The rx queues are read by using the function *rte_eth_rx_burst()* which retrieves a burst of incoming packets of the Ethernet device and places them in the structure *rte_mbuf* which is a buffer containing the packets. If the function returns a non-empty value the function *rte_crypto_op_bulk_alloc()* is called which bulk allocates and fills crypto operations from the crypto_op_pool memory pool.

Each packet is now individually processed before being enqueued at the crypto device. If the packet contains user application data and is sent by the peer the *rte_hash_lookup_data()* is used to look up the connection information. If the hosts sends user application data, the index in the packet is used to find an entry in the structure where the information is stored. For each encrypted packet, a crypto operation is attached with all necessary information on how the packet should be processed by the crypto device. For this step, the structure *rte_crypto_op* is used

which points to a *rte_crypto_sym_op* structure since a symmetric cryptographic processing will be performed. Parts of this structure can be seen in listing 4.3. The offset is where the AEAD algorithm should start processing, i.e. the number of bytes with the start of the source buffer to the start of the payload data that will be encrypted. The length is the number of bytes of the plaintext that will be encrypted. The length start from the offset position. The digest data is the authentication tag which will be produced by the AEAD algorithm. When encrypting, the pointer is set to the start bytes of the allocated space where the tag will be placed after processing. If the packet will be decrypted the pointer is set to the start byte of the existing tag, which is 128 bits counting from the end when using AEAD. The physical address is set to the address the data pointer is set to. The aad of a QUIC encryption is the QUIC header. The data and physical address is set to point at this byte in the mbuf.

```
    struct {
        struct {
            uint32_t offset;
            uint32_t length;
        } data;
        struct {
            uint8_t *data;
            rte_iova_t phys_addr;
        } digest;
        struct {
            uint8_t *data;
            rte_iova_t phys_addr;
        } aad;
    } aead;
```

Listing 4.3: Parts of the rte_crypto_sym_op structure

The rest of the *rte_crypto_sym_xform* structure is now to be filled in before enqueuing the packet at the crypto device. If the packet is sent from the peer, the offloading application needs to decode the truncated packet number received in the QUIC header. This process is done by calling the function decode_pkt_nr() shown in listing 4.4.

```
uint64_t decode_pkt_nr(uint64_t largest_pn, uint64_t truncated_pn, uint64_t
    pn_nbits){
        uint64_t expected_pn, pn_win, pn_hwin, pn_mask, candidate_pn;

        expected_pn = largest_pn + 1;
        pn_win = 1<< pn_nbits;
        pn_hwin = pn_win / 2;
        pn_mask = pn_win -1 ;

        candidate_pn = (expected_pn & ~pn_mask) | truncated_pn;
        if((candidate_pn <= expected_pn - pn_hwin) && (candidate_pn < (1ull
    << 62) - pn_hwin))
                return candidate_pn + pn_win;
        if((candidate_pn > expected_pn + pn_hwin) && (candidate_pn >=
   pn_win))
                return candidate_pn - pn_win;
```

```
        return candidate_pn;
}
```

Listing 4.4: Decoding the truncated packet number into the full 62 bit packet number

The nonce is created by using the function set_nonce(), shown in listing 4.5. The key is set with information from the tables. The packets are sent to the crypto device by calling the *rte_cryptodev_enqueue_burst()* function.

```
void set_nonce(uint32_t path_id, uint8_t *iv, uint64_t full_pn, struct
    rte_crypto_op *op, uint8_t algo){
    int tal_cid = 24, tal_pn = 56;
    uint8_t *iv_ptr = rte_crypto_op_ctod_offset(op, uint8_t *,IV_OFFSET);
    uint8_t iv_size = 12;
    uint8_t *nonce = (uint8_t*)malloc(iv_size);
    uint8_t *byte_pair = (uint8_t*)malloc(6);

    uint32_t cid_seq = htonl(path_id);

/* store bytes from mbuf into a variable and creating the nonce */
    for(int i = 0; i< iv_size; i++){
        if(i<4){
            *(nonce+i) = cid_seq >> tal_cid;
            tal_cid-= 8;
        }
        else{
            *(nonce+i) = full_pn >> tal_pn;
            tal_pn -= 8;
        }
        *(nonce+i) = *(iv+i) ^ *(nonce+i);
    }
    if(algo == AES_CCM128_AES_ECB)
        rte_memcpy(iv_ptr +1, nonce, iv_size);
    else
        rte_memcpy(iv_ptr, nonce,iv_size);
}
```

Listing 4.5: Creating the nonce

The packets are dequeued from the crypto device by using the function *rte_cryptodev_dequeue_burst()*. If the packets are originally sent from the peer the final step is to remove the IPv4 and UDP header by using *rte_pktmbuf_adj()*. The Ethernet header *ethertype* is set to 0x88B6 and hardware offloading is used for calculating the checksum of IPv4 and UDP, seen in listing 4.6. The packet is placed in the tx queue and sent with *rte_eth_tx_burst()* towards the host. If the packet was originally sent from the host, the QUIC header is encrypted by calling *header_encrypt()* and finally the index and full packet number is removed by using *rte_pktmbuf_adj()*. The IPv4 and UDP checksums are hardware offloaded. The packet is sent *rte_eth_tx_burst()* towards the peer.

```
m->ol_flags |= RTE_MBUF_F_TX_IPV4 | RTE_MBUF_F_TX_IP_CKSUM |
RTE_MBUF_F_TX_UDP_CKSUM;
```

```
3
4 m->l2_len = sizeof(struct rte_ether_hdr);
5 m->l3_len = sizeof(struct rte_ipv4_hdr);
6 ip_hdr->hdr_checksum = 0;
7 udp_hdr->dgram_cksum = rte_ipv4_phdr_cksum(ip_hdr, m->ol_flags);
```

Listing 4.6: Offloading the checksum calculations of IPv4 and UDP

4.3 Control plane

The control plane consists of two types of tables, and depending on which direction the packet traverse the offloading application a different table is used.

The packets sent from the host is processed by the information stored in a array consisting of the structure *QUIC_connections_host*, seen in listing 4.7. Since the QUIC packets sent from the host contains an index, the corresponding entry can easily be found. The signaling packets used to create an entry in this table contains the 8 bits long header field *Type* which is set to 1. The mbuf is read with the function *rte_pktmbuf_mtod_offset()* and the offset is the Ethernet header size plus 8 bits to leave out the *Type* field. The result is casted into the structure *QUIC_connections_host* at the array index retrieved from the index field of the packet.

```
1 struct QUIC_connections_host{
2         uint16_t index;
3         uint8_t algo;
4         uint8_t hp_key[MAX_KEY_LEN];
5         uint8_t quic_key[MAX_KEY_LEN];
6         uint8_t iv[IV_LEN];
7         uint32_t path_id;
8         uint8_t dcid_len;
9 } __rte_packed;
```

Listing 4.7: The table used for storing signaling packets for processing packets from the host

Packets containing signaling information used when processing packet from the peer are stored in the structure *QUIC_connections_peer* which can be seen in listing 4.7. The information read from the mbuf by using *rte_pktmbuf_mtod_offset()* where the offset is the Ethernet header plus 8 bits due to the header field *Type*. The result is casted into the structure and the structure is for easier handling. Before the packet is discarded, the structure is casted into a void type and stored in a *rte_hash_table*. The destination connection ID is used as a key to the table. To add a new entry the function *rte_hash_add_key_data()* is used.

```
1 struct QUIC_connections_peer{
2         uint16_t index;
3         uint8_t algo;
4         uint8_t hp_key[MAX_KEY_LEN];
5         uint8_t quic_key[MAX_KEY_LEN];
```

```
        uint8_t iv[IV_LEN];
        uint32_t path_id;
        uint8_t dcid[DCID_LEN];
        uint64_t full_pn;
} __rte_packed;
```

Listing 4.8: Table stored in the rte hash table. Stores signaling packets for processing packets from the peer

4.4 Generating test cases

In order to evaluate the implementation, test cases are created. A real QUIC stack and user application will not be used to evaluate the offloading application, as this is part of future work to integrate these with the offloading application. The packets will instead be crafted. These test cases consists of user application data packets, i.e. packets that will be sent to the application, processed and forwarded, and packets containing signaling information that will be stored in the tables. The signaling packets are sent in a burst before starting the performance evaluation with data packets. The signaling and data packets can be sent mixed, but before a data packet is received a table entry with its index or destination connection ID needs to be created or the packet will be dropped.

4.4.1 Generating table entries

To generate table entries, signaling packets need to be created. To achieve this, the packet manipulation library Scapy in Python is used. A Python program is created, which is used to send signaling information from the x86 host to the application. This program creates 10 different connections, with information of the connection for packets sent from both directions. If the peers x and y is communicating, the application receives one packet with signaling information linked to packets sent from peer x and one packet linked to packets sent from peer y. The programs will receive signaling two packets per communication session. For the evaluation of 10 communication the application receives 20 signaling packets containing information to populate the tables.

The signaling header from the host can be seen in listing 4.9. The header fields is implemented using Scapys *BitField* method. The input to this method is the header field name, a default value, and the number of bits the field will be encoded with. For example has the header field *Type* the default value 1 and is encoded using 8 bits. The header field *Index* needs to be changed for each connection.

```
class signaling_host(Packet):
        name = "signaling_host"
        fields_desc = [
                BitField("Type",1,8),
                BitField("Index",0,16),
```

```
                BitField("Algo", 0,8),
                BitField("Hp_key",0x0,256),
                BitField("Quic_key",0x0,256),
                BitField("IV",0x0,96),
                BitField("Path_id", 0, 32),
                BitField("DCID_len",0,8)
        ]
```

Listing 4.9: Signaling packets for processing packets from the host

The signaling header from the peer is similar to the signaling header from the host and can be seen in listing 4.10. The header fields *DCID* and *Index* need to be changed for each connection.

```
class signaling_peer(Packet):
        name = "signaling_peer"
        fields_desc = [
                BitField("Type",0,8),
                BitField("Index",0,16),
                BitField("Algo", 0,8),
                BitField("Hp_key",0x0,256),
                BitField("Quic_key",0x0,256),
                BitField("IV",0x0,96),
                BitField("Path_id", 0, 32),
                BitField("DCID",0x0,64),
                BitField("Full_pn", 0x0, 64)
        ]
```

Listing 4.10: Signaling packets for processing packets from the peer

The packets are created by attaching an Ethernet header in front of the signaling header. The Ethernet header field *Ethertype* is set to 0x88B5 which is a local experimental ethertype.

4.4.2 Generating packets

A set of user application data packets are generated using Scapy. These packets are sent to a port and by using the packet capturing program *tcpdump* which is sniffing on the port, the packets are stored in a packet capture pcap file. The pcap file is the source to stream packets from, when using the packet generator Pktgen-DPDK. The packet generator Pktgen-DPDK is a software based on DPDK and is used to generate traffic. It is used both as a receiver and a transmitter on the x86 processor.

A user application data packet received by the offloading application from the host consists of an Ethernet header, the data pkt header, the QUIC header and the unencrypted payload. The *Ethertype* is set to the local experimental ethertype 0x88B6. The data pkt header can be seen in listing 4.11, and consists of an index and the full packet number. The index must match to an existing index in control plane table in the application, or the packet will be dropped. The payload is different depending on

which packet size will be evaluated. Packets are created to contain 128, 256, 512, 1024, and 1400 Bytes of payload.

```
class data_pkt(Packet):
        name = "data_pkt_host"
        fields_desc = [
                BitField("Index",0,16),
                BitField("Full_pn",0x0 ,64)
        ]
class quic(Packet):
        name = "quic"
        fields_desc = [
                BitField("Header_form",0,1),
                BitField("Fixed_bit",1,1),
                BitField("Spin_bit",0,1),
                BitField("Reserved_bits",0,2),
                BitField("Key_phase",0,1),
                BitField("Pn_len",0,2),
                BitField("DCID",0x0,64),
                BitField("Pn", 0x0,16)
        ]
```

Listing 4.11: Data packets received by the offloading application

Chapter 5

Evaluation and Result

In this chapter, the following questions will be answered:

1. Which throughput can be achieved when different key sizes are used in the offloading application?
2. What latency can be expected when using the offloading application?
3. What throughput can be achieved when encrypting AES_GCM_128 in DPDK using a hardware accelerated crypto device compared to a software crypto device?
4. What throughput can be achieved of the crypo algorithms in AEAD used by QUIC when hardware not supported by DPDKs crypto devices are used?

To answer question 1, the testbed presented in section 5.1 is used. The offloading application is placed on the DPU and packets are sent by a packet generator running at the x86 host. The rx throughput at the receiver will be measured in gigabits per second and packets per second.

To answer question 2, the same testbed described in section 5.1 will be used. Modifications of the offloading application have to be made due to the latency measurement packet format from the traffic generator. Only an estimated latency can be measured since these packets will not be processed in the same way as QUIC packets.

Question 3 will be answered by using the QAT hardware accelerated crypto device and the AESNI-GCM software crypto device supported in DPDK. Measurements from Intel [19] using the *dpdk-test-crypto-perf* tool will be used to provide an answer to this question.

Question 4 will be answered by evaluating all supported crypto algorithms using different hardware devices with the OpenSSL speed engine.

5.1 Testbed

To test the QUIC crypto offload a testbed is used which consists of two servers. The setup can be seen in figure 5.1.1. Each server consists of a x86 processor and a BlueField 2 DPU. The x86 processor is a Intel(R) Xeon(R) CPU E5-2660 v4 running at the frequency 2.00GHz. Ubuntu 20.04.4 LTS is installed on both the CPU and DPU. A 25 Gbps link is attached between the DPU on Device Under Test (DUT) 1 port P0 and DUT 2 port P0. The DPUs are connected to the board on the servers via PCIe. The CPU is running the packet generator software application PktgenDPDK version 22.2.0. The DPU which is running the offloading application is using DPDK version 21.11.0. The BlueField 2 DPU supports hardware crypto acceleration, but the acceleration engine cannot be reached via DPDK due to driver limitations. As seen in figure 5.1.1, the public key crypto accelerator is not used and the crypto is performed in software using the Arm CPU instead.

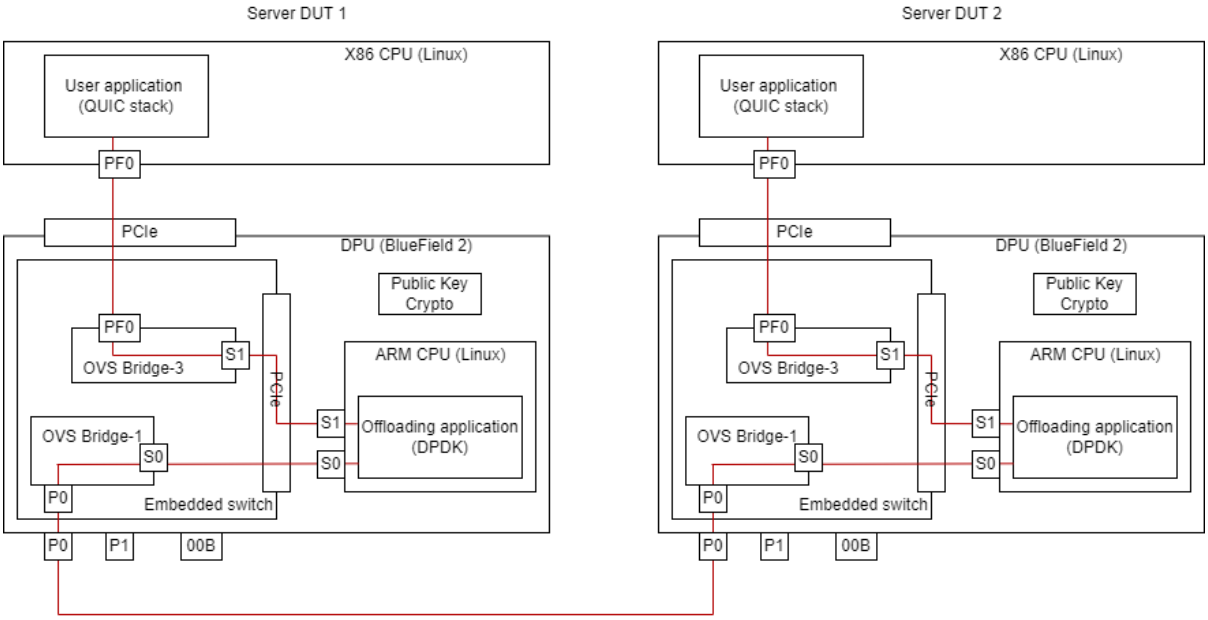

Figure 5.1.1: The testbed used for evaluating the offloading application

5.2 Offloading application evaluation

In this section the performance of the QUIC offloading application is evaluated using the BlueField 2 DPU without a crypto engine. All crypto processing will be done in software on the Arm cores of the DPU. The testbed described in section 5.1 is used. The offloading applications running on each DPU uses the OpenSSL PMD crypto device for the AEAD crypto operations.

In a first evaluation of the offloading application, the throughput is measured in gigabits per second and packets per second. The results can be seen in figures 5.2.1 and 5.2.2. The packets are sent from the pktgen-DPDK application on the x86 on DUT 1, processed and encrypted on the offloading application on the DPU. It is then sent

to the offloading application on the DPU on DUT 2 and processed and decrypted and sent to the pktgen-DPDK application on the x86. The path can be seen in figure 5.1.1 as the red line. The rx throughput is measured on the pktgen-DPDK application on DUT 2.

The results showed that the key size of the Advanced Encryption Standard - Galois Counter Mode (AES-GCM) algorithm had little impact on the performance. The maximum difference in throughput observed when using different key sizes was 62 Mbps using 1400 Bytes packets. The minimum difference was 9 Mbps using 256 Bytes packets. When only forwarding the packets on the DPUs without any packet processing or crypto operations, the throughput was significantly improved. These packets are expected to gain higher throughput than the QUIC packets, since the QUIC packets are encrypted in software on the Arm cores and en/decapsulated. The unprocessed packets are forwarded by DPDK using the PMD, which is used to achieve kernel-bypassing. The speed of the link is 25 Gbps, which becomes the limiting factor for the packets. The increase in throughput scaling with the packet size was expected since smaller packet sizes implies more packet processing, such as packet header processing and setting up buffers, will be performed. A future evaluation would consist of measuring the throughput of the packet processing and en/decryption on the host in the user application. Since this is how QUIC packets normally are processed.

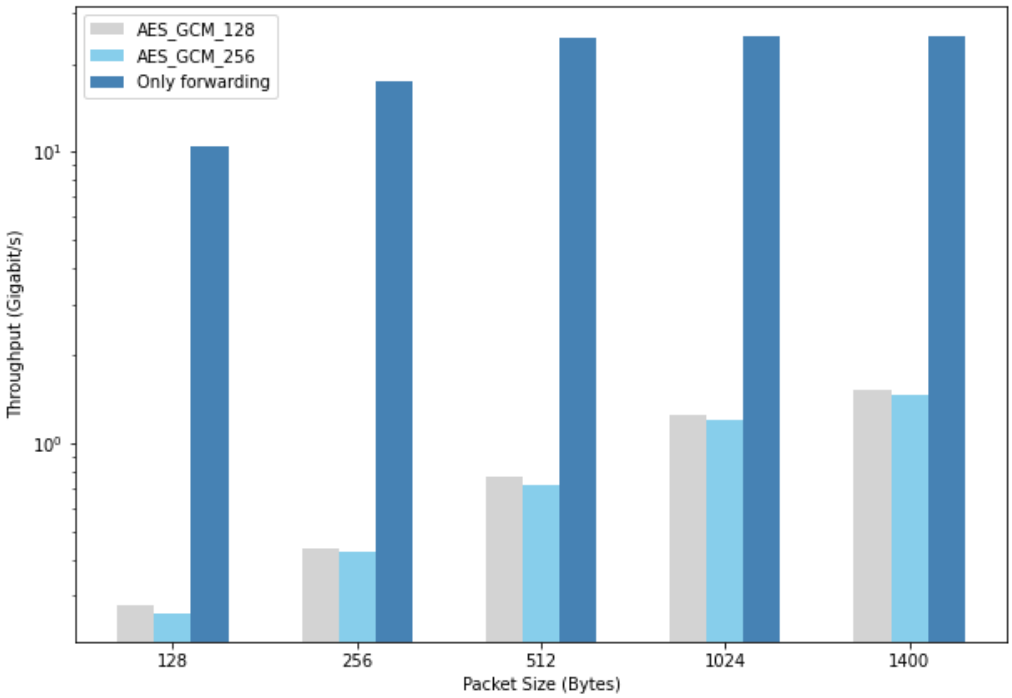

Figure 5.2.1: Throughput of different packet sizes using AES_GCM_128, AES_GCM_256 and only forwarding

In the second evaluation of the offloading application, the throughput is measured

using Kpps. The results are presented in figure 5.2.2. These results were excepted as well, since larger flows implies that the offloading applications and packet forwarding functionality during the path will require more time processing each packet. This increases the number of small packets that are able to be sent through the path compared to larger packets. This measurement also indicates a small negative impact of the performance when using 128 bits keys compared to 256 bits. The minimum packet throughput difference observed between the key sizes is 4 Kpps using 1024 Bytes packets. The maximum difference is 12 Kpps at 128 Bytes packets.

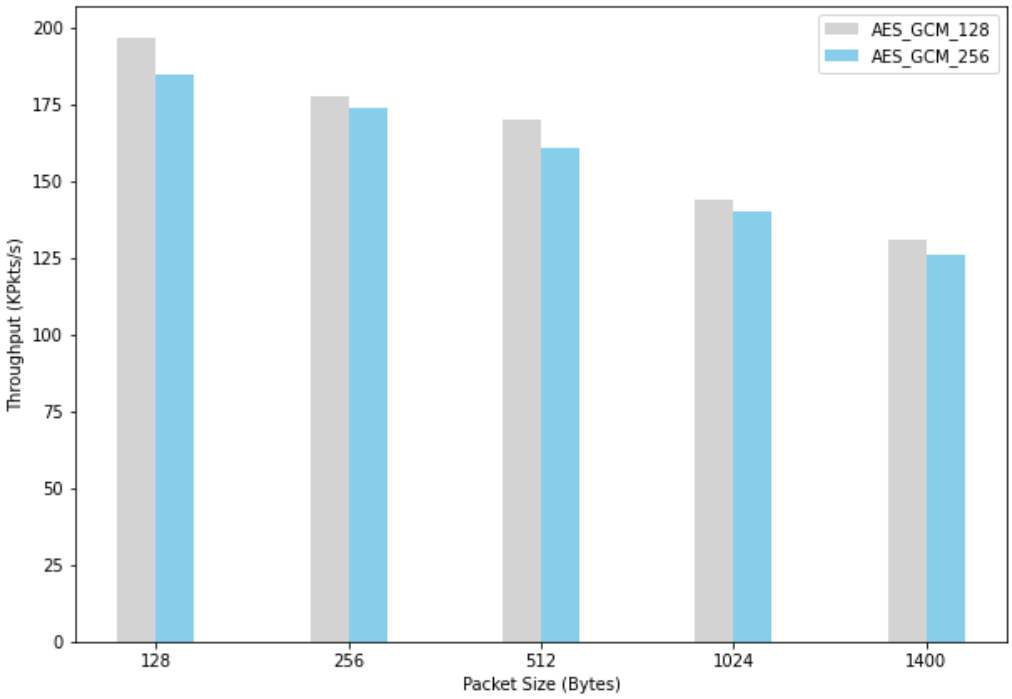

Figure 5.2.2: Packet rate of different packet sizes using AES_GCM_128 and AES_GCM_256

The estimated latency of using the offloading applications were measured. Two challenges were faced during the latency evaluation. One was that the pktgen-DPDK measures latency by sending packets containing an Ethernet, IPv4, UDP header with timestamp data attached at the end. Therefore, the packet would not be exposed to the same processing as the packets presented in section 3.2.1 that the offloading application expects to receive. In order to evaluate the latency, the offloading application was adapted to process these packets. Since the same functionality would not work using these latency packets, the processing pipeline consists of only encryption or decryption using the crypto device. The crypto parameters are set to static values, similar to the ones QUIC uses. The path in section 5.1 is used. Instead of only sending the packets from DUT 1 to DUT 2 the packets are looped back on the x86 of DUT 2 using the same path back to the x86 on DUT 1. The estimated latency of 1-RTT

can then be measured on the x86 on DUT 1. The other challenge that was faced during the latency evaluation was that the results were very inconsistent. The latency was significantly different using the same parameters during different evaluation periods. The results will not be presented, since a stable result could not be achieved.

In the next evaluation of latency, the jitter was measured. The results are seen in figure 5.2.3. By examining the jitter, the inconsistency of the latency measurements are presented. The results are shown in figure 5.2.3. As seen in the figure, the jitter percent increases to a high level already when sending at 2 percent of line rate. The packet loss starts at this rate. The bottleneck of the path is at the DPU on DUT 1. The packet loss past this device is low. This is most likely due to queues, i.e the rx, crypto or tx queue being full, resulting in packets being dropped.

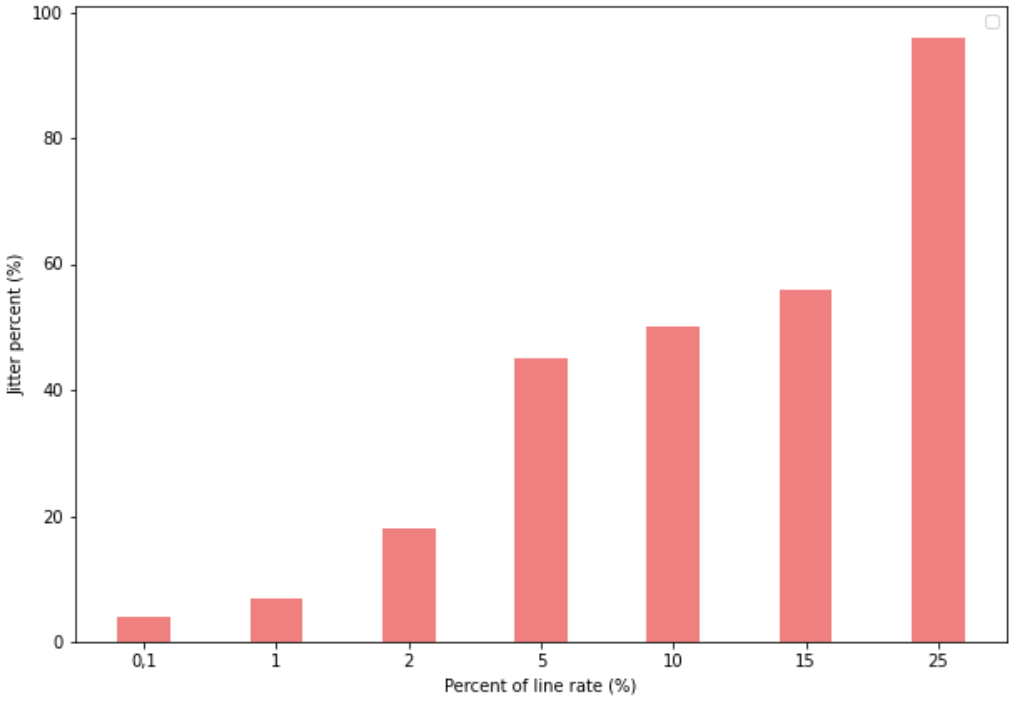

Figure 5.2.3: The jitter percent when sending at different rates.

5.3 DPDK supported crypto device benchmarks

DPDK offers the tool *dpdk-test-crypto-perf*, which is used to measure the performance of the PMD available on the hardware. In this section, measurements Intel QAT crypto PMD, and AES-NI GCM crypto PMD are presented. The parameters to the tool is set to represent the AEAD crypto parameters used in QUIC. The key size is set to 128 bits, the IV is 96 bits, the authentication tag is 128 bits. The associated data is set to 128 bits in figure 5.3.1. The input parameters to *dpdk-test-crypto-perf* can be seen in Appendix

A. Since AES-NI GCM does not support Advanced Encryption Standard - CBC counter mode (AES-CCM) and CHACHA20_POLY1305, only AES_GCM_128 will be evaluated on these benchmarks.

In figure 5.3.1, measurements by Intel are presented. The data comes from the DPDK Intel Cryptodev Performance Report and can be found in [19]. The hardware and software used for these measurements are described in the report. Intel presents measurements using the *dpdk-test-crypto-perf* tool with QAT and AESNI-GCM crypto devices. It should be noted that the aim of the QAT measurements was to reach the maximum throughput and therefore the measurements use parameters such as multi core configurations that are not used in the other tests.

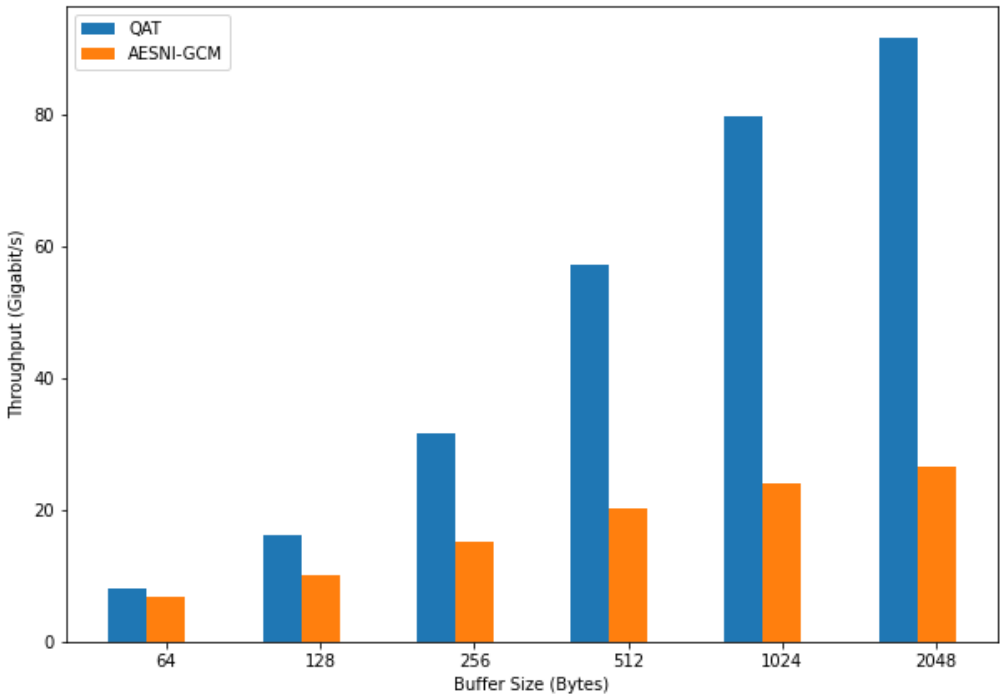

Figure 5.3.1: Measurements from the Intel Cryptodev Performance Report using the QAT and AESNI-GCM crypto devices [19]

The evaluation of crypto devices using *dpdk-test-crypto-perf* enlighten the impact of using a hardware accelerated crypto device compared to a crypto device running in software. The maximum difference using QAT and AESNI-GCM reached 65 Gbps. The evaluation also enlightens the impact of using a hardware crypto engine at larger buffer sizes. At small buffer sizes, such as 64 Bytes, the gain is minimal. If a hardware crypto device is used by user applications, such as video streaming that send elephant flows, the gain will have a greater impact.

5.4 OpenSSL benchmark

As a final evaluation, the openSSL library is used to evaluate the performance of processing all the supported AEAD crypto algorithms in QUIC using different devices. In this performance test, the *openssl-speed* utility is used to measure the throughput in gigabits per second of AES_GCM_128 and AES_GCM_256, AES_CCM_128 and CHACHA20_POLY1305 on different machines. The following hardware devices are used:

- BlueField 2 - Arm (Cortex-A72), 8 cores, 2.5GHz
- Intel Core i5 8250U - 4 cores, 1.6GHz
- Intel Xeon E5-2660 v4 - 14 cores, 2GHz
- Raspberry Pi 4 - ARM (Cortex-A72), 4 cores, 1.5 GHz

The Raspberry pie has no crypto support. The BlueField 2 DPU uses the *Armv8 Cryptographic Extension* in order to accelerate the crypto processing, which is suboptimal since the card has a hardware crypto accelerator but it cannot be reached using the *openssl-speed utility*. The two x86 CPUs have crypto acceleration supported through the AES-NI instruction set for Intel processors. The hardware devices have multiple other parameters than the crypto operations affecting the throughput. Therefore, the throughput should only be compared using different crypto algorithms for each device, and not the results of the devices compared with each other.

The results are presented in figure 5.4.1, 5.4.2, 5.4.3, and 5.4.4. The throughput on each hardware device is similar regardless if AES_GCM_128 or AES_GCM_265 is used. A decreased throughput can be observed when AES_CCM_128 is measured. In these evaluations, the Raspberry Pi card had a significant lower throughput than the other devices except when CHACHA20_POLY1305 was measured. The maximum difference in throughput using the Intel Core i5 8250U CPU is when AES_GCM_128 is compared to AES_CCM_128. By using AES_GCM_128 41 Gbps throughput is achieved compared to 10 Gbps using AES_CCM_128. A significant increased throughput can be seen on all devices except the Raspberry Pi 4 card when using AES-GCM compared to AES-CCM or CHACHA20_POLY1305.

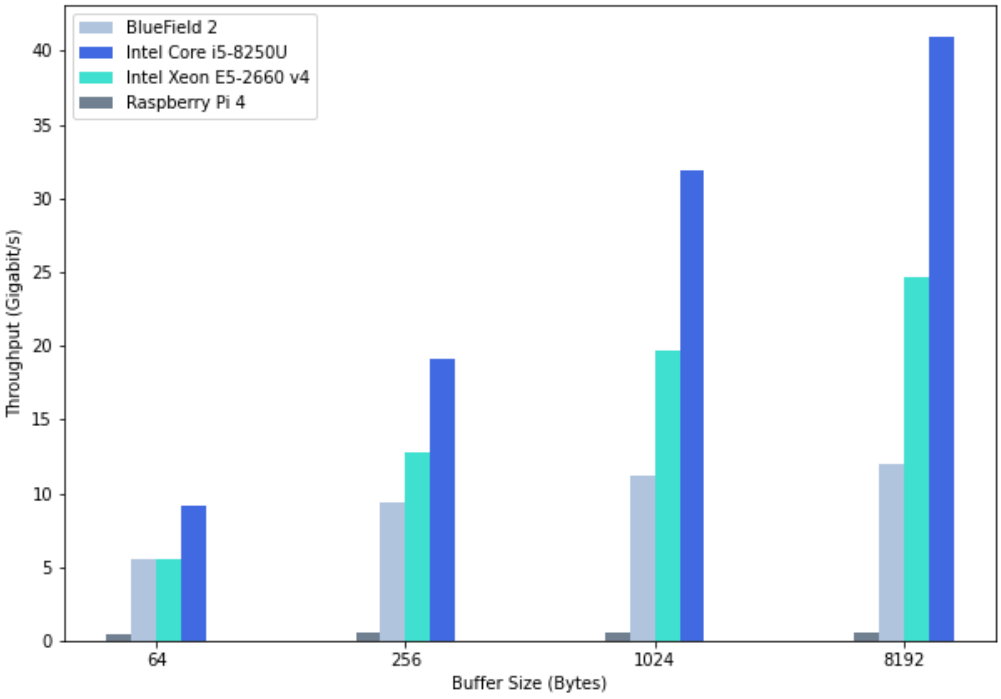

Figure 5.4.1: Throughput for different hardware using AES_GCM_128

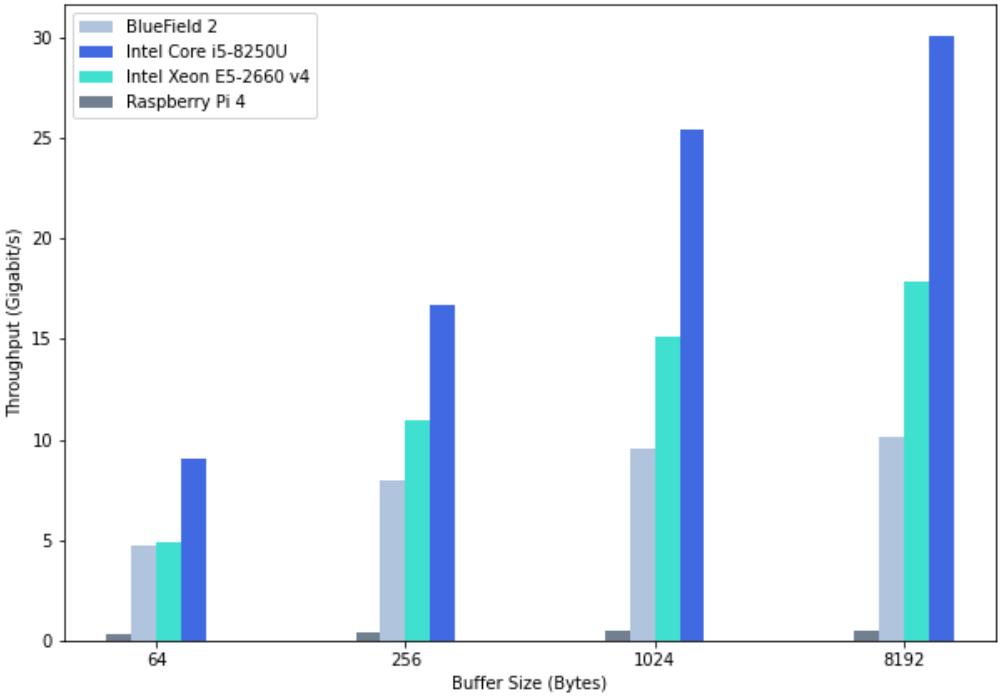

Figure 5.4.2: Throughput for different hardware using AES_GCM_256

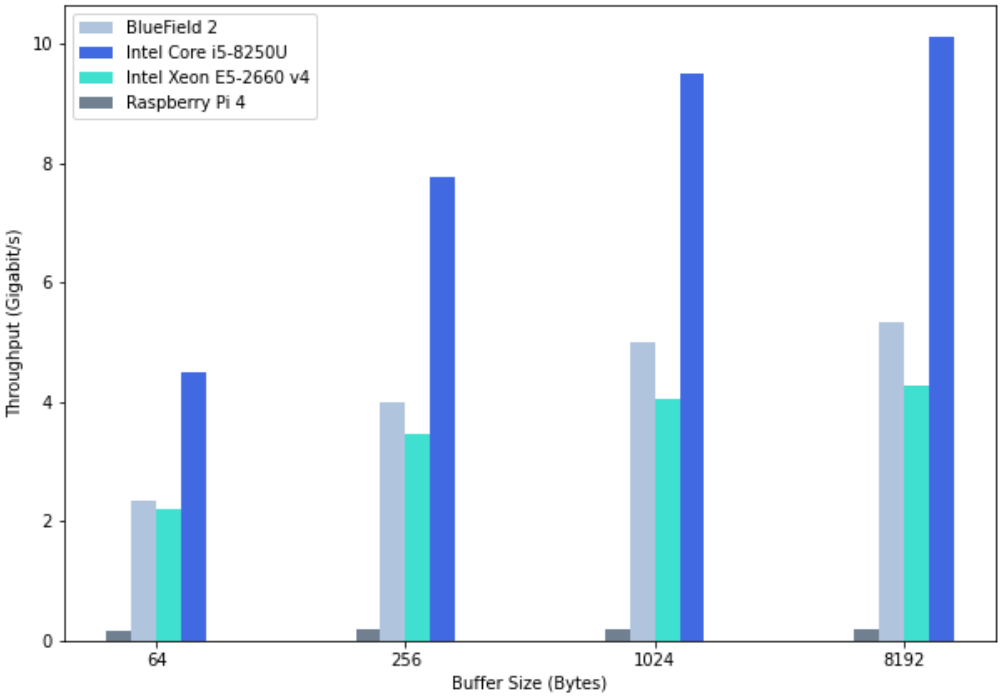

Figure 5.4.3: Throughput for different hardware using AES_CCM_128

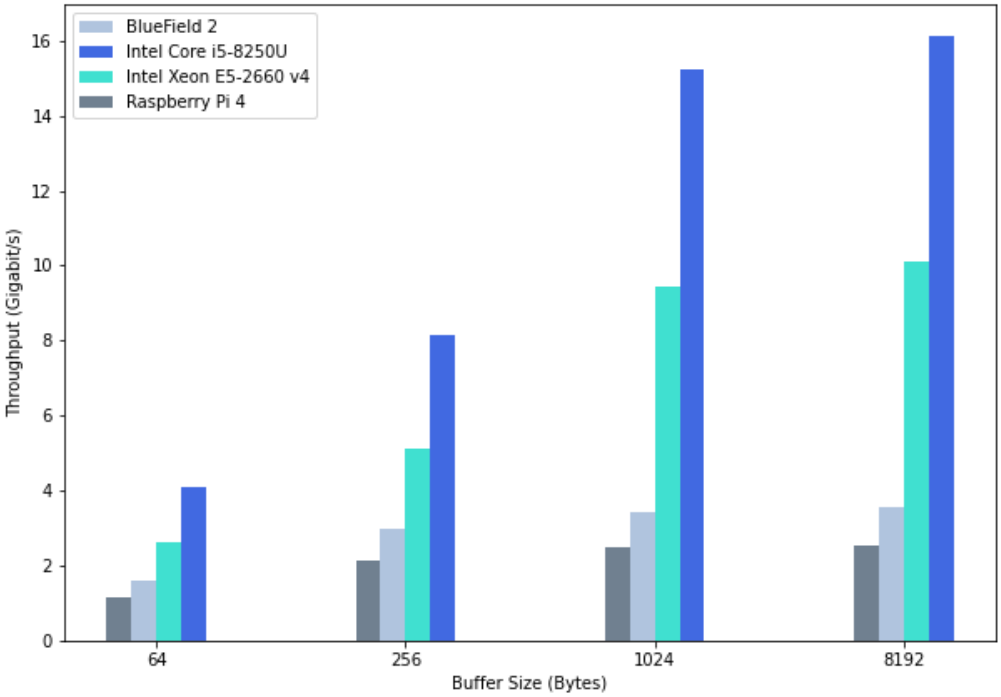

Figure 5.4.4: Throughput for different hardware using CHACHA20_POLY1305

Chapter 6

Conclusions and Future Work

The objective of this degree project was to investigate how different functions of multipath QUIC packet processing can be offloaded from the CPU or accelerated with hardware. To answer this research question, multiple frameworks, programming languages, and hardware devices have been examined. When investigating the DOCA and IPDK framework and Mount Evans IPU, little information was published. Therefore, the scouting has involved contacting Intel, NVIDIA and employees at Ericsson when seeking information. The results of this investigation was that these frameworks and the Mount Evans IPU was not mature for being used in this degree project. The results also proved that offloading, acceleration, DPUs/IPUs and QUIC have a growing interest in the industry.

To answer the question ”how can multipath QUIC be offloaded and/or accelerated“ an offloading application was designed and implemented. Two designs for the offloading application was first created. One design using the DPDK framework and another using the P4 programming language. Since these designs were very similar and the P4 design would have to rely on another tool for functionality which is not supported in the language, for example by DPDK when compiling with T4P4S, the decision was made to use a native DPDK design. After investigating, the most feasible solution to implement a offloading application for multipath QUIC was considered to be using DPDK running on a NVIDIA’s BlueField 2 DPU. The BlueField 2 card has support for hardware accelerated AES-GCM, but proved to be supported only in kernel and unreachable via DPDK. If the offloading application is running on a hardware device with a crypto engine supported in DPDK, a specific crypto device can be created and leveraged through DPDK as an input parameter to the application. This solution makes the offloading application more generalized and independent of the hardware. The list of supported crypto cards in DPDK that can be used to accelerate the AEAD algorithms used in multipath QUIC were presented.

The evaluation of the offloading application consisted of measuring the throughput, latency and jitter. During the throughput evaluation, varying the key size of AES-GCM was explored to investigate the impact of the throughput. These results showed that

the key size had a low effect on the throughput regardless of which packet size was used. The latency measurements were not presented in this thesis, because of the unstable results. When measuring the jitter percent, the results indicated a high packet loss when increasing the sending rate. The plan was originally to evaluate the offloading application using different hardware devices, with crypto engines, in order to compare how these measurements would be affected by using hardware acceleration on the offloading application. Since other cards were not accessible, the decision was made to only evaluate the application on the BlueField 2 card without crypto acceleration.

Benchmarks of DPDK crypto devices were presented. The results from Intel using QAT and AES-NI and OpenSSL showed the increased throughput that can be achieved when using a crypto hardware accelerator compared to executing crypto operations in software. In these test, only AES_GCM_128 was measured. Therefore, another benchmark was measured. This evaluation consisted of different hardware devices using the *openssl speed* utility in order to investigate the performance of these devices when using all AEAD algorithms supported in QUIC. These results also showed the increased throughput when acceleration was possible.

The key insight of our evaluation is the need for accelerating hardware and/or a high-performance CPU when offloading functionality. The gain of throughput in the measurements are significant in the cases where a crypto engine is used, compared to when not. By using an offloading application in for example a data center, the offloading application cannot become the bottleneck of the path, introducing high latency and packet loss. Without a powerful CPU or crypto engine, the performance might decrease. This enlightens the need of hardware acceleration in hardware devices, such as SmartNICs, DPUs and IPUs.

If future work were to be performed on the offloading application, additional functionally could be added. Currently all IP and MAC addresses and port numbers are set statically. To better represent reality, information to populate the Ethernet, IPv4 and UDP header would also be sent in the signaling information packets from user application on the host to the offloading application. Access to an Address Resolution Protocol (ARP) table could also be added to the offloading application. This would offload the packet encapsulation process further from the user application, since the user application would not have to be aware of MAC addresses for the next hop on the path towards its peer.

Bibliography

[1] Admanabhan, Arvind. *QUIC*. 2021. URL: https://devopedia.org/quic (visited on 06/06/2022).

[2] Amend, Markus, Brunstrom, Anna, Kassler, Andreas, Rakocevic, Veselin, and Johnson, Stephen. *DCCP Extensions for Multipath Operation with Multiple Addresses*. Internet-Draft draft-ietf-tsvwg-multipath-dccp-04. Work in Progress. Internet Engineering Task Force, Mar. 2022. 36 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-tsvwg-multipath-dccp-04.

[3] Bhalgat, Ash. *Choosing the Best SmartNIC*. 2021. URL: https://developer.nvidia.com/blog/choosing-the-best-dpu-based-smartnic/ (visited on 06/06/2022).

[4] Bonaventure, Olivier. *Apple Music on iOS13 uses Multipath TCP through load-balancers*. 2019. URL: http://blog.multipath-tcp.org/blog/html/2019/10/27/apple_music_on_ios13_uses_multipath_tcp_through_load_balancers.html (visited on 06/06/2022).

[5] Bosshart, Pat, Daly, Dan, Gibb, Glen, Izzard, Martin, McKeown, Nick, Rexford, Jennifer, Schlesinger, Cole, Talayco, Dan, Vahdat, Amin, Varghese, George, and Walker, David. "*P4: Programming Protocol-Independent Packet Processors*". In: SIGCOMM Comput. Commun. Rev. 44.3 (2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: https://doi.org/10.1145/2656877.2656890.

[6] Burres, Brad. *Intel's Hyperscale-Ready Infrastructure Processing Unit (IPU)*. Intel. 2021. URL: https://www.hc33.hotchips.org/assets/program/conference/day1/Intel%5C%20TLM%5C%20Hotchips%5C%202021%5C%20-%5C%20Mt%5C%20Evans%5C%20R2a%5C%20-%5C%20Final%5C%20version%5C%202.pdf (visited on 06/20/2022).

[7] Bursi, Alberto. *Cryptographic Hardware Accelerators*. 2020. URL: https://openwrt.org/docs/techref/hardware/cryptographic.hardware.accelerators#:~:text=A%5C%20Cryptographic%5C%20Hardware%5C%20Accelerator%5C%20can%5C%20be%5C%20integrated%5C%20into,to%5C%20the%5C%20mainboard%5C%20via%5C%20some%5C%20BUS%5C%2C%5C%20e.g.%5C%20PCI (visited on 06/06/2022).

[8] Coninck, Quentin De. "*The Packet Number Space Debate in Multipath QUIC*". In: CoRR abs/2112.01068 (2021). arXiv: 2112.01068. URL: https://arxiv.org/abs/2112.01068.

[9] Deierling, Kevin. *What Is a DPU?* 2020. URL: https://blogs.nvidia.com/blog/2020/05/20/whats-a-dpu-data-processing-unit/ (visited on 06/06/2022).

[10] DPDK. URL: https://www.dpdk.org/ (visited on 06/06/2022).

[11] DPDK. *10. Mbuf Library*. version 22.03.0. URL: https://doc.dpdk.org/guides/prog_guide/mbuf_lib.html (visited on 06/06/2022).

[12] DPDK. *3. Environment Abstraction Layer*. version 22.03.0. URL: https://doc.dpdk.org/guides/prog_guide/env_abstraction_layer.html (visited on 06/06/2022).

[13] DPDK. *Crypto Device Drivers*. version 22.03.0. URL: https://doc.dpdk.org/guides/cryptodevs/index.html (visited on 06/06/2022).

[14] Ericsson. *About us*. URL: https://www.ericsson.com/en/about-us (visited on 06/06/2022).

[15] Ford, Alan, Raiciu, Costin, Handley, Mark J., and Bonaventure, Olivier. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 6824. Jan. 2013. DOI: 10.17487/RFC6824. URL: https://www.rfc-editor.org/info/rfc6824.

[16] Hauser, Frederik, Häberle, Marco, Schmidt, Mark, and Menth, Michael. "*P4-IPsec: Site-to-Site and Host-to-Site VPN With IPsec in P4-Based SDN*". In: IEEE Access 8 (2020), pp. 139567–139586. DOI: 10.1109/ACCESS.2020.3012738.

[17] Helme, Scott. *Top 1 Million Analysis - March 2020*. 2020. URL: https://scotthelme.co.uk/top-1-million-analysis-march-2020/ (visited on 06/06/2022).

[18] Hillmancurtis. *FPGA programming and its cost comparison*. URL: https://hillmancurtis.com/fpga-programming-and-its-cost-comparison/ (visited on 06/06/2022).

[19] Intel-DPDK-Validation-team. *DPDK Intel Cryptodev Performance Report*. Mar. 2022. URL: http://fast.dpdk.org/doc/perf/DPDK_21_11_Intel_crypto_performance_report.pdf.

[20] IPDK. URL: https://ipdk.io/ (visited on 06/06/2022).

[21] IPDK. *OPI Event - IPDK and its role in enabling Open Programmable Infrastructure - Dan Daly March 15 2022*. YouTube. 2022. URL: https://youtu.be/OquYERUJcMA (visited on 06/06/2022).

[22] Iyengar, Jana and Thomson, Martin. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: 10.17487/RFC9000. URL: https://www.rfc-editor.org/info/rfc9000.

[23] Joshua, Hay, Machnikowski, Maciej, Bowers, Gregory, Wochtman, Natalia, Muniak, Joanna, and Deval, Manasi. *Accelerating QUIC via Hardware Offloads through a Socket Interface*. 2019. URL: https://legacy.netdevconf.info/0x13/session.html?talk-quic-offload.

[24] Kehr, James. *What's Quic?* 2021. URL: https://techcommunity.microsoft.com/t5/networking-blog/what-s-quic/ba-p/2683367 (visited on 06/06/2022).

[25] Keim, Robert. *What Is a Hardware Description Language (HDL)?* 2022. URL: https://www.allaboutcircuits.com/technical-articles/what-is-a-hardware-description-language-hdl/ (visited on 06/06/2022).

[26] Kennedy, Patrick. *Intel Unveils Infrastructure Processing Unit*. 2021. URL: https://www.intel.com/content/www/us/en/newsroom/news/infrastructure-processing-unit-data-center.html#gs.2htoth (visited on 06/06/2022).

[27] Kennedy, Patrick. *This Changes Networking Intel IPU Hands-on with Big Spring Canyon*. 2022. URL: https://www.servethehome.com/this-changes-networking-intel-ipu-hands-on-with-big-spring-canyon/#:~:text=IPU%5C%20is%5C%20the%5C%20Intel-specific%5C%20term%5C%20for%5C%20its%5C%20%5C%E2%5C%80%5C%9CInfrastructure,Intel%5C%20IPUs%5C%20for%5C%20networking%5C%2C%5C%20in%5C%20the%5C%20same%5C%20system. (visited on 06/06/2022).

[28] Kit, Ariel. *Programming the Entire Data Center Infrastructure with the NVIDIA DOCA SDK*. 2020. URL: https://developer.nvidia.com/blog/programming-the-entire-data-center-infrastructure-with-the-nvidia-doca-sdk/#:~:text=P4%5C%20support%5C%20is%5C%20a%5C%20component%5C%20of%5C%20DOCA%5C%20enabling,already%5C%20have%5C%20a%5C%20rich%5C%20ecosystem%5C%20of%5C%20VNF%5C%20offerings. (visited on 05/15/2022).

[29] Kordek, Alyssa. *What is Hardware Offload?* 2021. URL: https://www.inmotionhosting.com/blog/hardware-offload/ (visited on 06/06/2022).

[30] Langlet, Jonatan. *Offloading Virtual Network Functions–Hierarchical Approach*. 2020. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:kau:diva-79090.

[31] Liu, Yanmei, Ma, Yunfei, Coninck, Quentin De, Bonaventure, Olivier, Huitema, Christian, and Kühlewind, Mirja. *Multipath Extension for QUIC*. Internet-Draft draft-ietf-quic-multipath-01. Work in Progress. Internet Engineering Task Force, Mar. 2022. 28 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-multipath-01.

[32] M. Satran, S. White. *How RPC Works*. 2019. URL: https://docs.microsoft.com/en-us/windows/win32/rpc/how-rpc-works (visited on 05/15/2022).

[33] Mepits. *Hardware Description Language*. 2014. URL: https://www.mepits.com/tutorial/143/vlsi/hardware-description-language (visited on 06/06/2022).

[34] *NVIDIA BLUEFIELD-2 DPU*. NVIDIA. 2021. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf (visited on 06/20/2022).

[35] *OCTEON 10 DPU Family*. Marvell. 2021. URL: https://www.marvell.com/content/dam/marvell/en/company/media-kit/octeon-10/marvell-octeon-10-media-deck.pdf (visited on 06/20/2022).

[36] P4. URL: https://p4.org/ (visited on 06/06/2022).

[37] Papastergiou, Giorgos, Fairhurst, Gorry, Ros, David, Brunstrom, Anna, Grinnemo, Karl-Johan, Hurtig, Per, Khademi, Naeem, Tüxen, Michael, Welzl, Michael, Damjanovic, Dragana, and Mangiante, Simone. "*De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives*". In: IEEE Communications Surveys Tutorials 19.1 (2017), pp. 619–639. DOI: 10.1109/COMST.2016.2626780.

[38] Pauly, Tommy, Kinnear, Eric, and Schinazi, David. *An Unreliable Datagram Extension to QUIC*. RFC 9221. Mar. 2022. DOI: 10.17487/RFC9221. URL: https://www.rfc-editor.org/info/rfc9221.

[39] Rescorla, Eric. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[40] Rott, Jeffrey Keith. *Intel® Advanced Encryption Standard Instructions (AES-NI)*. 2012. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html#:~:text=AES-NI%5C%20can%5C%20be%5C%20used%5C%20to%5C%20accelerate%5C%20the%5C%20performance,rounds%5C%20to%5C%20produce%5C%20the%5C%20final%5C%20encrypted%5C%20cipher%5C%20text. (visited on 06/06/2022).

[41] Thomson, Martin and Turner, Sean. *Using TLS to Secure QUIC*. RFC 9001. May 2021. DOI: 10.17487/RFC9001. URL: https://www.rfc-editor.org/info/rfc9001.

[42] Valdellon, Lionel. *What is an SDK? Everything You Need to Know*. 2020. URL: https://clevertap.com/blog/what-is-an-sdk/ (visited on 06/06/2022).

[43] Viernickel, Tobias, Froemmgen, Alexander, Rizk, Amr, Koldehofe, Boris, and Steinmetz, Ralf. "*Multipath QUIC: A Deployable Multipath Transport Protocol*". In: May 2018, pp. 1–7. DOI: 10.1109/ICC.2018.8422951.

[44] Vörös, Péter, Horpácsi, Dániel, Kitlei, Róbert, Leskó, Dániel, Tejfel, Máté, and Laki, Sándor. "*T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors*". In: 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR). 2018, pp. 1–8. DOI: 10.1109/HPSR.2018.8850752.

[45] Xilinx. *What is an FPGA?* URL: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html (visited on 06/06/2022).

[46] Yang, Xiangrui, Eggert, Lars, Ott, Jörg, Uhlig, Steve, Sun, Zhigang, and Antichi, Gianni. "*Making QUIC Quicker With NIC Offload*". In: Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC. EPIQ '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 21–27. ISBN: 9781450380478. DOI: 10.1145/3405796.3405827. URL: https://doi.org/10.1145/3405796.3405827.

Appendix A

Command lines for dpdk-test-crypto-perf

```
./x86_64-native-linuxapp-gcc/app/dpdk-test-crypto-perf --socket-mem 2048,0
   --legacy-mem -a 0000:1a:01.0 -a 0000:1c:01.0 -a 0000:1e:01.0 -a 0000:1a
   :01.1 -a 0000:1c:01.1 -a 0000:1e:01.1 -a 0000:1a:01.2 -a 0000:1c:01.2 -
   a 0000:1e:01.2 -a 0000:1a:01.3 -a 0000:1c:01.3 -a 0000:1e:01.3 -a
   0000:1a:01.4 -a 0000:1c:01.4 -a 0000:1e:01.4 -a 0000:1a:01.5 -a 0000:1c
   :01.5 -a 0000:1e:01.5 --vdev crypto_scheduler_pmd_1,worker=0000:1a:01.0
   _qat_sym,worker=0000:1c:01.0_qat_sym,worker=0000:1e:01.0_qat_sym,mode=
   round-robin --vdev=crypto_scheduler_pmd_2,worker=0000:1a:01.1_qat_sym,
   worker=0000:1c:01.1_qat_sym,worker=0000:1e:01.1_qat_sym,mode=round-
   robin --vdev=crypto_scheduler_pmd_3,worker=0000:1a:01.2_qat_sym,worker
   =0000:1c:01.2_qat_sym,worker=0000:1e:01.2_qat_sym,mode=round-robin --
   vdev=crypto_scheduler_pmd_4,worker=0000:1a:01.3_qat_sym,worker=0000:1c
   :01.3_qat_sym,worker=0000:1e:01.3_qat_sym,mode=round-robin --vdev=
   crypto_scheduler_pmd_5,worker=0000:1a:01.4_qat_sym,worker=0000:1c:01.4
   _qat_sym,worker=0000:1e:01.4_qat_sym,mode=round-robin --vdev=
   crypto_scheduler_pmd_6,worker=0000:1a:01.5_qat_sym,worker=0000:1c:01.5
   _qat_sym,worker=0000:1e:01.5_qat_sym,mode=round-robin -l
   9,10,66,11,67,12,68 -n 6 -- --aead-key-sz 16 --buffer-sz
   64,128,256,512,1024,2048 --optype aead --ptest throughput --aead-aad-sz
    16 --devtype crypto_scheduler --aead-op encrypt --burst-sz 32 --total-
   ops 30000000 --silent --digest-sz 16 --aead-algo aes-gcm --aead-iv-sz
   12
```

Listing A.1: Command line argument for evaluating the QAT crypto device

```
./x86_64-native-linuxapp-gcc/app/dpdk-test-crypto-perf --socket-mem 2048,0
   --legacy-mem --vdev crypto_aesni_gcm_pmd_1 -l 9,10 -n 6 -- --aead-key-
   sz 16 --buffer-sz 64,128,256,512,1024,2048 --optype aead --ptest
   throughput --aead-aad-sz 16 --devtype crypto_aesni_gcm --aead-op
   encrypt --burst-sz 32 --total-ops 10000000 --silent --digest-sz 16 --
   aead-algo aes-gcm --aead iv-sz 12
```

Listing A.2: Command line argument for evaluating the AESNI-GCM cryto device