2021•2022

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Hardware offload van decryptie in het QUIC-protocol op een Xilinx Zynq-7000 SoC

PROMOTOR :

dr. Robin MARX

dr. ing. Jo VLIEGEN

Lowie Deferme

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

Gezamenlijke opleiding UHasselt en KU Leuven

UHASSELT KU LEUVEN

UHASSELT KU LEUVEN

2021•2022

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Hardware offload van decryptie in het QUIC-protocol op een Xilinx Zynq-7000 SoC

PROMOTOR :

dr. Robin MARX

dr. ing. Jo VLIEGEN

Lowie Deferme

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

UHASSELT KU LEUVEN

Woord vooraf

Dit is een masterproef voor de gezamenlijke opleiding Industriële Ingenieurswetenschappen van KU Leuven en UHasselt in Diepenbeek. Ze is uitgevoerd voor de onderzoeksgroep: *Emerging technologies, Systems & Security* van de KU Leuven.

Ik heb deze masterproef ervaren als een zeer uitdagend project vermits mijn voorkennis omtrent computer-netwerken niet erg uitgebreid was. Het bekomen resultaat zou dan ook nooit mogelijk geweest zijn zonder de ondersteuning van mijn promotoren: dr. ing. Jo Vliegen en dr. Robin Marx. Zo bijvoorbeeld is de expertise van dr. ing. Jo Vliegen over het ontwerpen van *hardware* in VDHL van onschatbare waarde gebleken en heeft dr. Robin Marx me ontzettend goed geholpen met zijn uitzonderlijk uitgebreide kennis over het QUIC-protocol. Verder wil ik ook Arish Sateesan nadrukkelijk bedanken om een stuk van zijn docteraatsonderzoek te delen voor de DCID-*lookup*-module. Daarnaast wil ik ook mijn medestudenten, Lennert Purnal en Jonathan Butaye, bedanken voor hun aandeel in dit project. Ik vond het een zeer aangename en leerrijke ervaring om in dit team te mogen werken. Tot slot gaat mijn dank ook uit naar iedereen die me gesteund en geholpen heeft tijdens dit eindwerk.

Lowie Deferme

Inhoudsopgave

Lijst van tabellen

Lijst van figuren

Lijst met afkortingen

Afkorting	Beschijving
ACK	*Acknowledgment*
AEAD	*Authenticated Encryption with Associated Data*
AES	*Advanced Encryption Standard*
ARM	*Acorn RISC Machine*
CAM	*Content Addressable Memory*
CID	*Connection ID*
CPU	*Central Processing Unit*
CRC	*Cyclic Redundancy Check*
DCID	*Destination CID*
DoS	*Denial of Service*
HP	*Header Protection*
HTTP	*Hypertext Transport Protocol*
IANA	*Internet Assigned Numbers Authority*
ID	*Identifier*
IO	*Input/Output*
IP	*Internet Protocol*
IPv4	*IP version 4*
IPv6	*IP version 6*
IV	*Initialization Vector*
LAN	*Local Area Network*
LSO	*Large Send Offload*
MAC	*Media Access Control*
MAC	*Message Authentication Code*
MTU	*Maximum Transmission Unit*
NIC	*Network Interface Controller*
PCI	*Peripheral Component Interconnect*
PCIe	*PCI Express*
PP	*Packet Protection*
RAM	*Random Access Memory*
RISC	*Reduced Instruction Set Computer*
RSS	*Receive Side Scaling*
RTT	*Round Trip Time*
SCID	*Source CID*
SFP	*Small Form-Factor Pluggable (transceiver)*
SoC	*System-on-a-chip*
TCP	*Transport Control Protocol*
TLS	*Transport Layer Security*
UDP	*User Datagram Protocol*

Afkorting	**Beschijving**
VLAN	*Virtual LAN*

Abstract

De derde en recentste HTTP-versie maakt gebruik van het QUIC-protocol. Omdat QUIC een geëncrypteerd, *connection-oriented* protocol is, vereist het veel processorintensieve berekeningen. Hierdoor kunnen servers slechts een beperkt aantal QUIC-connecties bedienen. Deze masterproef tracht dat te verbeteren d.m.v. *hardware offloading*.

De voornaamste bottleneck is het kopiëren van gegevens tussen *user space* en *kernel space*. Op de tweede plaats staan de cryptografische berekeningen. Controleoperaties, als derde, kunnen best in software blijven gezien de complexiteit van QUIC. Daarom is er gekozen voor een architectuur waarbij de cryptografie geoffload wordt. Het proof-of-concept-systeem ontleedt inkomende 1-RTT-QUIC-pakketten en controleert of er sleutels beschikbaar zijn op basis van de *destination connection ID*. Deze *lookup* gebeurt met behulp van een *probabilistic data structure*. Vervolgens worden de sleutels uitgelezen zodat het pakket gedecrypteerd kan worden. Tot slot wordt het ontcijferde pakket doorgegeven aan het *operating system*.

De maximale klokfrequentie van de proefoptelling, bepaald voor een Zync-7000 SoC, is 83,33 MHz. Dit levert een *throughput* van 2,67 Gbit/s op. De proefopstelling introduceert 756 ns extra *latency*. Verder duurt de decryptie van een 88 bytes lang QUIC-pakket in hardware 1,16 µs, terwijl de geteste softwaresystemen er gemiddeld 1,59 µs en 5,69 µs over doen. Er kan dus besloten worden dat de proefopstelling aantoont dat een *hardware offload* voor decryptie van 1-RTT-QUIC-pakketten een snelheidswinst oplevert.

Abstract in English

The third and most recent version of HTTP uses the QUIC protocol encapsulated by UDP. Since QUIC is an encrypted, connection oriented protocol, it requires processor intensive calculations. This causes servers to only be able to serve a limited number of connections. Therefore this master's thesis aims to reduce CPU load by implementing a hardware software codesign of the QUIC protocol.

The main bottleneck is the copy of data between user space and kernel space. Second, are the cryptography operations. Control operations are third but should remain in software due to the complex nature of QUIC. Hence the proposed architecture focuses on offloading cryptography to hardware. The proof-of-concept system parses incoming 1-RTT QUIC packets and checks whether or not keys are available based on destination connection ID. This check is implemented using a probabilistic data structure. Next, keys are fetched and the packet is decrypted after which it is handed over to the operating system.

The maximum clock frequency of the test set-up with a Zync-7000 SoC is 83.33 MHz. This results in a throughput of 2.67 Gbit/s and 756 ns of extra latency. Furthermore, decryption of an 88-byte long packet takes 1.16 µs in hardware while the two tested software systems needed 1.59 µs and 5.69 µs on average. Therefore, the test set-up proves that offloading decryption of 1-RTT QUIC packets improves performance. However, further research is needed in order to make this system deployable in the field.

Hoofdstuk 1

Inleiding

Het internet maakt veelvuldig gebruik van het *HyperText Transfer Protocol* (HTTP). Bij hedendaagse diensten staat er in deze afkorting vaak nog een S waardoor HTTPS gevormd wordt, deze S staat voor *Secure*. Sinds 2016 is er een voorstel om HTTPS niet bovenop *Transport Layer Security* (TLS) en *Transport Control Protocol* (TCP) maar via het relatief nieuwe QUIC en *User Datagram Protocol* (UDP) te laten werken [1]. Dit voorstel heeft geleid tot de ontwikkeling van een nieuwe HTTP versie: HTTP/3 [2].

QUIC is dus een relatief nieuw protocol dat enkele voordelen heeft vergeleken met het oude systeem van TCP + TLS. Zo lost QUIC het TCP-*head-of-line-blocking*-probleem op en belooft het een snellere *connection*-setup. QUIC heeft echter ook nadelen: het is complexer dan TLS + TCP en kost veel tijd op de processor. Verder zijn hedendaagse QUIC-implementaties volledig uitgevoerd in *user space* terwijl moderne systemen TCP in de *kernel* behandelen. Deze nadelen hebben tot gevolg dat de *throughput* per connectie snel daalt bij meerdere concurrente connecties. Daardoor moeten diensten sneller opschalen en meer *servers* gebruiken indien ze grote hoeveelheden *cliens* willen bedienen over HTTP/3.

Daarom tracht deze masterproef om een deel van de processorintensieve berekeningen uit te voeren in *dedicated hardware* zodat de *server resources* beter benut kunnen worden. Concreet zal dit werk een architectuur voorstellen om de cryptografische berekeningen van het QUIC-protocol in hardware te behandelen met behulp van een Xilinx Zynq-7000 SoC.

Deze masterproef begint met een theoretische achtergrond in Hoofdstuk 2. Daarna maakt Hoofdstuk 3 hiervan gebruik om to een QUIC-accelerator te komen. Omdat deze QUIC-accelerator relatief uitgebreid is en niet geïmplementeerd kon worden in het tijdsbestek van een masterproef geeft Hoofdstuk 4 een overzicht van de werkelijke proefopstelling. De resultaten van deze proefopstelling zijn dan voorgesteld in Hoofdstuk 5. Tot slot geeft Hoofdstuk 6 een besluit deze masterproef.

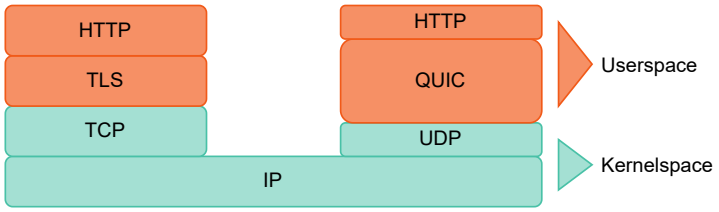
Hoofdstuk 2

Literatuurstudie

2.1 QUIC

2.1.1 Introductie

Volgens [3] is QUIC altijd geëncapsuleerd in UDP. Dit is staat in contrast met oudere HTTPS-*stacks* die gebruik maakten van TLS over TCP. Figuur 2.1 geeft deze verandering weer.



Figuur 2.1: Overview van een typische HTTP *stack*.

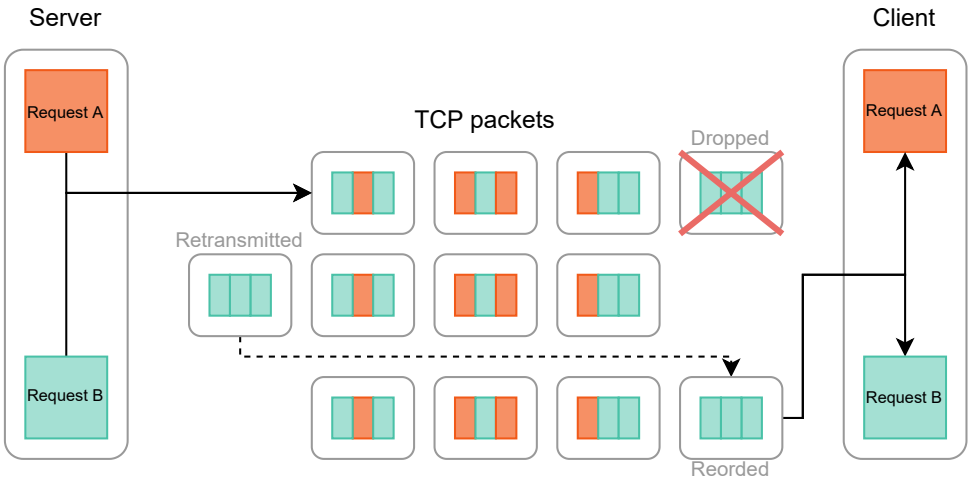
Het relatief nieuwe QUIC biedt een aantal voordelen in vergelijking met TCP, zo belooft het een snellere *connection setup* [3]. Verder lost het het *head of line blocking* probleem op. *Head of line blocking* komt voor bij HTTP/2 omdat meerdere *requests* over één TCP connectie gemultiplexed kunnen worden [4]. Wanneer er nu een pakket verloren gaat is de volledige TCP connectie geblokkeerd totdat het verloren pakket opnieuw verzonden en aangekomen is. Omdat het mogelijk is dat het verloren pakket niet van alle *requests* data bevat, blokkeert het deze *requests* dus onnodig zoals weergegeven in Figuur 2.2. Een *out-of-order* pakket heeft hetzelfde effect totdat het aangekomen en verwerkt is.

QUIC lost dit op via het concept van *streams*. Dit zijn abstracties die de applicatie kan gebruiken om data te verzenden/ontvangen. Aangezien *streams lightweight* zijn kan er voor iedere nieuwe *request* gemakkelijk en snel een *stream* aangemaakt worden. Indien er dan een QUIC-pakket verloren gaat, zal de QUIC-implementatie enkel die *streams* blokkeren waarvan er data zat in het verloren pakket [3].

2.1.2 Pakketten en frames

QUIC maakt een onderscheid tussen *packets* en frames waarbij frames de *payload* van een *packet* zijn. De UDP *payload* bestaat dus uit één of meerdere[1] QUIC-*packets* waarin één

[1]Sommige pakket-types kunnen samengevoegd worden in één enkel datagram



Figuur 2.2: *Head of line blocking* bij TCP door een verloren pakket

of meerdere QUIC-frames zitten [3].

Pakketten

Packets zijn geëncrypteerd en geauthenticeerd. [3] definieert zes soorten QUIC pakketten:

1. *Initial* pakketten worden gebruikt tijdens de *connection setup* zoals beschreven in paragraaf 2.1.3
2. *Zero round-trip time* (0-RTT) pakketten dienen om tijdens de *connection setup* meteen applicatie data te versturen, nog voor de eerste *round trip* gebeurd is.
3. *Handshake* pakketten bevatten o.a. *crypto frames* en worden gebruikt tijdens de cryptografische *handshake*.
4. *Retry* pakketten dienen om het adres van de *client* te verifiëren.
5. *Version negotiation* pakketten bieden de mogelijkheid om een specifieke QUIC versie af te spreken tussen zender en ontvanger.
6. *One round-trip time* (1-RTT) pakketten zijn het meest voorkomend. Nieuwe data wordt in dit geval na één *round-trip* verzonden.

[3] definieert twee soorten *header*-formaten voor pakketten: *short header* en *long header*. Deze formaten zijn respectievelijk voorgesteld in Tabel 2.1 en Tabel 2.2. De enige pakketsoort die gebruik maakt van de *short header* is 1-RTT. Tot de *long-header*-pakketten kunnen dus alle overige soorten gerekend worden. Over welk soort pakket het gaat, is bepaald door het *long-packet-type*-veld. De enige uitzondering hierop is het *version-negotiation*-pakket, dat is gedefinieerd door een *version*-veld dat steeds `0x00000000` is.

Frames

Zoals paragraaf 2.1.5 bespreekt gebeurt encryptie op niveau van pakketten. Verder zal paragraaf 2.3.1 bespreken waarom controle operaties best in software uitgevoerd blijven. De verschillende typen frames zijn dan ook niet zo relevant in het kader van deze masterproef.

Tabel 2.1: Formaat van een *short header* [3]

Veld	**Lengte (bits)**	**Beschermd**
Header form	1	
Fixed bit (altijd '1')	1	
Spin bit	1	
Reserved	2	x
Key phase	1	x
Packet number length	2	x
Destination connection ID	0 .. 160	
Packet number	8 .. 32	x
Payload	..	

Tabel 2.2: Formaat van een *long header* [3]

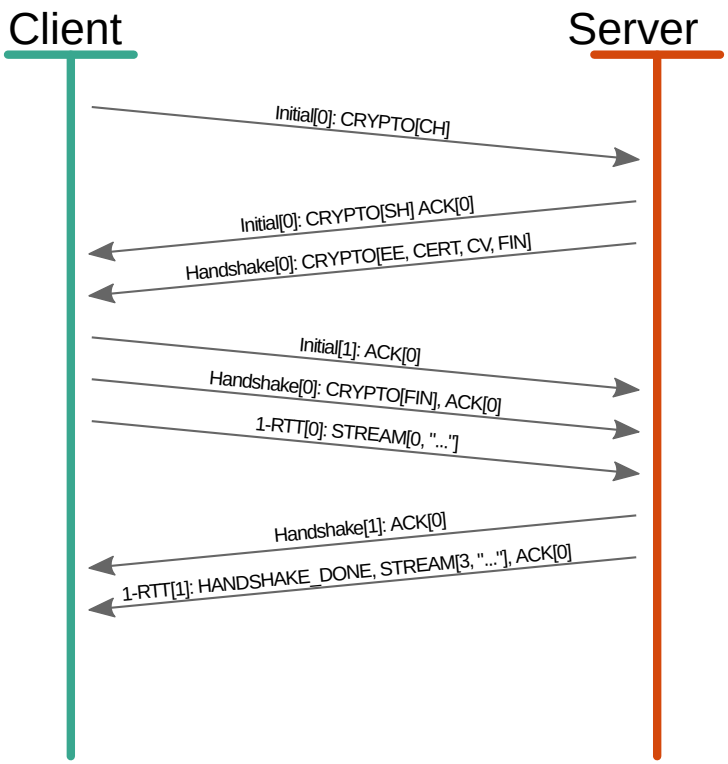
Veld	**Lengte (bits)**
Header form	1
Fixed bit (altijd '1')	1
Long packet type	2
Type-specific bits	4
Version	32
Destination connection ID length	8
Destination connection ID	0 .. 160
Source connection ID length	8
Source connection ID	0 .. 160
Payload	..

2.1.3 Connections

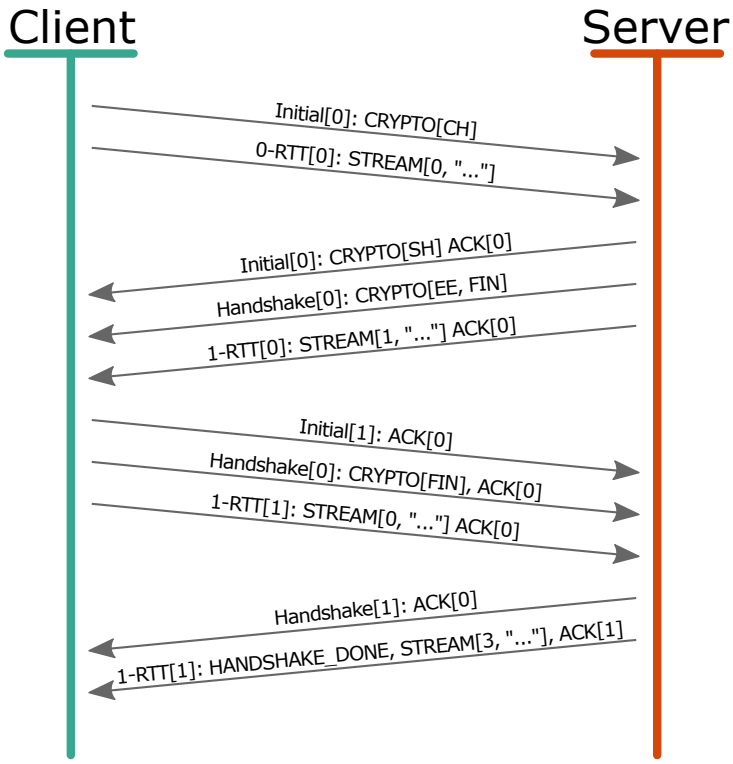
Een QUIC connectie kan op twee manieren tot stand komen: in 1-RTT en in 0-RTT [3]. In het eerste geval, weergegeven in Figuur 2.3, heeft de *client* geen informatie over de *server* en zal die een *initial*-pakket sturen dat een *client-hello*-crypto-frame bevat. Dit wordt beantwoord door de *server* met een *initial*-pakket dat naast een *server-hello*-crypto-frame ook een *acknowledgment*-frame bevat. Verder stuurt de server een *handshake*-pakket dat verdere cryptografische info bevat. Wanneer dit aankomt bij de *client*, is alle info beschikbaar en kan er geëncrypteerde data naar de server worden gestuurd met behulp van 1-RTT-pakketten. De *handshake* wordt dan gelijktijdig vervolledigd. Het duurt dus één *round trip* vooraleer er data verzonden kan worden [3].

In het geval van 0-RTT heeft de *client* nog info over de *server* uit een voorgaande connectie [3]. Figuur2.4 toont dat er naast het *initial*-pakket met het *client-hello*-crypto-frame nu ook een 0-RTT-pakket met geëncrypteerde data verzonden wordt. De *server* reageert met een met *server-hello*-crypto-frame in een *initial*-pakket, een *handshake*-pakket met extra cryptografische info en een 1-RTT-pakket dat geëncrypteerde data bevat. Nadat het *initial*- en het *handshake*-pakket aangekomen zijn, heeft de *client* alle info om 1-RTT-pakketten te encrypteren en decrypteren. Er kan dus al meteen data verzonden worden zonder te wachten op een *round trip*.

Vanaf het moment dat de connectie tot stand gekomen is, worden er enkel nog 1-RTT-pakketten gebruikt voor het verzenden van applicatie data [3]. Dit type zal dus het globaal gezien het meest voorkomen.



Figuur 2.3: 1-RTT QUIC *connection establishment*



Figuur 2.4: 0-RTT QUIC *connection establishment*

2.1.4 Connection ID's

Iedere partner in een QUIC connectie heeft een *Source Connection ID* (SCID) en een *Destination Connection ID* (DCID) [3]. Beide partners, *client* en *server*, kiezen hun eigen SCID en delen deze tijdens de *connection setup*. Voor de ontvangende partner is de net ontvangen SCID de DCID van de andere partner voor deze connectie.

Connection IDs (CID's) worden hoofdzakelijk gebruikt om de selectie van een *endpoint* onafhankelijk te maken van de adressen in de onderliggende protocollen [3]. Zo kunnen bijvoorbeeld het IP adres en UDP poort van een partner veranderen zonder dat de connectie verloren gaat. Verder kan de lengte van CID's variëren van 0 tot 20 bytes. Op die manier moeten er niet telkens lange DCIDs verzonden worden. Indien een partner op voorhand weet dat zijn IP adres en UDP poort niet zullen veranderen, kan deze partner zelfs de *zero-length* SCID kiezen. De lengte van de CID's is meegegeven in de *header* van *long header* pakketten zoals Tabel 2.2 illustreert. Bij *short header* pakketten is de lengte niet aanwezig aangezien de ontvanger zijn eigen CID al kent voor de zender 1-RTT pakketten kan verzenden met deze CID.

Indien iedere partner slechts één CID heeft per connectie kan een waarnemer een verandering in de adressen van onderliggende protocollen volgen aangezien de DCID ongeëncrypteerd verzonden wordt [3]. Dit zou een waarnemer bijvoorbeeld in staat stellen om te ontdekken dat een gebruiker zich eerst op een netwerk bevindt in Hasselt (op basis van het IP adres), vervolgens verandert naar een netwerk in Brussel (mogelijks een mobiel netwerk) om tot slot, nogmaals te veranderen naar een netwerk in Leuven. Dit is enkel indien de gebruiker regelmatig data uitwisselt met de *server* van een applicatie via een QUIC-connectie. Daarom voorziet [3] het gebruik van meerdere CID's per partner die na de initiële *connection setup* aangeleverd kunnen worden door een partner met behulp van het *new-connection-id*-frame. Natuurlijk mag er geen correlatie zijn tussen de CID's van een partner. Anders zou een waarnemer de connectie nog steeds kunnen volgen.

Soms, bijvoorbeeld voor *load balancing*, moeten *middleboxes* de DCID's kunnen lezen. Dit is niet triviaal aangezien DCID lengtes variabel zijn en niet mee worden verzonden in de *header* van 1-RTT-pakketten. Daarom moeten, in dat geval, *endpoints* samenwerken met deze *middleboxes* door bijvoorbeeld altijd DCID's met een gegeven lengte te kiezen of door een codeerschema af te spreken om de lengte in de DCID zelf te encoderen [3].

2.1.5 Crypto

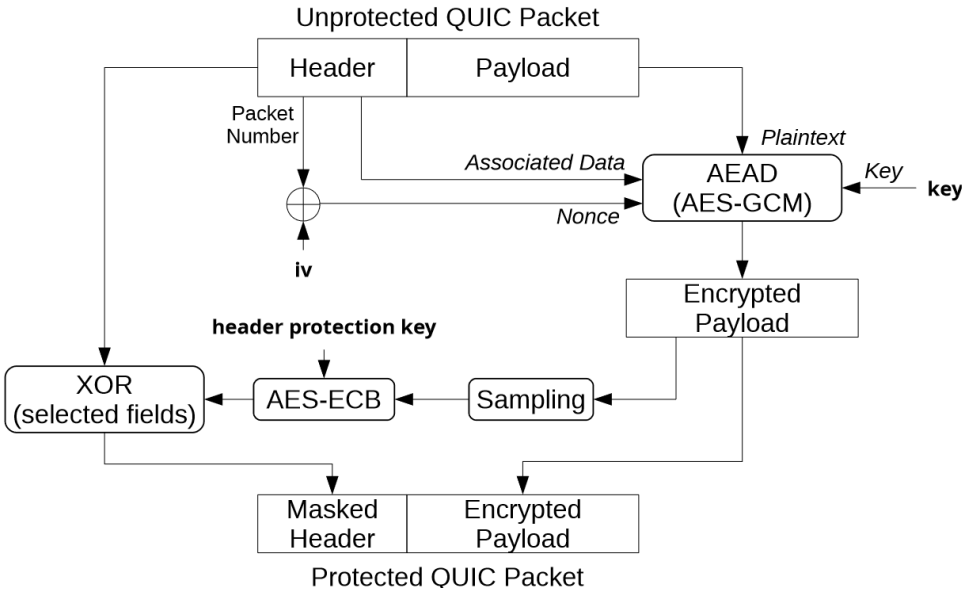
Zoals paragraaf 2.1.2 beschrijft, bestaan QUIC-pakketten uit een *header* en een *payload*. Waarbij de laatste bestaat uit QUIC-frames. Encryptie in QUIC gebeurt op het niveau van pakketten. Concreet betekent dit dat de *payload* als één blok data beschouwd wordt, ongeacht het aantal en type van de frames dat het bevat [5].

Packet protection

Encryptie van een QUIC-pakket, zoals beschreven in [5], gebeurt in twee stappen: eerst wordt de *payload* geëncrypteerd met behulp van een *Authenticated Encryption with Associated Data* (AEAD)-algoritme. Beide stappen zijn weergegeven in Figuur 2.5. De volledige *header*, van de *flags*-byte tot en met het pakket nummer, wordt meegegeven als *associated data*. Dat wil zeggen dat deze data wel ingevoerd wordt in de AEAD-functie waardoor de interne *state* veranderd, maar dat ze niet geëncrypteerd wordt. QUIC ondersteunt alle *cipher suites* uit [6] met uitzondering van `TLS_AES_128_CCM_8_SHA256`. Dit heeft tot gevolg dat de AEAD-tag, ook wel *Message Authentication Code* (MAC) genoemd, steeds 16 bytes lang is [5].

Daarna wordt de *header* gemaskeerd in de *header protection* (HP)-stap door een XOR uit te voeren tussen de beschermde header-velden, aangeduid in Tabel 2.1, en een masker [5]. Dat masker is bekomen door encryptie van de bemonsterde *payload*. Dit monster bestaat

uit 16 bytes en begint met de 4de byte na de start van het *packet nummer*[2]. Het gebruikte encryptie-algoritme is afhankelijk van de gekozen *cipher suite*.

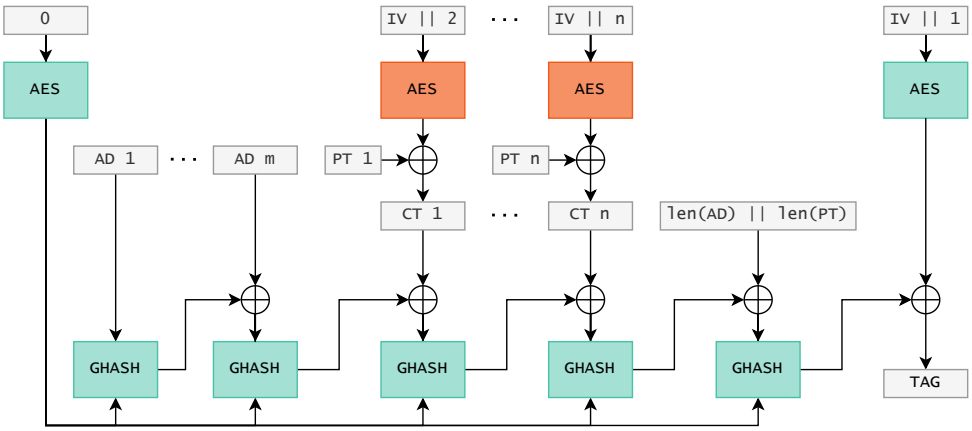


Figuur 2.5: Encryptie van een QUIC-Pakket [7, p. 5]

AES-128-GCM

Om te onderzoeken of het mogelijk is QUIC te versnellen met behulp van een *hardware offload*, beperkt deze masterproef zich tot de `TLS_AES_128_GCM_SHA256` *cipher suite*. Daarom is hieronder kort uitgelegd welke *ciphers* er in dat geval gebruikt worden om een QUIC-pakket te encrypteren.

Volgens [5] zullen AES-128-gebaseerde-*suites* AES-128 in *Electronic Codebook* (ECB) mode gebruiken voor HP. Bij deze *cipher suite* hoort het `AEAD_AES_128_GCM` algoritme [6]. Dit is AES met sleutellengtes van 128-bit in *Galois Counter Mode* (GCM) [8]. Figuur 2.6 geeft een blokschema van dit algoritme weer.



Figuur 2.6: Blokschema van de AES-GCM architectuur

[2]Op deze manier zullen er zich op de plaats van het monster altijd *payload*-bytes bevinden, ongeacht de lengte van het *packet number*.

Key update

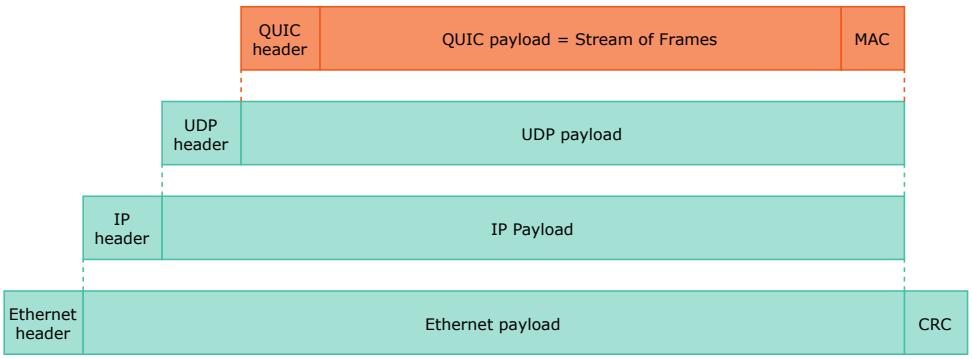
Algoritmes voor encryptie hebben een maximale hoeveelheid data die versleuteld mag worden met één dezelfde sleutel. Volgens [8] is dit voor `AEAD_AES_128_GCM` gelijk aan $2^{36} - 32$ bytes. Verder definieert [8] ook een limiet op de hoeveelheid *associated data* waarvan de integriteit gegarandeerd kan worden, dit is $2^{61} - 1$ bytes. Daarom voorziet [5] de mogelijkheid om een *key update* te doen.

Een *key update* gebeurt door de *key-phase*-bit uit Tabel 2.1 te veranderen [5]. Indien een *endpoint* een QUIC-pakket ontvangt met een hoger pakket-nummer en een andere *key phase* dan het pakket met het hoogste nummer dat reeds verwerkt is, moet er een *key update* gebeuren. De nieuwe sleutel wordt berekend door middel van de *Key Derivation Function* (KDF) uit TLS. Merk op dat dit enkel de sleutel en IV voor *packet protection* update, de sleutel voor *header protection* blijft gelijk. Dit is nodig aangezien het *key-phase*-veld zelf beschermd is door de *header protection*.

Omdat een nieuwe sleutel afleiden door middel van de KDF tijd kost, kan er info gelekt worden indien de nieuwe sleutel pas berekend wordt wanneer er een pakket met nieuwe *key phase* arriveert. Daarom schrijft [5] dat implementaties steeds de huidige en de volgende sleutels beschikbaar moeten hebben. Dit biedt ook een oplossing voor volgende aanval: aangezien *key phase* slechts gemaskeerd is d.m.v. een XOR kan een aanvaller, door de gemaskerde bit te veranderen, de *key phase* veranderen waardoor de implementatie een nieuwe sleutel moet afleiden. Dit zou dan gebruikt kunnen worden voor een *Denial-of-service* (DoS)-aanval indien de implementatie de nieuwe sleutel niet bijhoudt als "volgende sleutel" waardoor de berekening steeds opnieuw moet gebeuren [5]. Merk op dat de nieuwe sleutel in dit geval nooit als "huidige sleutel" opgeslagen zou worden, omdat de AEAD-tag bij deze aanval niet klopt.

2.2 Onderliggende protocollen

Zoals beschreven in paragraaf 2.1.1 bevindt het QUIC protocol zich bovenop UDP. UDP is typisch de *payload* van een IP pakket wat over Ethernet verzonden kan worden. Figuur 2.7 geeft een voorbeeld van dergelijke encapsulatie weer.



Figuur 2.7: Een voorbeeld van pakket encapsulatie met Ethernet, IP, UDP en QUIC

Om een pakket in *hardware* te behandelen moeten eerst de lagen van de onderliggende protocollen deels geïnterpreteerd worden. Daarom geeft het volgende hoofdstuk een overzicht van de protocollen die typisch onder QUIC gebruikt worden.

2.2.1 Ethernet

In het geval van ethernet moet er eerst een onderscheid gemaakt worden tussen ethernet *packets* en *frames*.

Een *packet* is datgene dat volledig over de fysieke infrastructuur verzonden wordt en bestaat uit een *preamble*, *start frame delimiter*, *ethernet frame* en *interpacket gap* [9]. Ethernet *frames* kunnen dus beschouwd worden als de *payload* van een ethernet pakket.

Origineel bestaan deze frames uit vijf stukken zoals weergegeven in Tabel 2.3 [9]. Ten eerste is er een veld van 48 bits lang dat het MAC adres van de ontvanger bevat. Daarna wordt ook het MAC adres van de zender meegegeven. Verder zijn er 16 bits ter beschikking om aan te geven welk protocol er zich in de *payload* bevindt. De *IEEE Registration Authority* definieert de waarde van het *EtherType* veld voor een welbepaald protocol, bijvoorbeeld voor IPv4 zullen deze bits de waarde `0x0800` hebben [10].

Tabel 2.3: Ethernet *frame* formaat zonder VLAN [9]

Veld	**Lengte (bits)**
Destination MAC address	48
Source MAC address	48
EtherType	16
Payload	..
Frame check sequence	32

Sinds 1998 is er met de introductie van VLAN een veld van 32 bits bijgekomen [11]. Dit resulteert in een ethernet-*frame*-formaat zoals weergegeven in Tabel 2.4. Het onderscheid tussen een pakket met en zonder VLAN kan gemaakt worden door de 13^{de} en 14^{de} byte te lezen (waar zich zonder VLAN het *EtherType* bevindt). Tot slot, is het door middel van *IEEE 802.1ad* mogelijk om twee VLAN lagen te gebruiken [12].

Tabel 2.4: Ethernet *frame* formaat met VLAN [12]

Veld	**Lengte (bits)**
Destination MAC address	48
Source MAC address	48
VLAN	32
Ethertype	16
Payload	..
Frame check sequence	32

2.2.2 Internet protocol

Er bestaan twee versies van het IP-protocol: IPv4 en IPv6. Het voornaamste verschil tussen beide is de lengte van de gebruikte adressen. IPv4 gebruikt namelijk adressen van 4 bytes lang terwijl een IPv6-adres 16 bytes lang is. Om een onderscheid te maken tussen beide kunnen de vier meest beduidende bits van de *header* gebruikt worden. De waarde `0x4` op deze plaats duidt op een IPv4 pakket terwijl `0x6` duidt op een IPv6 pakket [13], [14]. Verder kan het verschil ook gemaakt worden met behulp van het *EtherType*-veld van ethernet wat `0x0800` zal zijn voor IPv4 en `0x86dd` voor IPv6 [10]. Hieronder zullen, van beide versies, de *header*-velden die van belang zijn voor deze masterproef besproken worden.

Tabel 2.5 geeft het formaat van een IPv4-*header* weer. Na het versie veld volgt de *Internet Header Length* (IHL). Om de IPv4-*header*-lengte (in bytes) te berekenen moet de waarde van deze vier bits met 4 vermenigvuldigd worden [9]. Verder geeft een IPv4-*header* het protocol van de IP-*payload* aan door middel van het protocol-veld [13]. Aangezien deze masterproef enkel QUIC behandelt, zal dit het UDP-protocol (`0x11`) zijn [15].

Tabel 2.5: Formaat van een IPv4 *header* [13]

Veld	**Lengte (bits)**
Version	4
Internet Header Length	4
Type of service	8
Total Length	16
Identification	16
Flags	3
Fragment Offset	13
Time to Live	8
Protocol	8
Header Checksum	16
Source Address	32
Destination Address	32
Options	0 .. 320
Padding	0 .. 24

Aangezien een IPv6-*header* altijd 40 bytes lang is, heeft het geen IHL-veld [14]. Wel bevat de *header* een veld om het volgende protocol aan te geven. Dit veld wordt het *next-header*-veld genoemd. Normaal gesproken zou dit voor een QUIC-pakket UDP (`0x11`) zijn, maar het zou ook om een IPv6-*extension-header* kunnen gaan. Dergelijke uitbreidingen dienen hetzelfde doel als het IPv4-optie-veld en bevatten extra informatie [16]. De *extension headers* (EH) bevatten elk zelf ook een *next header field* als eerste byte. Sommige EH-types hebben een variabele lengte en bevatten daarom de *header extension length* als tweede byte [16].

Tabel 2.6: Formaat van een IPv6 *header* [14]

Veld	**Lengte (bits)**
Version	4
Traffic Class	8
Flow Label	20
Payload Length	16
Next Header	8
Hop Limit	8
Source Address	128
Destination Address	128

2.2.3 User Datagram Protocol

UDP, gedefinieerd door [17], is een licht protocol dat de gebruiker voorziet van het concept van poorten. De UDP *header*, weergegeven door Tabel 2.7, is dan ook relatief klein met enkel een *source port*, *destination port*, *payload lenght* en een *checksum*.

Poort nummers kunnen een waarde hebben van 0 tot en met 65535 ($2^{16}-1$) waarbij poorten 0 tot 1023 *well-known* zijn [17]. Het gebruik van *well-known* poorten is gedefinieerd en gelimiteerd door IANA [18].

Tabel 2.7: UDP *header* formaat [17]

Veld	**Lengte (bits)**
Source port	16
Destination port	16
Lengte	16
Checksum	16

2.3 NIC Offload

2.3.1 Het belang van offloaden

In hedendaagse servers worden sommige netwerk gerelateerde taken zoals *packet segmentation* en het berekenen van *checksums* niet door de CPU uitgevoerd maar door de NIC [19]. Dit heeft als voordeel dat de CPU een groter gedeelte van de tijd kan spenderen aan applicatie taken [20].

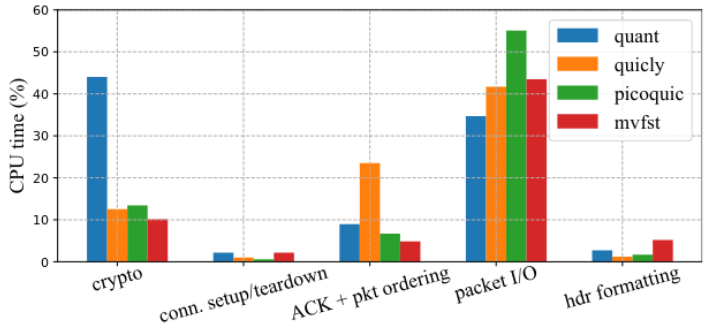
Specifiek over QUIC beschrijft [21] de voordelen om cryptografie en *packet segmentation* in de NIC te verwerken. Verder presenteert [22] drie belangrijke lessen in verband met het *offloaden* van QUIC:

1. Het kopiëren van gegevens tussen *user space* en *kernel space* kost ongeveer 50% van het QUIC-gerelateerde CPU gebruik.
2. Zoals weergegeven in Figuur 2.8 is 40% van het CPU gebruik te linken aan cryptografie indien de overhead voor het kopiëren van gegevens afwezig is (dit werd bereikt via *kernel-bypass* technieken).
3. *Packet reordening* is vaak te traag wat er voor zorgt dat *out-of-order* pakketten als verloren worden beschouwd. Hierdoor zal de *congestion window* kleiner worden.

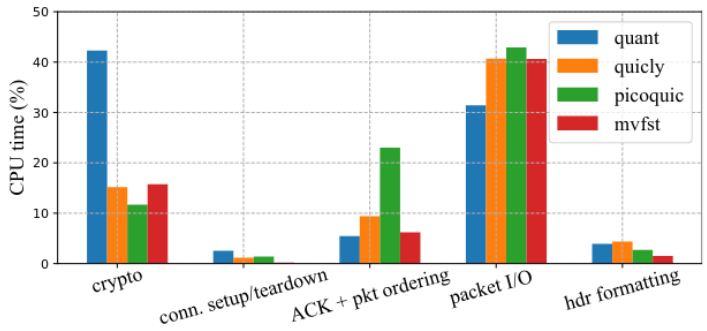
Uit deze lessen leidt [22] drie ontwerp richtlijnen af. Ten eerste: ontlast de CPU door het offloaden van de cryptografie. Zeker encryptie en decryptie via het AEAD algoritme omdat deze functies veel CPU tijd verbruiken en *stateless* zijn. Ten tweede: gebruik *dedicated hardware* voor het herordenen van pakketten. Hierbij is het belangrijk dat de decryptie al (in hardware) is uitgevoerd aangezien delen van de *header* en de volledige *payload* van een pakket geëncrypteerd zijn [5]. Tot slot: laat controle operaties zoals *version negotiation* en de TLS *handshake* in de CPU afgehandeld worden omdat offloaden hier geen significante winst zal veroorzaken.

2.3.2 Offload architecturen

Intel Corp beschrijft in [23] een architectuur om een deel van het QUIC-protocol te offloaden. Deze architectuur bevat, naast voor de hand liggende onderdelen zoals CPU, geheugen, etc. ook een accelerator. Deze accelerator kan fysiek deel uitmaken van de NIC of, zoals in Figuur 2.9, apart uitgevoerd zijn. De accelerator moet de *Network Interface Controller* in staat stellen om drie functies uit te voeren: Ten eerste kan de NIC een grotere *Maximum Transmission Unit*(MTU) dan de werkelijke MTU van het netwerk



(a) server

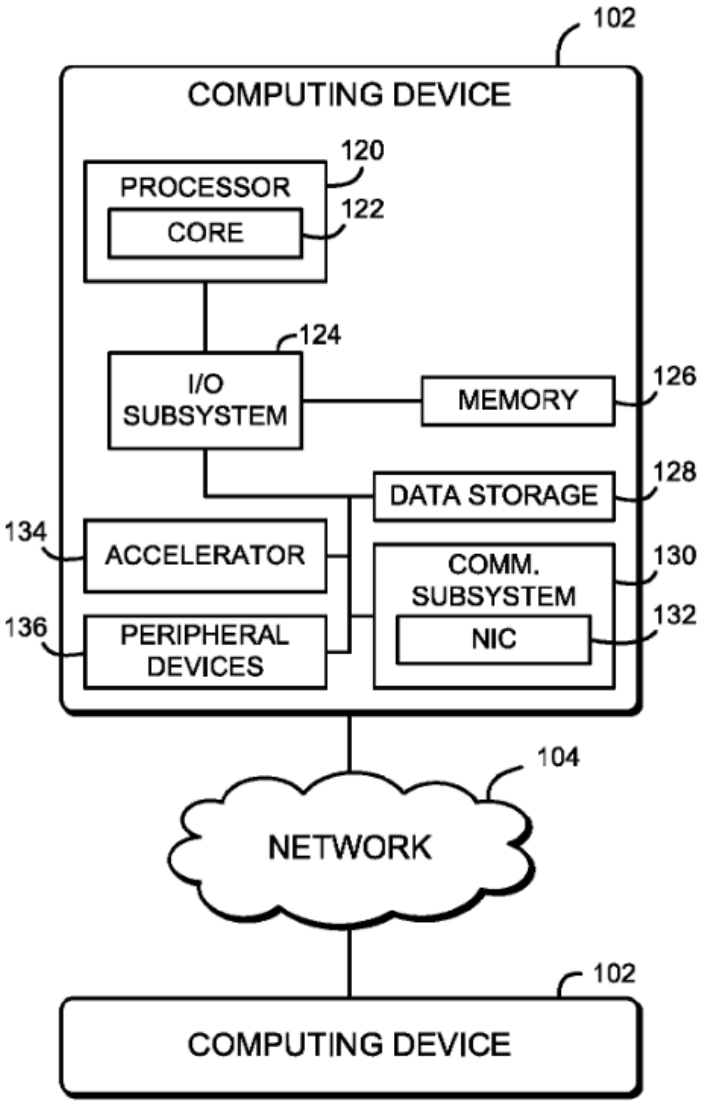


(b) client

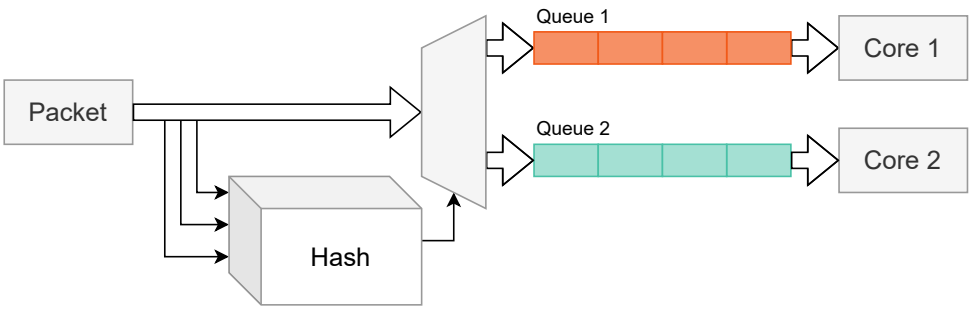
Figuur 2.8: QUIC gerelateerd CPU gebruik in *client* en *server* systemen. [22, p. 2]

adverteren en de data daarna opdelen in pakketten die even groot of net iets kleiner dan de netwerk MTU zijn. Dit wordt *large send offload* genoemd [24]. Ten tweede moet de NIC ook in staat zijn tot *receive side scaling*. Dit betekent dat er voor iedere processor-*core* een *queue* bijgehouden wordt waarin pakketten voor die *core* zich opstapelen. Om te bepalen aan welke *queue* een nieuw pakket toegevoegd moet worden, wordt vaak een *hash*-systeem gebruikt zoals Figuur 2.10 illustreert [25]. Tot slot bevat de architectuur van [23] ook een *cryptographic accelerator* beschikbaar zijn die de mogelijkheid biedt om veelgebruikte cryptografische functies te versnellen.

Bij het offloaden van netwerk taken is het belangrijk een onderscheid te maken tussen inkomende en uitgaande pakketten. Het verwerken van inkomende QUIC pakketten moet volgens [23] als volgt gebeuren: Eerst moet de NIC in staat zijn om binnenkomende QUIC pakketten te kunnen herkennen. Daarna moet, indien mogelijk, het pakket gedecrypteerd worden. Tot slot stroomt de data via RSS tot aan de processor-*cores* waar ze verwerkt wordt. Uitgaande QUIC pakketten worden eerst door middel van LSO naar de NIC gestuurd. De *cryptographic accelerator* helpt de NIC met het encrypteren van het pakket zodat het uiteindelijk verzonden kan worden over het netwerk.



Figuur 2.9: Een systeem voor de acceleratie van QUIC *packet processing* [23, p. 15]



Figuur 2.10: Principe van *Receive Side Scaling*

Hoofdstuk 3

QUIC accelerator

Dit hoofdstuk bespreekt een ontwerp van een QUIC-*accelerator* in een NIC. Deze accelerator moet aan vier belangrijke eisen voldoen. Het systeem moet namelijk:

1. In staat zijn om de cryptografische berekeningen te maken aangezien deze veel tijd van de processor vragen.[1]
2. 1-RTT-pakketten kunnen behandelen aangezien deze het meest voorkomend zijn.[2]
3. Efficiënt zijn met het encrypteren van uitgaande pakketten aangezien een typische *webserver* meer data moet verzenden dan zal ontvangen.
4. Een *throughput* halen die minstens even groot is als een software-QUIC-implementatie in *user space*.

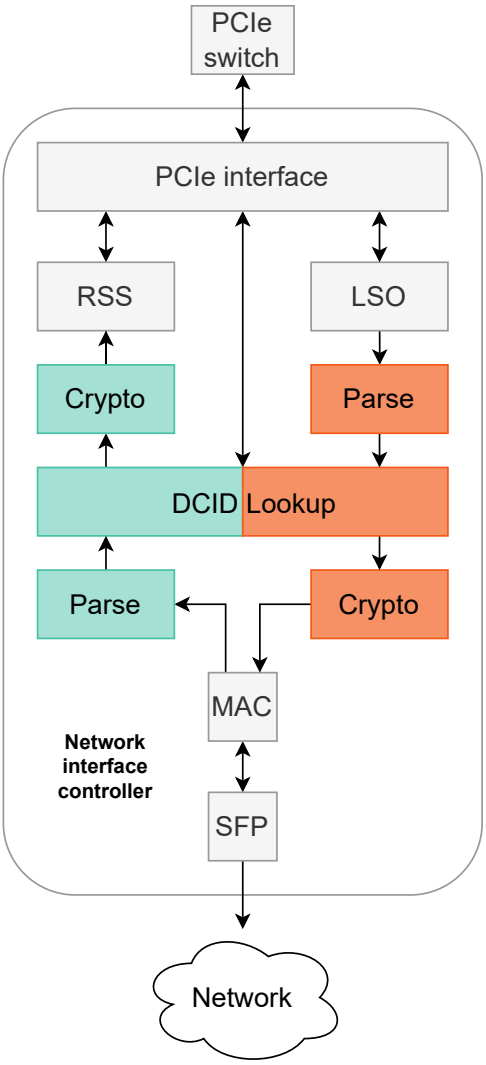
3.1 Overzicht

Zoals Figuur 3.1 illustreert is de NIC verbonden met het netwerk door middel van een SFP-*tranceiver*. Dergelijke componenten werken met een datastroom die door de MAC-component vertaald worden naar discrete ethernet *frames* en omgekeerd. Verder doet de MAC-chip vaak een eerste filtering op basis van het MAC-adres. Daarboven bevindt zich de QUIC-*accelerator* die ervoor moet zorgen dat de CPU's minder tijd verliezen aan het verwerken van QUIC data. De QUIC *accelerator* is naast de MAC ook verbonden met een RSS- en een LSO-module. Tot slot is de NIC d.m.v. een PCIe-*interface* verbonden met de rest van de server.

De RSS-module determineert voor welke *CPU core* binnenkomende pakketten bedoeld zijn. Zoals paragraaf 2.3.2 reeds vermeldt, maakt het hierbij gebruik van een *hash*-functie. In het geval van QUIC moet de DCID aan de input van deze *hash* toegevoegd worden zodat verschillende connecties efficiënt op verschillende CPU-*cores* verwerkt kunnen worden in *mutli-threaded* QUIC-implementaties.

Large Send Offload dient, zoals paragraaf 2.3.2 bespreekt, om te voorkomen dat er regelmatig kleine hoeveelheden data gekopieerd worden van het OS naar de NIC. De meest voor de hand liggende manier is door één groot UDP-pakket door te geven naar de NIC. De *payload* van dit pakket bestaat dan uit verschillende QUIC pakketten. De LSO-module zal dan de UDP-*header* uit het grote pakket vóór elk QUIC pakket plaatsen. Merk op

[1] Zie paragraaf 2.3.1
[2] Zie paragraaf 2.1.3



Figuur 3.1: Blokschema een QUIC accelerator in een NIC

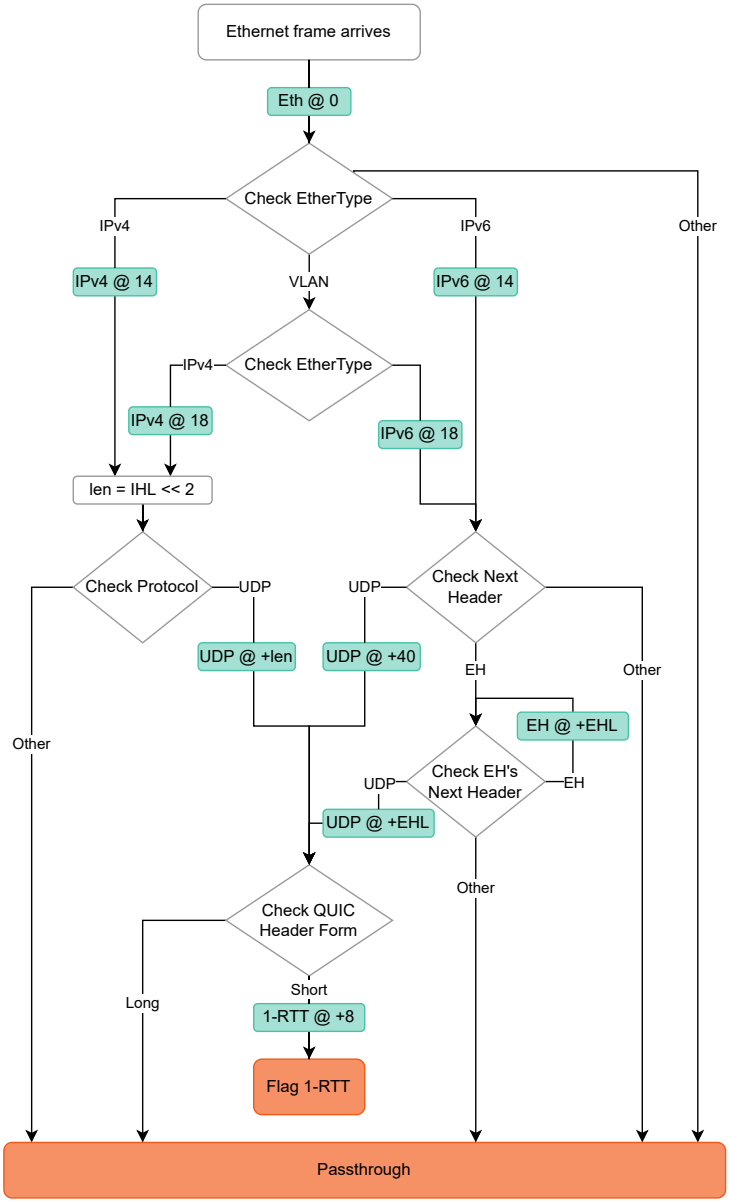
dat dit verschilt met een conventionele UDP-LSO-module aangezien de splitsing moet gebeuren tussen de QUIC-pakketten en niet bij de *Maximum Transmission Unit*.

De QUIC *accelerator* bestaat uit drie belangrijke delen. Ten eerste, een **parser** die bytes in het pakket markeert op basis van hun functie zodat de achterliggende modules extra info hebben over de data die ze behandelen. Ten tweede bevat de accelerator een ***DCID-lookup*-module**. Deze module zoekt op basis van de DCID of er voor het huidige pakket cryptografische sleutels beschikbaar zijn. Indien er sleutels beschikbaar zijn worden ze doorgegeven aan de **crypto-module**. Het doel van deze module is de encryptie/decryptie van het QUIC pakket.

3.2 Pakket parsing

De belangrijkste taak van de *packet parser* is nagaan of het een 1-RTT-QUIC-pakket betreft en, in dat geval, markeren waar het QUIC pakket begint (dit is hetzelfde als de *flags byte* aanduiden). Verder kan de parser ook markeren waar de *Message Authentication Code* (MAC) van het QUIC pakket zich bevindt. Zo kunnen achterliggende modules simpelweg de markering gebruiken in plaats van zelf 16 bytes terug te moeten tellen van de laatste byte.

Waar de *parser* deze markeringen moet aanbrengen, kan bepaald worden door de *headers* van de onderliggende protocollen te interpreteren zoals Figuur 3.2 illustreert. In een typisch systeem zijn deze protocollen Ethernet, IP en UDP. Dit stuk over *packet parsing* zal dan ook van deze van *protocol stack* uitgaan. Hieronder staat voor elk van deze protocollen besproken welke *header*-velden gecontroleerd moeten worden om het QUIC-pakket correct te kunnen markeren. Merk op dat byte nummers op basis van nul genummerd zijn, bijvoorbeeld: met "byte 9" wordt dus de 10^{de} byte bedoeld.



Figuur 3.2: Flowchart van het controle pad in de *parser*

3.2.1 Ethernet

Zoals paragraaf 2.2.1 beschrijft, bevat de ethernet *header* een *EtherType* veld (bytes 12 en 13) waarin het volgende protocol beschreven is. Aangezien dit IP is voor een QUIC pakket, bevat dit veld de waarde `0x0800` (IPv4) of `0x86dd` (IPv6) [10]. Het kan ook zijn dat het een VLAN ethernet pakket betreft en in dat geval zal er de waarde `0x8100` te vinden zijn [12]. Het werkelijke *EtherType* veld bevindt zich dan op bytes 16 en 17.

3.2.2 Internet Protocol

Eens ethernet gecontroleerd is en bepaald is waar het IPv4 pakket begint kan met behulp protocol-veld (byte 9 in de IPv4-*header*) bepaald worden of het volgende protocol UDP (`0x11`) is [13], [17]. De lengte van IPv4 *headers* is variabel en is gegeven door het *Internet Header Length* (IHL) veld.

In het geval van IPv6 is de *header lenght* altijd 40 bytes maar bevat de *header* een *next header* veld om aan te geven wat er volgt [14]. Dit is normaal gesproken UDP (`0x11`) bij een QUIC pakket maar kan ook een IPv6 *extension header* (EH) zijn. Deze extensies bevatten elk ook een *next header* veld dat aangeeft wat erachter komt. Sommige *extension-header*-types hebben een variabele lengte. Deze wordt dan meegegeven in het *Extension Header Length field* (EHL) [16].

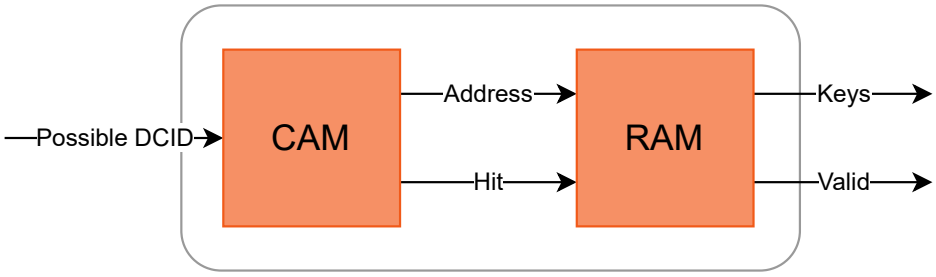
Een goede *parser* moet deze velden interpreteren om er zeker te zijn van de *protocol stack* en om te kunnen aanduiden waar het UDP-pakket begint.

3.2.3 User Datagram Protocol

De UDP *header*, weergegeven in Tabel 2.7, heeft geen veld om aan te geven wat het volgende protocol is [13]. Ieder UDP pakket waarvan de *payload* dus lijkt op een QUIC pakket[3] zou dan als QUIC beschouwd worden. Dit is echter geen probleem omdat, om *zero-length* DCID's te ondersteunen, het IP adres en UDP poort meegegeven worden naar de DCID-*lookup*. Er zal geen hit zijn indien het IP adres en de UDP poort niet op voorhand door de software aan de NIC doorgegeven zijn en dus wordt het pakket in dat geval onveranderd doorgegeven.

3.3 DCID lookup

De tweede module in de QUIC accelerator is de *DCID Lookup*. Deze module bepaalt tot welke connectie een pakket behoort en leest de bijhorende AEAD-sleutels uit zodat ze beschikbaar zijn voor de crypto-module. Dit is weergegeven in Figuur 3.3. Verder moet deze module ook markeren welke bytes tot de DCID behoren aangezien ze een variabele lengte hebben en latere modules hierover geen info meer hebben in het geval van 1-RTT-pakketten.



Figuur 3.3: Basisschema van de DCID-*lookup*-module

In essentie doet de DCID *lookup* module dus een *key-value lookup*. De software zal de juiste combinaties opslaan wanneer er 1) een nieuwe connectie opgezet is, 2) nieuwe DCID's beschikbaar zijn voor een bestaande connectie, 3) het IP adres of UDP poort veranderd zijn of 4) bij een *key phase* verandering.

[3]Een QUIC pakket heeft '1' op de plaats van de *fixed bit* zoals Tabel 2.2 en Tabel 2.1 illustreren.

3.3.1 Content Addressable Memory

De meest voor de hand liggende manier om dergelijke *key-value lookups* te doen is met behulp van een *Content Addressable memory* (CAM). Het meest eenvoudige CAM-type, *Binary Content Addressable memory* (BCAM), geeft de *value* terug indien de *key* volledig overeenkomt. Dit is niet erg efficiënt om DCID's op te zoeken aangezien, zoals paragraaf 2.1.4 beschrijft, DCID's een variabele lengte hebben[4]. Om ervoor te zorgen dat alle BCAM-sleutels toch even lang zijn, moeten kortere DCID's aangevuld worden met bijvoorbeeld `0x0`. De variabele DCID-lengte betekent ook dat de hardware voor iedere mogelijke lengte, dus 21 keer, moet nagaan of er sleutels bekend zijn. Indien deze opzoekingen sequentieel gebeuren d.m.v. één BCAM resulteert dit in een maximale *latency* gelijk aan $t_{lookup} \cdot 21$. Door meerdere BCAM's parallel te gebruiken is het mogelijk om de *latency* te verkleinen. In het geval van maximale parallellisatie heeft de *lookup* een *latency* van t_{lookup}, maar zijn er 21 BCAM's nodig die dus 21 keer zoveel oppervlakte innemen. Een meer genuanceerde oplossing betreft een compromis tussen *latency* en *area*. Zo kunnen bijvoorbeeld drie BCAM's gebruikt worden voor een maximale *latency* van $t_{lookup} \cdot 7$.

In de plaats van een BCAM-*setup* kan er ook gebruik gemaakt worden van een Ternaire CAM (TCAM). Dit soort CAM laat toe om naast de *key* ook een masker mee te geven dat aangeeft welke bits/bytes er gebruikt moeten worden in de *lookup* [26]. Een TCAM-systeem heeft als voordeel dat het mogelijk is om te zoeken naar een *prefix* van de 20 bytes die mogelijk een DCID zijn in plaats van een exacte match [27]. Maar, aangezien er nergens in [3] beschreven is dat een DCID geen *prefix* mag zijn van een andere DCID, moet er gezocht worden naar de *longest matching prefix*. Aangepaste ternaire CAM's worden door [28] gebruikt voor *longest prefix matching* in $log_2(N)$ waarbij N het maximaal aantal *keys* is dat gedeeltelijk matcht.

Longest matching prefix levert in alle gevallen, behalve twee, het correcte resultaat op. Bij de eerste uitzondering komt het toevallig voor dat de DCID en het geëncrypteerde *packet number* toevallig overeen komen met een andere DCID. Dit resulteert steeds in ongeldige sleutels waardoor het pakket verloren gaat. Ten tweede is er het probleem van de *zero-lenght*-DCID, die een *prefix* is van alle mogelijke *connection IDs*, waardoor alle pakketten waarvan de DCID niet in de CAM zit toch (foute) AEAD-*keys* krijgen[5]. Hierdoor zal de *crypto*-module ten onrechte proberen om deze pakketten te decrypteren. De oplossing hiervoor is, zoals paragraaf 3.2 reeds aanhaalde, een *lookup key* te gebruiken die niet enkel bestaat uit de DCID maar ook het IP-adres en de UDP-poort bevat.

Tabel 3.1 geeft de uiteindelijke structuur van de *key-value pairs* weer. De voorbeeld data zijn zo gekozen dat rij drie en vier een applicatie illustreren die gebruik maakt van twee verschillende QUIC-connecties. Rij twee en drie geven een voorbeeld van een connectie met meerdere DCID's en rij één is een connectie van een andere applicatie die op een *socket* met een IPv6-adres luistert. Verder illustreren de voorbeelden uit Tabel 3.1 dat de *lookup keys* behoorlijk groot kunnen worden. De maximale lengte ervan, 304 bits, is berekend door Formule 3.1.

[4]DCID's kunnen 0 tot en met 20 bytes lang zijn

[5]Enkel in het geval dat er een *zero-length ID* in de CAM zit.

Tabel 3.1: Voorbeeld van *key-value pairs* in de CAM

Key			***Value***
IP adres	UDP poort	DCID	RAM adres
2001:4ea8:c4d9:8e1c::	4578	`0xd81a78`	`0x00`
192.168.1.97	3443	`0xb36001e2`	`0x48`
192.168.1.97	3443	`0xe784fed8`	`0x48`
192.168.1.97	3443	`0x0198d84f78`	`0x90`
...	...	...	...

$$\begin{aligned} \text{maxlen}(key) &= \text{maxlen}(address) + \text{maxlen}(port) + \text{maxlen}(DCID) \\ &= 16 \text{ bytes} + 2 \text{ bytes} + 20 \text{ bytes} \\ &= 38 \text{ bytes} = 304 \text{ bits} \end{aligned} \tag{3.1}$$

3.3.2 Random Acces Memory

Zoals paragraaf 2.1.5 reeds vermeld moet de *packet protection key* voor de volgende *key phase* (KP) reeds berekend zijn voordat er een pakket met binnenkomt met deze KP. Op die manier ontstaat er geen *timing side-channel* waaruit afgeleid kan worden of er al dan niet een *key* update heeft plaats gevonden. Deze vereiste impliceert dat de RAM voor beide *phases* sleutels moet bijhouden. Tabel 3.2 geeft weer welke data de RAM per connectie moet bijgehouden.

Tabel 3.2: Data per connectie in de RAM

Data	**Lengte (bits)**
Header protection key	128
Packet protection key (KP_0)	128
Initialization Vector (KP_0)	96
Packet protection key (KP_1)	128
Initialization Vector (KP_1)	96

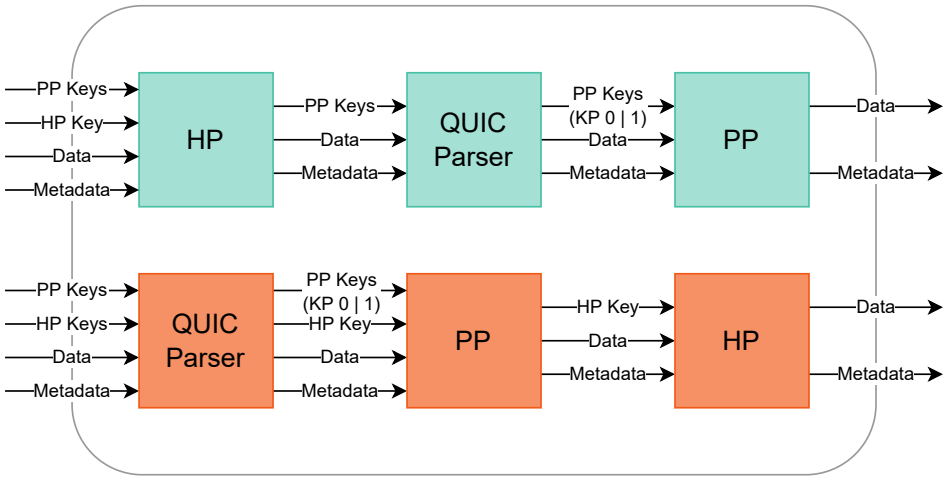
3.4 Crypto

De crypto module is zonder twijfel de belangrijkste module uit de QUIC-accelerator vermits er gesteld kan worden dat de twee andere modules enkel ter ondersteuning dienen. Ze zorgen er namelijk voor dat de crypto-module voldoende info heeft om het QUIC-pakket te encrypteren/decrypteren.

Zoals paragraaf 2.1.5 bespreekt, ondergaat een pakket twee crypto-stappen: *header protection* (HP) en een AEAD-stap die in dit hoofdstuk *packet protection* (PP) genoemd zal worden. Aangezien *header*-encryptie na PP gebeurt, zal dit bij decryptie omgekeerd zijn. Dat is geïllustreerd is Figuur 3.4 waarin het bovenste, blauwe pad decryptie en het oranje pad encryptie voorstelt.

3.4.1 Decryptie

In het geval van decryptie komt de data, samen met de resultaten van de DCID-*lookup* module, aan bij de HP-stap. In deze stap wordt eerst het 16-byte lange monster ge-



Figuur 3.4: Basisschema van de crypto-module

nomen zoals beschreven in paragraaf 2.1.5. De HP-module encrypteert dit monster via `AES_128_ECB`[6] waardoor er een *keystream* ontstaat. De XOR van deze *keystream* met delen van de geëncrypteerde *header* levert dan de *plaintext header* op. Nu kan de QUIC-*parser* het pakket nummer interpreteren en aanduiden waar de *payload* begint. Verder kan er nu ook op basis van de *key phase* besloten worden welke PP-sleutels er gebruikt moeten worden. Daarna zal de *payload* gedecrypteerd worden in de PP-module met behulp van `AEAD_AES_128_GCM`[7]. Tot slot wordt de AEAD-tag gecontroleerd. Een pakket waarvan de tag niet klopt is vermoedelijk niet betrouwbaar en moet gedropt worden. Wel moet er in dat geval een teller, eventueel per afzender/connectie, verhoogd worden om te kunnen ontdekken wanneer het systeem een DoS-aanval meemaakt.

3.4.2 Encryptie

In het geval van encryptie krijgt de crypto-module een *plaintext*-pakket aan. Hiervan moet dan bepaald worden hoe lang het pakket-nummer is en waar de payload begint. In principe zou dit in de algemene *parser* kunnen gebeuren maar aangezien dit voor decryptie niet mogelijk is, is ervoor gekozen om dit voor beide paden in een QUIC-*parser* in de crypto-module te doen.

Daarna kan het pakket geëncrypteerd worden m.b.v. het AEAD-algoritme. Vervolgens kan het juiste monster genomen worden om tot slot de *header* te maskeren. Indien een pakket niet geëncrypteerd moet worden omdat het reeds in software gebeurd is m.b.v. een ander algoritme, zal de DCID-*lookup*-module geen hit vinden. Er zullen dan dus geen sleutels beschikbaar zijn waardoor het pakket onveranderd doorgegeven wordt.

3.4.3 Controle

Zoals paragraaf 3.4.1 aanhaalt, bevat de crypto-module een teller die het aantal ondecrypteerbare pakketten bijhoudt. Natuurlijk heeft dit geen enkele zin als de software deze teller niet kan uitlezen. Daarom zal de waarde van deze teller beschikbaar gesteld worden via de PCIe interface. Verder kan de software de teller ook op nul zetten indien dit nodig geacht wordt.

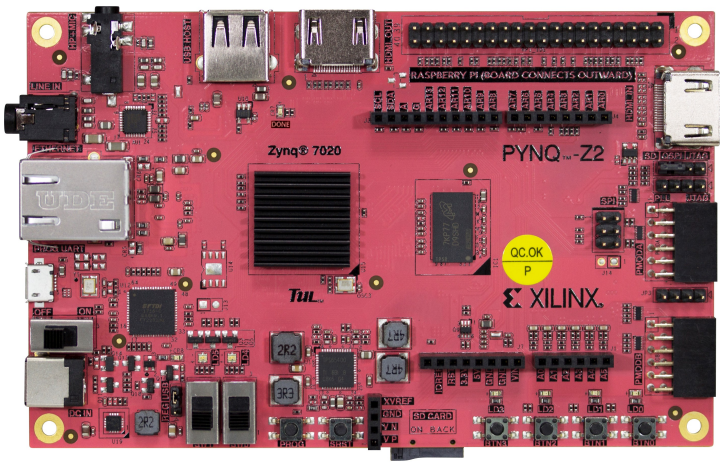
[6] Enkel voor AES-128-gebaseerde TLS-*suites*
[7] Indien TLS_AES_128_GCM_SHA256 de gekozen *cipher suite* is

Hoofdstuk 4

Ontwerp van de proefopstelling

4.1 Overzicht

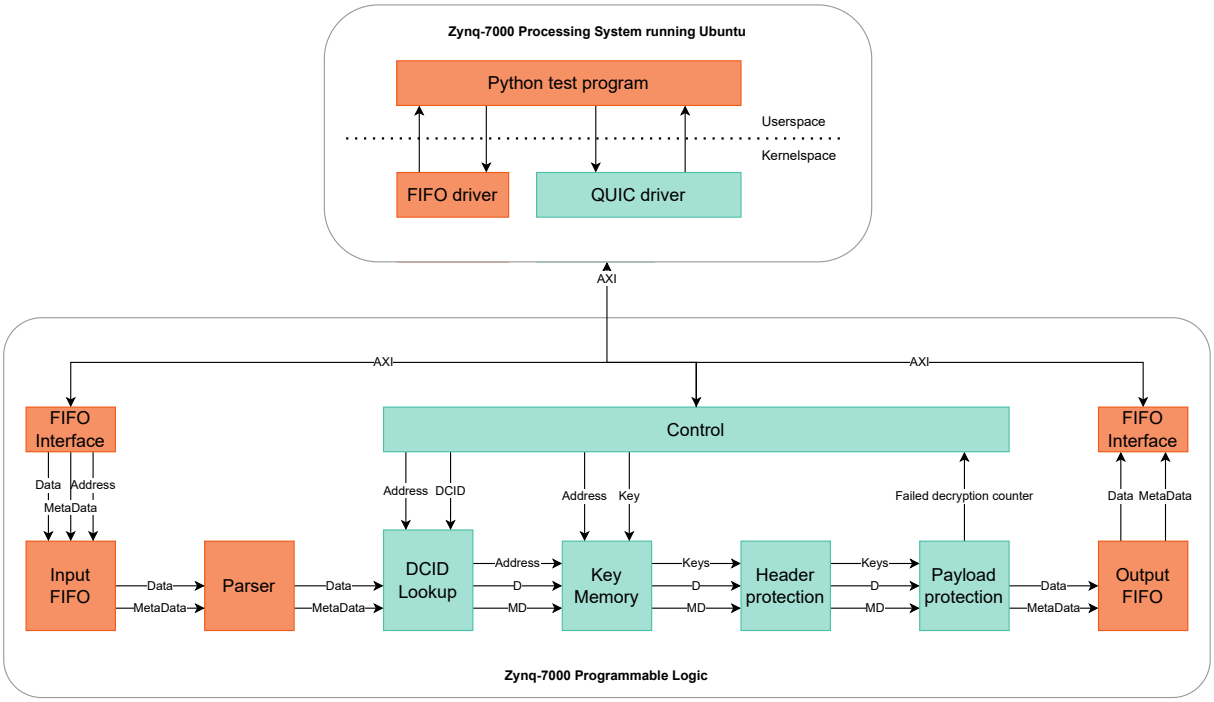
Figuur 4.1 beeldt een PYNQ™-Z2 *development board* af omdat deze masterproef dit bord gebruikt om de proefopstelling te bouwen. Het bord bevat een Zynq-7020 SoC van Xilinx [29]. Deze SoC's hebben als voordeel dat ze naast *Programmable Logic* (PL) ook een *Processing System* (PS) bevatten. Dit PS bestaat uit een *dual-core ARM Cortex-A9 processor* wat het mogelijk maakt om er een Ubuntu-Linux versie op uit te voeren [30].



Figuur 4.1: TUL PYNQ™-Z2 *development board* [29]

Dankzij de mogelijkheid om Linux te gebruiken, kon proefopstelling in theorie bestaan uit een stuk *dedicated hardware* (in de PL) en een bestaande, maar aangepaste, QUIC-implementatie die op het PS draait. In praktijk is de integratie met de bestaande QUIC-implementatie niet gemaakt wegens tijdsgebrek. In de plaats daarvan bestaat de proefopstelling, weergegeven in Figuur 4.2, uit een QUIC-accelerator die pakket-data krijgt via een *input*-FIFO en terug aan de software geeft door middel van een *output*-FIFO. De modules van de proefopstelling geven de pakket-data parallel door per 32 bits. Verder is ervoor gezorgd dat de data en metadata in een module dezelfde vertraging ondervinden dan de module nodig heeft om zijn taak uit te voeren. Bijvoorbeeld: als de *key memory* vijf klok-cycli nodig heeft om alle sleutels op te halen van een bepaald adres zal de (meta)data in deze module evenveel cycli vertraging oplopen.

Uit Figuur 4.2 is verder op te maken dat de proefopstelling enkel uit het decryptie-



Figuur 4.2: Blokschema van de proefopstelling zonder controle signalen

pad bestaat. De keuze om enkel arriverende pakketten te behandelen is gemaakt omdat software de DCID van deze pakketten kiest waardoor de hardware dus slechts voor één, vaste, DCID-lengte moet werken. Deze lengte is in de proefopstelling gelijk aan 20 bytes, de maximale lengte van een QUIC-DCID volgens [3].

4.2 Ondersteunende systemen

De proefopstelling is opgedeeld in twee delen: een QUIC-accelerator en een deel hulpsystemen om de test te kunnen uitvoeren. Deze hulpsystemen zijn aangeduid met oranje in Figuur 4.2. Om een test uit te voeren zal het pythonprogramma de geëncrypteerde pakket-data in de *input*-FIFO schrijven d.m.v. de FIFO-*driver*. Verder zullen ook de DCID's en bijhorende sleutels d.m.v. de QUIC-*driver* naar de accelerator geschreven worden. Eens dit gebeurd is, krijgt de *input*-FIFO een *drive-enable*-commando waardoor de data aan de *parser* ingevoerd wordt.

4.2.1 Data generatie

Een data-generator vormt testdata in JSON-formaat, zie Bijlage A, om naar een *binary string* formaat zodat de *simulation testbench* en het *python test program* er mee overweg kunnen. Dit formaat bevat 32 databits en 6 metadatabits per regel. De databits stellen 4 bytes van het pakket voor terwijl de 6 metadatabits opgebouwd zijn uit een "*data valid*", "*data last*" en "*data strobe*".

Om de omzetting te vergemakkelijken maakt de data-generator gebruik van een python-bibliotheek die *Scapy* noemt. Volgens [31] kan *Scapy* pakketten van verscheidene protocollen aanmaken en decoderen. Het QUIC-protocol was echter wel nog niet gedefinieerd. Daarom hebben we een uitbreiding gemaakt voor 1-RTT-QUIC-pakketten. Deze uitbreiding is gebaseerd op Bijlage A van [7] en is weergegeven in Bijlage B van dit werk.

Naast het genereren van pakket-data produceert de generator ook twee extra bestanden: één met de DCID's van de pakketten in de data en één met bijhorende sleutels voor deze DCID's. Zo kunnen de *testbench* en het *python test programma* de hardware van deze gegevens voorzien.

4.2.2 Parser

De *parser* uit de proefopstelling kan ook beschouwd worden als een hulpsysteem omdat er namelijk vanuit gegaan wordt dat de onderliggende protocollen Ethernet zonder VLAN, IPv4 zonder extra opties en UDP zijn. Op die manier volstaat een simpele teller, in plaats van de volwaardige *parser* uit hoofdstuk 3.2, om de protocollen te markeren. Algoritme 1 illustreert deze teller. Verder zal de *parser* ook de 20 bytes van de DCID markeren m.b.v. dezelfde teller.

Algoritme 1 Protocol determinatie in de proefopstelling

```
c ← 0
while not end-of-packet do
  if valid(c) = 0 then
    byte(c) ← padding
  else
    if c < 14 then
      byte(c) ← ethernet
    else if c < 34 then
      byte(c) ← ip
    else if c < 42 then
      byte(c) ← udp
    else
      byte(c) ← quic
    end if
    c ← c + 1
  end if
end while
```

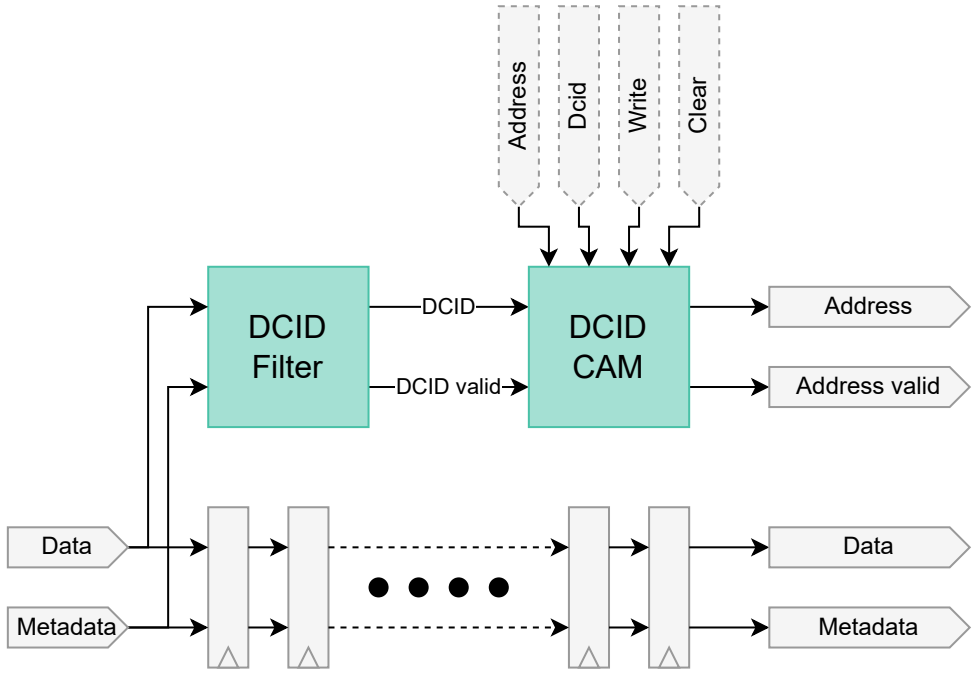
4.3 QUIC accelerator

4.3.1 DCID lookup

Deze DCID-*lookup* module verschilt met die van de voorgestelde accelerator uit paragraaf 3.3 omdat het CAM en RAM gedeelte om praktische redenen apart uitgevoerd zijn zoals Figuur 4.2 illustreert. De DCID-*lookup*-module van de proefopstelling heeft een gelijkaardige functie als de CAM uit paragraaf 3.3.1 en is weergegeven in Figuur 4.3.

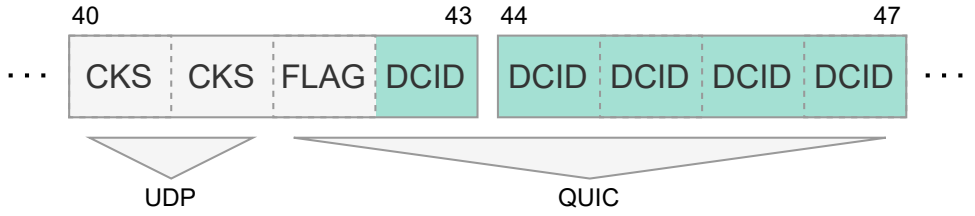
DCID filter

Nadat de *parser* 20 bytes gemarkeerd heeft als DCID, filtert de DCID-filter-module deze uit de data. Aangezien de data aan een snelheid van vier bytes per klokperiode arriveert, duurt het minstens vijf klok-cycli voor de volledige DCID is ingelezen. Vermits de DCID niet noodzakelijkerwijs uitgelijnd is op vier bytes zullen er in 75% van de mogelijkheden zes cycli nodig zijn. Zoals Algoritme 1 illustreert, gaat de proefopstelling uit van een vaste *protocol stack* waardoor de DCID zich steeds op dezelfde *offset* bevindt. Figuur 4.4 toont



Figuur 4.3: Werking van de DCID-*lookup*-module

dat de 44e byte steeds de eerste byte van de DCID zal zijn. In de proefopstelling zal de DCID dus nooit uitgelijnd zijn op vier bytes en zullen er altijd zes klok-cycli nodig zijn. Om tijdens deze cycli aan te geven dat de DCID op de *output* niet geldig is, wordt een *DCID-valid*-signaal gebruikt.



Figuur 4.4: Vaste *offset* van de DCID bij de proefopstelling

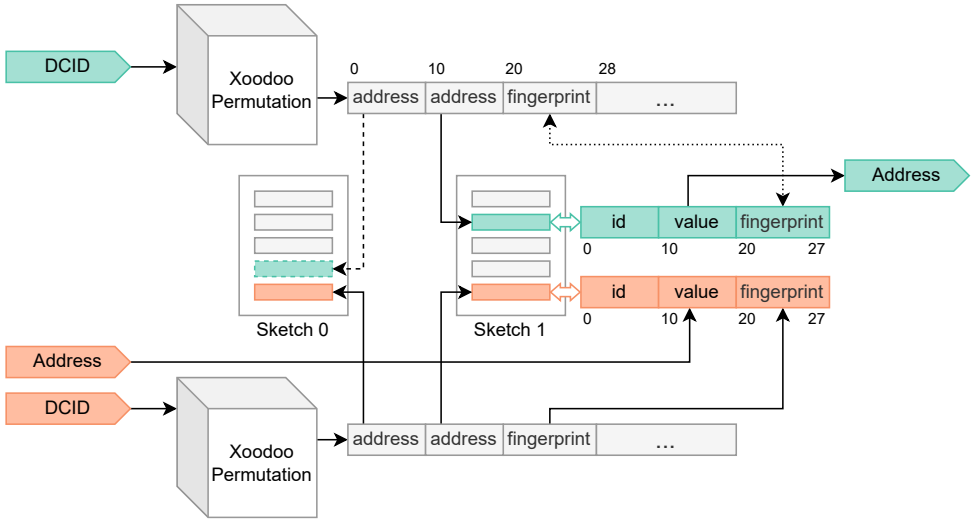
DCID CAM

Eens de filter een geldige DCID heeft, moet de DCID-CAM nagaan of er voor deze DCID sleutels beschikbaar zijn. Zoals paragraaf 3.3.1 uitlegt, kan dit m.b.v. een TCAM gebeuren. Omdat Xilinx geen CAM ter beschikking stelt voor de Zynq-7020 SoC, is er een interne, nog niet gepubliceerde oplossing van mr. Sateesan gebruikt.

Figuur 4.5 geeft een overzicht van de werking van deze oplossing. Een DCID-*lookup* gebeurt als volgt: de DCID uit de DCID-filter-module wordt door een Xoodoo permutatie, beschreven in [32], gevoerd. Het resultaat van deze permutatie doet dan dienst als adres voor een geheugen (aangeduid met de *sketches*, in Figuur 4.5), waarin het adres van de *key-memory*-module opgeslagen zit. Deze data wordt dan buiten de CAM aangeboden door middel van de *address output*.

Om een nieuwe DCID toe te voegen aan deze structuur, zijn er nog twee extra inputs beschikbaar: een extra DCID-input en een adres-input. Om dubbelzinnigheden te voorkomen, zal die laatste verder ook wel de geheugen-waarde genoemd worden. Door dezelfde

permutatie uit te voeren als de lees-logica kan de schrijf-logica de geheugen-waarde op het correct geheugen-adres schrijven. Merk op dat deze permutatie dezelfde is, maar niet door "dezelfde" hardware gebeurt. Dit komt omdat de software-QUIC-implementatie, en dus het controle-blok, onafhankelijk van de DCID-*lookup*-module DCID's moet kunnen toevoegen en verwijderen.



Figuur 4.5: Blokschema van de gebruikte CAM

Om de kans op *collisions* te verkleinen gebruikt de CAM twee geheugens. Het eerste geheugen is geadresseerd door de eerste 10 bits van het permutatieresultaat terwijl het 2^{de} geheugen bits 10 tot en met 19 gebruikt. Indien de permutatie nu één van de twee adressen doet botsen met het overeenkomstige adres van een andere DCID is het onmogelijk om te weten op welk adres de correcte waarde opgeslagen is. Daarom slaan de geheugens ook een *fingerprint* op. Deze komt overeen met bits 20 tot 28 van de permutatie output. De *fingerprint* kan nu vergeleken worden met die van beide adressen. Enkel indien de *fingerprint* overeenkomt, is de waarde op dat adres geldig.

Stel nu het schrijven van DCID_1 voor nadat DCID_0 reeds is toegevoegd aan de CAM. In dit geval is het mogelijk dat een adres of *fingerprint* botst. Combinaties van deze mogelijke botsingen kunnen ondergebracht worden in 5 scenario's die hieronder opgesomd zijn:

1. Geen van de adressen botst.
2. Eén van de adressen botst maar de *fingerprint* verschilt.
3. Eén van de adressen botst en de *fingerprint* botst ook.
4. Beide adressen botsen maar de *fingerprint* verschilt.
5. Beide adressen botsen en de *fingerprint* botst ook.

Indien er daarna een *lookup* van DCID_0 gebeurt, leveren deze scenario's elk een verschillend resultaat op. Zo zal het eerste scenario gewoon het correcte resultaat opleveren. In het tweede scenario wordt de juiste waarde gekozen m.b.v. de *fingerprint*. Als derde is de CAM niet in staat om de juiste waarde te kiezen, maar deze fout kan wel worden opgemerkt. Bij scenario 4 is de correcte *fingerprint* niet meer beschikbaar waardoor het lijkt dat DCID_0 nooit is toegevoegd. Tot slot, zal de CAM een foute waarde teruggeven in het laatste scenario.

Deze scenario's en bijhorende resultaten zijn samengevat in Tabel 4.1. Een overschreven geheugen-waarde door een adres-botsing, is aangeduid met "o" terwijl een correcte een "c" krijgt. Wanneer de *fingerprint* in het geheugen overeenkomt met de *calculated fingerprint* tijdens de *lookup*, is dat aangeduid met F_c.

Tabel 4.1: Verschillende scenario's voor *collisions* in de CAM.

Scenario	S_1	S_2	S_3	S_4	S_5
Value$_0$	c	c	c	o	o
Value$_1$	c	o	o	o	o
Fingerprint$_0$	F_c	F_c	F_c	$!F_c$	F_c
Fingerprint$_1$	F_c	$!F_c$	F_c	$!F_c$	F_c
Result	ok	ok	error	not found	wrong

De scenario's uit Tabel 4.1 zijn dus combinaties van drie mogelijke botsingen: die van adres_0, adres_1 of van de *fingerprint*. Indien Formule 4.1 de kansen op deze botsingen voorstelt, zal Formule 4.2 de kansen op de scenario's bevatten.

$$\begin{cases} V_0 = P(Xoodoo(dcid_0)_{0..9} = Xoodoo(dcid_1)_{0..9}) = \dfrac{1}{2^{10}} \\ V_1 = P(Xoodoo(dcid_0)_{10..19} = Xoodoo(dcid_1)_{10..19}) = \dfrac{1}{2^{10}} \\ F = P(Xoodoo(dcid_0)_{20..27} = Xoodoo(dcid_1)_{20..27}) = \dfrac{1}{2^8} \end{cases} \tag{4.1}$$

$$\begin{cases} P(S_1) = 1 - (V_0 + V_1) \\ P(S_2) = (V_0 + V_1) \cdot (1 - F) \\ P(S_3) = (V_0 + V_1) \cdot F \\ P(S_4) = V_0 \cdot V_1 \cdot (1 - F) \\ P(S_5) = V_0 \cdot V_1 \cdot F \end{cases} \tag{4.2}$$

De volledige kans op een succesvolle *lookup* (S_1 of S_2) is berekend door middel van Formule 4.2 en is gelijk aan $99,999237\%$. De kans op een opgemerkte fout (S_3) is dan $7,63 \cdot 10^{-6}$ terwijl de kans op een onopgemerkte fout (S_4 of S_5) gelijk is aan $9,54 \cdot 10^{-7}$. Voor een grotere kans op succes kunnen het adres en de *fingerprint* breder gemaakt worden. Dit gaat wel ten koste van extra resources.

Protocol update

In deze proefopstelling gebruiken de crypto-modules de protocol-metadata om te bepalen of een pakket al dan niet gedecrypteerd moet worden. Daarom zal de DCID-*lookup*-module deze metadata naar `C_PROTO_UNKNOWN` zetten indien er geen DCID-match gevonden is.

4.3.2 Key memory

De *key-memory*-module gebruikt het adres van de DCID-*lookup*-module om de bijhorende sleutels aan te bieden aan de *header-protection*-module. Zoals paragraaf 2.1.5 uitlegt, moet deze module sleutels bijhouden voor beide *key phases*. Vermits de eenvoudige en concrete eisen voor deze module, is de data per connectie in de proefopstelling hetzelfde als in de voorgestelde accelerator. Deze data is samengevat in Tabel 3.2.

Er worden dus vijf verschillende sleutels van maximum 128 bits opgehaald uit een RAM geheugen. Dit geheugen is 128 bits breed zodat er per klok-cyclus een volledige sleutel uitgelezen kan worden. De sleutels van één connectie volgen elkaar op. Zo volstaat een eenvoudige optelling om het adres van de volgende sleutel te bepalen. Wanneer de vijf sleutels van een connectie uitgelezen zijn maakt de *key-memory*-module deze beschikbaar voor de *header-protection*-module.

Tot slot is de *write interface* bij het geheugen in deze module apart uitgevoerd van de *read interface* zodat software ten alle tijden nieuwe connecties kan toevoegen of sleutels kan updaten. Deze controle operaties gebeuren met behulp van de *QUIC driver* en *control*-module uit Figuur 4.2.

4.3.3 Header protection

Sleutels komen in de *Header Protection* (HP)-module gelijktijdig aan met de eerste byte van het QUIC-pakket. Dit is een resultaat van de manier waarop modules de (meta)data vertragen, beschreven in paragraaf 4.1, en het feit de DCID niet uitgelijnd is op vier bytes. Dit laatste is gedemonstreerd in paragraaf 4.3.1. De HP-module zal deze sleutels dan opslaan in een lokaal register totdat de arriverende bytes niet langer deel uitmaken van een 1-RTT-QUIC-pakket.

De eerste stap van HP is de *payload* bemonsteren [5]. Zoals paragraaf 2.1.5 bespreekt moet er voor dit bemonsteren steeds vanuit gegaan worden dat de *packet number lenght* vier bytes is. Verder zal de DCID een vaste offset en lengte hebben[1]. Dankzij deze constante lengtes zal ook het monster een vaste offset hebben, deze vereenvoudiging gebruikt de HP-module om het 128-bit lange monster te nemen.

Daarna berekent de HP-module het masker door het monster met de *header protection key* te encrypteren met het `AES_128_ECB`-algoritme. Hiervan is er een iteratieve implementatie ter beschikking gesteld door de ES&S onderzoeksgroep. Door dit masker te XOR'en met de beschermde velden van de *flags*-byte, wordt de *packet number length* leesbaar. Zo kan de HP-module dit gebruiken om de bytes van het *packet number* ook te XOR'en met het masker, wat deze laatste ook leesbaar maakt.

4.3.4 Payload protection

Nu de 1-RTT-QUIC-*header* ontcijferd is, kan de payload gedecrypteerd worden door middel van `AEAD_AES_128_GCM`. Ook hiervan heeft ES&S een implementatie ter beschikking gesteld. Deze versie is echter niet snel genoeg om aan lijnsnelheid te kunnen decrypteren waardoor er buffers nodig zouden zijn. Verder zorgt dit ervoor dat het systeem verzadigd is nadat er een maximale hoeveelheid data verwerkt is. De *throughput* zou daardoor dus dalen. Om dat te voorkomen, is er een *pipelined* versie gemaakt die wel één blok van 128-bits per vier klok-cycli kan decrypteren. Deze versie maakt gedeeltelijk gebruik van implementatie van ES&S.

De AES-GCM-module laadt eerst de *header* in als *associated data*. De *header* is *padded* met `0x0` om de lengte te laten uitkomen op een veelvoud van 16 bytes, de AES-*blocksize*. Daarna wordt de *payload* gedecrypteerd zoals Figuur 2.6 weergeeft. Tot slot vergelijkt deze module de bekomen AEAD-tag met die van het pakket. Indien de tags niet overeen komen wordt de *authentication failure counter* met één verhoogt.

[1] Dit is gemotiveerd in paragraaf 4.3.1

Hoofdstuk 5

Resultaten

5.1 Timing

5.1.1 Frequentie

Met behulp van Xilinx Vivado is voor verschillende klokfrequenties de *Worst Negative Slack* (WNS) bepaald. De WNS is het tijdsverschil tussen het voltooien van het kritisch pad en de volgende stijgende klokflank. Bij toenemende frequenties zal de WNS op een gegeven moment negatief worden. In dat geval is het kritisch pad te traag voor de vooropgestelde klokfrequentie. Tabel 5.1 geeft de WNS van de proefopstelling weer rond dit kantelpunt. Daaruit volgt dat de maximale klokfrequentie van de QUIC-accelerator uit de proefopstelling op een PYNQ™-Z2 *development board* gelijk is aan 83,333336 MHz. Dit is minder dan 125 MHz, de kloksnelheid in een 1000BASE-T netwerk [33], maar vermits de dataverwerking per vier bytes gebeurt zal er toch meer dan 1000 Mbit/s gehaald kunnen worden[1].

Tabel 5.1: *Worst negative slack* bij de maximale frequentie

Frequentie (MHz)	**WNS** (ns)
76,923080	0,136
83,333336	0,054
90,909088	-0,220

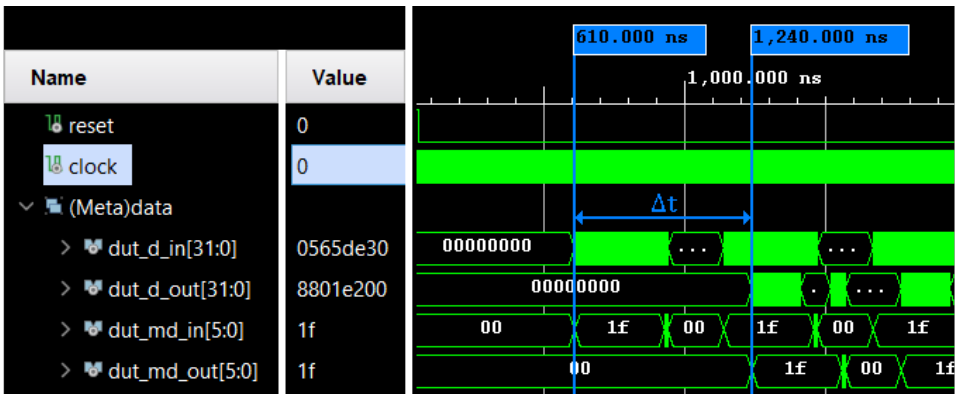
5.1.2 Latency

Zoals Figuur 5.1 illustreert is de *latency* van de proefopstelling in simulatie gelijk aan 630 ns. Vermits de klokperiode bij deze simulatie gelijk is aan 10 ns komt dat overeen met 63 klok-cycli. In combinatie met de maximale klokfrequentie uit paragraaf 5.1 kan berekend worden dat de totale ***latency* 756 ns** bedraagt op het *development board*. Formule 5.1 toont deze berekening.

$$\Delta t = \frac{63\ cc}{f_{max}} = \frac{63\ cc}{83,333336 \cdot 10^6\ Hz} = 756 \cdot 10^{-9}\ s = 756\ ns \tag{5.1}$$

De vertragingen van de individuele modules zijn op een gelijkaardige manier bepaald. Deze vertragingen zijn weergegeven in Tabel 5.2.

[1] Zie paragraaf 5.1.3



Figuur 5.1: Waveform van de simulatie met de *latency* aangeduid

Tabel 5.2: *Latency* van de verschillende modules

Module	**Klok-cycli**	***Latency*** (ns)
DCID lookup	11	132
Key memory	6	72
Header protection	14	168
Payload protection	32	384
Totaal	63	756

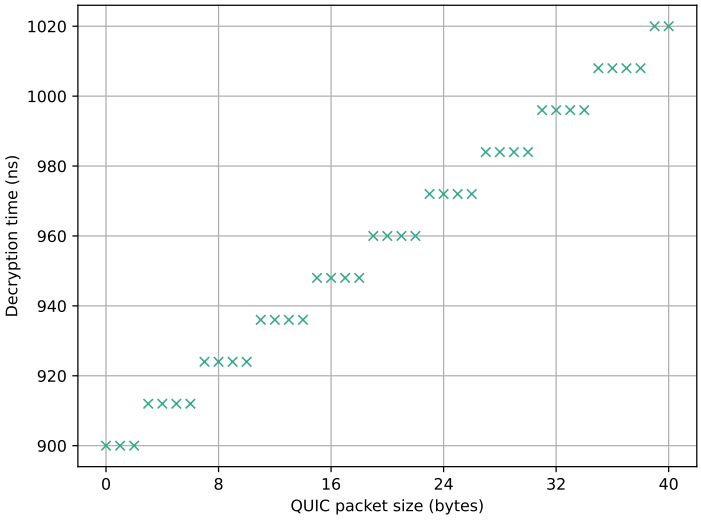
5.1.3 Throughput

Zoals paragraaf 4.3.4 bespreekt, is de proefopstelling ontworpen zonder buffers. Daardoor gebeurt de decryptie aan lijnsnelheid en is de *througput* enkel afhankelijk van de kloksnelheid. Verder vertelt paragraaf 4.1 dat data per 32 bits in parallel door de proefopstelling stroomt. Formule 5.2 combineert deze gegevens om de *throughput* te berekenen. Daaruit volgt dat de **maximale *throughput*** gelijk is aan **2,67 Gbit/s**.

$$throughput_{max} = f_{max} \cdot 32 = 83,33 \cdot 10^6 Hz \cdot 32bits = 2,67 \cdot 10^9 \ \frac{bits}{s} \tag{5.2}$$

5.1.4 Decryptietijd

De tijd die nodig is om een volledig QUIC-pakket te decrypteren is vanzelfsprekend afhankelijk van de lengte van het pakket. Vermits de proefopstelling 4 bytes per klokcyclus verwerkt en niet kan verzadigen zal de decryptietijd zich tot de pakketlengte verhouden zoals Figuur 5.2 weergeeft. Formule 5.7 geeft een wiskundige formulering voor deze verhouding. Merk op dat dit een louter theoretische berekening is. In werkelijkheid zullen pakketten steeds groter of gelijk zijn aan 3 bytes aangezien een QUIC-pakket altijd één *flags*-, minstens één *packet-number*- en minstens één *payload*-byte heeft [3].



Figuur 5.2: Decryptietijd van QUIC-pakketten met verschillende lengtes

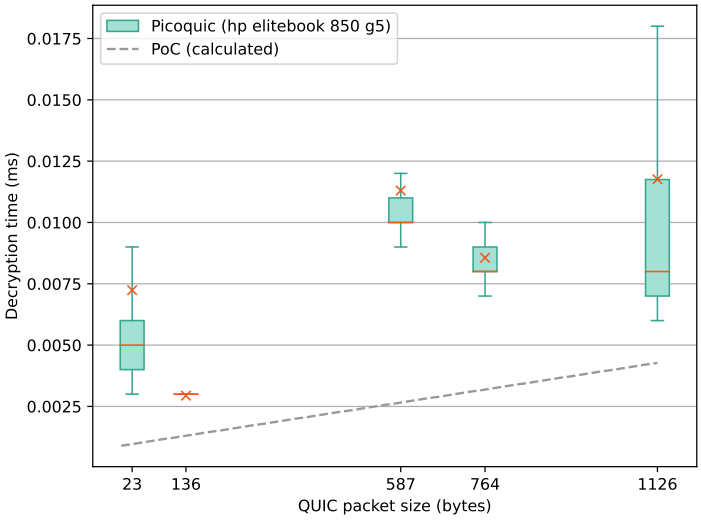
$$s_{Eh} = headersize(ETH) = 18\ bytes \tag{5.3}$$
$$s_{Ih} = headersize(IP) = 20\ bytes \tag{5.4}$$
$$s_{Uh} = headersize(UDP) = 8\ bytes \tag{5.5}$$
$$s_{Qt} = totalsize(QUIC) \tag{5.6}$$
$$\Downarrow$$
$$t_{decrypt} = \Delta t + \left\lceil \frac{s_{Eh} + s_{Ih} + s_{Uh} + s_{Qt}}{4} \right\rceil \cdot \frac{1}{f_{max}} \tag{5.7}$$
$$= 756\ ns + \left\lceil \frac{46 + s_Q}{4} \right\rceil \cdot 12\ ns \tag{5.8}$$

De proefopstelling is verder vergeleken met een software-implementatie genaamd picoquic [34]. Picoquic is een minimalistische QUIC-implementatie in C. Een volledige vergelijking van de proefopstelling met deze implementatie zou niet eerlijk zijn aangezien picoquic, wanneer het een 1-RTT-QUIC-pakket ontvangt, ook de frames interpreteert. Om het zo eerlijk mogelijk te maken is enkel de methode `picoquic_parse_header_and_decrypt` getimed. Deze interne methode neemt een UDP-*payload* als input en geeft, na decryptie, het *plaintext*-pakket terug. Verder is de *overhead* om sleutels op te slaan of te updaten niet in rekening gebracht bij deze test. De volledige broncode van deze test is beschikbaar in Bijlage C.

Figuur 5.3 geeft de tijd weer, die deze methode nodig heeft, voor decryptie in functie van de grootte van het QUIC-pakket. Aangezien picoquic uitgevoerd wordt in *user space* van een *operating system* levert niet elke test dezelfde tijd op. Daarom zijn deze picoquic resultaten weergegeven als boxplot van 50 afzonderlijke testen. Verder is ook $t_{decrypt}$ uit Formule 5.7 in het grijs weergegeven.

Volgens deze resultaten heeft de proefopstelling steeds minder tijd nodig dan picoquic om 1-RTT-QUIC-pakketten te decrypteren. Hierbij zijn wel twee belangrijke kanttekeningen te maken:

1. De decryptie m.b.v. picoquic is op een *consumer laptop* (HP Elitebook 850 G5) uitgevoerd en zal vermoedelijk sneller zijn op *server-grade*-hardware.

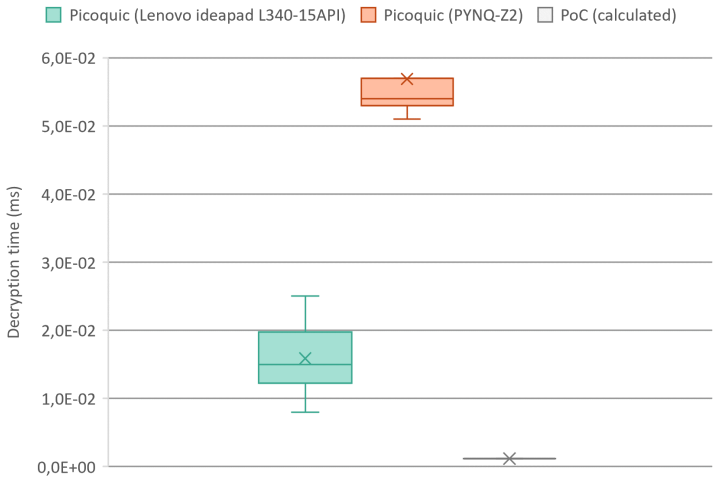


Figuur 5.3: Vergelijking tussen picoquic en de proefopstelling

2. De maximale frequentie van de proefopstelling is bepaald voor een relatief beperkte ZYNQ-7000 SoC. Bij een beter presterende SoC zal f_{max} vermoedelijk dus hoger zijn.

Omdat de spreiding van de picoquic resultaten relatief groot is, is er tot slot een test gedaan tussen een picoquic op een andere *consumer laptop* (Lenovo ideapad L340-15API) en op de PS van het PYNQ-bord. De pakketlengte was bij deze test steeds 88 bytes. Het resultaat, weergegeven door Figuur 5.4, toont dat de *AMD Ryzen™ 5 3500U* in de laptop steeds sneller was dan de *ARM dual-core Cortex-A9* in het PYNQ-bord. Hier zijn vermoedelijk twee redeneren voor: ten eerste heeft de *Ryzen™ 5* met 2,1GHz een hogere kloksnelheid dan *Cortex-A9* met 650MHz. Ten tweede kan picoquic met de *Ryzen™ 5* gebruik maken van de cryptografische *fusion AES-GCM engine* van picotls terwijl dit niet mogelijk is op de *Cortex-A9*. Voor de volledigheid geeft Figuur 5.4 ook de tijd weer die het de proefopstelling zou kosten om een 88-byte-lang pakket te behandelen.

Daaruit blijkt dus dat de performantie van picoquic zeer afhankelijk is van de hardware, daarom kunnen de resultaten uit Figuur 5.3 moeilijk veralgemeend worden.



Figuur 5.4: Picoquic decryptie tijd van een 88-byte-lang pakket op twee verschillende systemen

5.2 Resource gebruik

De *resource usage* van de proefopstelling is gevonden na *synthesis* m.b.v. Xilinx Vivado. Tabel 5.3 illustreert dit resultaat. De *payload-protection*-module is zonder twijfel de grootste module en gebruikt het meeste *Lookup Tables* (LUT's) en registers. Verder gebruikt de *key-memory*-module veel LUT's als geheugen vermits het *key*-geheugen in de proefopstelling bestaat uit *distributed RAM*. Dit kan verminderd worden door *block RAM* te gebruiken.

Tabel 5.3: *Resource* gebruik van de proefopstelling

Module	**Slice LUTs**	**Slice Registers**
DCID lookup	1147	427
Key memory	3375	871
Header protection	1548	1622
Payload protection	17146	4959
Totaal	23315	7995

Het totaal aantal *slice LUTs* dat de proefopstelling gebruikt komt overeen met $43,83\%$ van wat de Zynq-7020 SoC ter beschikking heeft. Bij de *slice registers* is deze verhouding $7,51\%$. Wat betreft het totaal aantal *slices* wordt $50,04\%$ van alle beschikbare gebruikt. De hoeveelheid gebruikte *resources* zal verder toenemen indien de proefopstelling uitgebreid wordt voor variabele DCID-lengtes. Een volledig systeem voor beide richtingen (ontvangen en zenden) zal dus niet op deze SoC passen en zal een grotere FPGA vereisen.

Hoofdstuk 6

Besluit

De proefopstelling toont aan dat het mogelijk is om QUIC-pakketten te decrypteren in hardware. Het kan 1-RTT-QUIC-pakketten decrypteren aan een lijnsnelheid van 2,67 Gbit/s, wat sneller is dan picoquic. De grootste en traagste module uit de proefopstelling is de AEAD-module. Een *hardware offload* van decryptie zorgt er dus voor dat de decryptie sneller gebeurt en dat de processor vrij is voor andere taken.

Echter, zoals voorgaande hoofdstukken bespreken is de proefopstelling in geen geval inzetbaar in *real-world*-toepassingen. Zo is er bijvoorbeeld geen integratie met een, al dan niet bestaande, QUIC-implementatie. Verder kunnen er enkel inkomende 1-RTT-QUIC-pakketten gedecrypteerd worden die geëncrypteerd zijn met `TLS_AES_128_GCM_SHA256`. Tot slot biedt de proefopstelling geen ondersteuning voor DCID's met een lengte verschillend van 20 bytes.

Deze resultaten zijn gegenereerd met behulp van de hardware die beschikbaar was en niet met *server grade hardware*. Verder onderzoek is dus nodig om met zekerheid te kunnen zeggen of een *hardware offload* op deze manier ook een significante tijdswinst oplevert voor *server grade hardware*. Daarnaast kan vervolgonderzoek zich focussen op encryptie van uitgaande pakketten of op variabele DCID-lengtes. Tot slot, moet ook de integratie met een bestaande QUIC-implementatie gemaakt worden. Enkel zo kan de controle-overhead, die het doorgeven van de DCID's en sleutels aan de hardware introduceert, gekwantificeerd worden.

Literatuurlijst

[1] M. Bishop, "Hypertext Transfer Protocol (HTTP) over QUIC," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-http-00, Nov. 2016, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-00

[2] ——, "Hypertext Transfer Protocol Version 3 (HTTP/3)," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-http-34, Feb. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34

[3] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://rfc-editor.org/rfc/rfc9000.txt

[4] D. Ferreira, "Http2 multiplexing: The devil is in the details," Aug. 2019. [Online]. Available: https://blog.codavel.com/http2-multiplexing

[5] M. Thomson and S. Turner, "Using tls to secure quic," RFC 9001, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9001

[6] E. Rescorla, "The transport layer security (tls) protocol version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8446

[7] E. Gagliardi and O. Levillain, "Analysis of quic session establishment and its implementations," in *IFIP International Conference on Information Security Theory and Practice*. Springer, 2019, pp. 169–184.

[8] D. McGrew, "An interface and algorithms for authenticated encryption," RFC 5116, Jan. 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5116

[9] C. Sanders and J. Smith, "Packet analysis," in *Applied Network Security Monitoring*. Boston: Syngress, 2014, ch. 13, pp. 341–384.

[10] D. Eastlake 3rd and J. Abley, "Iana considerations and ietf protocol and documentation usage for ieee 802 parameters," RFC 7042, Oct. 2013. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7042.html

[11] IEEE, "Ieee standards for local and metropolitan area networks: Virtual bridged local area networks," *IEEE Std 802.1Q-1998*, pp. 1–214, 1999.

[12] Huawei Technologies Co., Ltd., "What is a vlan," Huawei Technologies, Tech. Rep., May 2019. [Online]. Available: https://support.huawei.com/enterprise/en/doc/EDOC1100086556/1d08ffe0

[13] J. Postel, "Internet protocol," RFC 791, Sep. 1981. [Online]. Available: https://www.rfc-editor.org/info/rfc791

[14] S. Deering and R. Hinden, "Internet protocol, version 6 (ipv6) specification," RFC 2460, Dec. 1998. [Online]. Available: https://www.rfc-editor.org/info/rfc2460

[15] IANA, "Assigned internet protocol numbers," Mar. 2021. [Online]. Available: https://www.iana.org/assignments/protocol-numbers/

[16] Cisco Systems, "Ipv6 extension headers review and considerations," Oct. 2006.

[17] J. Postel, "User datagram protocol," RFC 768, Aug. 1980. [Online]. Available: https://www.rfc-editor.org/rfc/rfc768

[18] W. Goralski, "User datagram protocol," in *The Illustrated Network*, 2nd ed., W. Goralski, Ed. Boston: Morgan Kaufmann, 2017, ch. 11, pp. 289–306.

[19] The kernel development community, "Segmentation offloads." [Online]. Available: https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html

[20] B. Pismenny, H. Eran, A. Yehezkel, L. Liss, A. Morrison, and D. Tsafrir, "Autonomous nic offloads," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 18–35.

[21] J. Hay, M. Machnikowski, G. Bowers, N. Wochtman, J. Muniak, and M. Deval, "Accelerating quic via hardware offloads through a socket interface," in *The Technical Conference on Linux Networking (Netdev)*, 2019.

[22] X. Yang, L. Eggert, J. Ott, S. Uhlig, Z. Sun, and G. Antichi, "Making quic quicker with nic offload," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 21–27.

[23] M. Deval and G. Bowers, "Technologies for accelerated quic packet processing with hardware offloads," European Patent 3 541 044, Sep. 1, 2021.

[24] S. Senapathi and R. Hernandez, "Tcp offload engines," *Network and Communications magazine*, pp. 103–107, 2004.

[25] A. Viviano, "Introduction to receive side scaling," Mar. 2022. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling

[26] Xilinx, Inc., "Semi-ternary cam search," May 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/PG319-stcam

[27] D. Medhi and K. Ramasamy, "Ip packet filtering and classification," in *Network Routing (Second Edition)*, second edition ed., ser. The Morgan Kaufmann Series in Networking, D. Medhi and K. Ramasamy, Eds. Boston: Morgan Kaufmann, 2018, ch. 15, pp. 500–547.

[28] A. Rasmussen, A. Kragelund, M. Berger, H. Wessing, and S. Ruepp, "Tcam-based high speed longest prefix matching with fast incremental table updates," in *2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*, 2013, pp. 43–48.

[29] Technology Unlimited, "Pynq-z2 reference manual v1.0," May 2018.

[30] Xilinx, Inc., "Zynq-7000 soc." [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[31] P. Biondi and the Scapy community, "Scapy - packet crafting for python2 and python3," 2022. [Online]. Available: https://scapy.net/

[32] J. Daemen, S. Hoffert, G. V. Assche, and R. V. Keer, "Xoodoo cookbook," *IACR Cryptology ePrint Archive*, vol. 2018, p. 767, 2018.

[33] G. Thompson, "How 1000base-t works," *Presentation at IEEE*, vol. 802, 1997.

[34] C. Huitema, B. Köcher, A. L. Goutte, and I. Lubashev, "picoquic," 2022. [Online]. Available: https://github.com/private-octopus/picoquic/

Bijlagenlijst

Bijlage A

Testdata definitie in JSON formaat

Testdata kan uit meerdere 1-RTT-QUIC-pakketten bestaan met daartussen een *gap* van variabele lengte. Code A.1 illustreert de JSON-definitie van een pakket terwijl Code A.2 die van een *gap* weergeeft.

```
{
    "structure" : ["ethernet", "ip", "quic1rtt"],
    "decrypt" : true,
    "eth-dest-mac" : "de:fa:ce:db:ab:e1",
    "eth-src-mac" : "38:de:ad:64:16:62",
    "eth-proto" : "0x0800",
    "ip-ver" : "0x4",
    "ip-ihl" : "0x5",
    "ip-src-addr" : "10.25.135.108",
    "ip-dest-addr" : "172.64.146.82",
    "udp-src-port" : "63968",
    "udp-dest-port" : "12345",
    "quic-spin-bit" : "0b1",
    "quic-reserved" : "0b0",
    "quic-key-phase" : "0b0",
    "quic-packet-nr-len" : "2",
    "quic-dcid" : "f3dec1bb98d92f7402e898cec879542cbd28234c",
    "quic-packet-nr" : "0xc6b0c9",
    "quic-hp-key" : "acfa8bcf0b477016ce23e75d5db6182f",
    "quic-pp-key" : "e80afdae24beb1764018f18e094a93ed",
    "quic-iv" : "d448041afa37c0322fe18eee",
    "quic-plaintext-payload" : "bbbbbbbbbbbbbbbbbbbbbbbb"
}
```

Code A.1: Pakket-definitie in JSON formaat

```
{
    "structure" : "gap",
    "gap-size" : 20
}
```

Code A.2: *Gap*-definitie in JSON formaat

Bijlage B

Scapy definitie van een 1-RTT-QUIC-pakket

Om data te genereren is gebruik gemaakt van Scapy. Code B.1 geeft de definitie van een 1-RTT-QUIC-pakket voor het Scapy-*framework*. Verder illustreren Code B.2 en Code B.3 methodes om deze pakketten respectievelijk te encrypteren en decrypteren.

```
class QUIC_1RTT(Packet):
    name = "1-RTT QUIC"
    fields_desc = [
        # Flags
        BitEnumField("hdr_form", 0, 1, {0: "short", 1: "long"}),
        BitEnumField("fixed_bit", 1, 1, {0: "error", 1: "ok"}),
        BitEnumField("spin_bit", 1, 1, {0: "0", 1: "1"}),
        BitField("reserved", 0, 2),
        BitEnumField("key_phase", 0, 1, {0: "0", 1: "1"}),
        BitFieldLenField("pnr_len", None, 2, length_of="packet_nr"),
        # Destination Connection ID
        StrField("dcid", None),
        # Packet number
        StrLenField("packet_nr", 0, lambda pkt: pkt.pnr_len + 1)
    ]
```

Code B.1: Definitie van 1-RTT-QUIC-pakketten voor Scapy

```
    def protect_packet(self, pp_key, iv, hp_key):
        header = self.copy()
        header.load = raw(b'')
        payload = self[1]

        # Compute nonce
        nonce = int.from_bytes(iv, byteorder='big') ^ int.from_bytes(
            header.packet_nr, byteorder='big')
        nonce = nonce.to_bytes(12, byteorder='big')

        # Perform AEAD
        cipher = AES.new(pp_key, AES.MODE_GCM, nonce=nonce)
        cipher.update(raw(header))
        ciphertext, tag = cipher.encrypt_and_digest(raw(payload))
        protected_payload = ciphertext + tag

        # Extract the sample
        PNL = 1 + header.pnr_len
        sample_start = 4 - PNL  # offset from payload start where
            ↪ sample begins
        sample = ciphertext[sample_start:sample_start+16]

        # Create mask
        cipher = AES.new(hp_key, AES.MODE_ECB)
        mask = cipher.encrypt(sample)

        # Protect header
        protected_header = bytearray(raw(header))
        protected_header[0] ^= (mask[0] & 0x1f)
        for i in range(PNL):
            protected_header[-PNL + i] ^= mask[i+1]
        protected_header = bytes(protected_header)

        # modify packet with protection
        self.hdr_form = (protected_header[0] & (0x1 << 7)) >> 7
        self.fixed_bit = (protected_header[0] & (0x1 << 6)) >> 6
        self.spin_bit = (protected_header[0] & (0x1 << 5)) >> 5
        self.reserved = (protected_header[0] & (0x3 << 3)) >> 3
        self.key_phase = (protected_header[0] & (0x1 << 2)) >> 2
        self.pnr_len = (protected_header[0] & 0x3)
        self.packet_nr = protected_header[-PNL:]
        self.load = protected_payload
```

Code B.2: Methode om 1-RTT-QUIC-pakketten te encrypteren

```
    def decrypt_packet(self, pp_key, iv, hp_key):
        header = self.copy()
        header.load = raw(b'')
        payload = raw(self[1])

        # Extract the sample
        PNL = len(self.packet_nr)
        sample_start = 4 - PNL  # offset from payload start where
            ↪ sample begins
        sample = payload[sample_start:sample_start+16]

        # Create mask
        cipher = AES.new(hp_key, AES.MODE_ECB)
        mask = cipher.encrypt(sample)

        # Protect header
        decrypted_header = bytearray(raw(header))
        decrypted_header[0] ^= (mask[0] & 0x1f)
        for i in range(PNL):
            decrypted_header[-PNL + i] ^= mask[i+1]
        decrypted_header = bytes(decrypted_header)

        # Compute nonce
        nonce = int.from_bytes(iv, byteorder='big') ^ int.from_bytes(
            decrypted_header[-PNL:], byteorder='big')
        nonce = nonce.to_bytes(12, byteorder='big')

        # Perform AEAD
        cipher = AES.new(pp_key, AES.MODE_GCM, nonce=nonce)
        cipher.update(decrypted_header)
        plaintext = cipher.decrypt_and_verify(self.load[:-16], self.
            ↪ load[-16:])

        # modify packet with protection
        self.hdr_form = (decrypted_header[0] & (0x1 << 7)) >> 7
        self.fixed_bit = (decrypted_header[0] & (0x1 << 6)) >> 6
        self.spin_bit = (decrypted_header[0] & (0x1 << 5)) >> 5
        self.reserved = (decrypted_header[0] & (0x3 << 3)) >> 3
        self.key_phase = (decrypted_header[0] & (0x1 << 2)) >> 2
        self.pnr_len = (decrypted_header[0] & 0x3)
        self.packet_nr = decrypted_header[-PNL:]
        self.load = plaintext
```

Code B.3: Methode om 1-RTT-QUIC-pakketten te decrypteren

Bijlage C

Picoquic test code

```
// ---------------------------------------------------------------------------
// Master's thesis: Hardware/software co-design for the new QUIC
    ↪ network protocol
// ---------------------------------------------------------------------------
// Characterization tests
//
// File: picoquic_1rtt_dec.c (c)
// By: Lowie Deferme (UHasselt/KULeuven - FIIW)
// On: 04 July 2022
// ---------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>

#include "../../libs/picoquic/picoquic/picoquic.h"
#include "../../libs/picoquic/picoquic/picoquic_internal.h"
#include "../../libs/picoquic/picoquic/picoquic_utils.h"
#include "../../libs/picoquic/picoquic/tls_api.h"
#include "../../libs/picotls/include/picotls.h"
#include "../../libs/picotls/include/picotls/openssl.h"

#include "picoquic_1rtt_dec.h"

/**
 * "quic-spin-bit" : "0b1",
 * "quic-reserved" : "0b0",
 * "quic-key-phase" : "0b0",
 * "quic-packet-nr-len" : "2",
 * "quic-dcid" : "f3dec1bb98d92f7402e898cec879542cbd28234c",
 * "quic-packet-nr" : "0xc6b0c9",
 * "quic-hp-key" : "acfa8bcf0b477016ce23e75d5db6182f",
 * "quic-pp-key" : "e80afdae24beb1764018f18e094a93ed",
 * "quic-iv" : "d448041afa37c0322fe18eee",
 * "quic-plaintext-payload" : "
    ↪ bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb"
```

```
 */
const uint8_t msg[] = "\x6d\xf3\xde\xc1\xbb\x98\xd9\x2f\x74\x02\xe8\
    ↪ x98\xce\xc8\x79\x54\x2c\xbd\x28\x23\x4c\x76\xb8\x64\x5e\xe1\xee
    ↪ \xe1\xc3\x7d\xb9\xc7\x79\x11\x2a\xe8\xbb\x5a\xfa\xaa\x6c\xbb\
    ↪ x71\xd3\xf0\xcd\x29\xbe\x3a\xb0\x40\x14\x65\x86\x30\xe8\xc0\x2b
    ↪ \xfc\x75\xb5\xe3\x22\xc2\xa4\x59\x32\x17\x02\xc2\x98\x52\x07\
    ↪ x94\x3d\x2b\x4b\xef\xbf\x29\x78\x7f\x2d\xdd\x2b\x74\x9d";
size_t msg_len = sizeof(msg) - 1; // Do not count string terminator
const uint8_t dcid_id[] = "\xf3\xde\xc1\xbb\x98\xd9\x2f\x74\x02\xe8\
    ↪ x98\xce\xc8\x79\x54\x2c\xbd\x28\x23\x4c";
size_t dcid_len = sizeof(dcid_id) - 1;
const uint8_t hp[] = "\xac\xfa\x8b\xcf\x0b\x47\x70\x16\xce\x23\xe7\
    ↪ x5d\x5d\xb6\x18\x2f";
const uint8_t pp[] = "\xe8\x0a\xfd\xae\x24\xbe\xb1\x76\x40\x18\xf1\
    ↪ x8e\x09\x4a\x93\xed";
const uint8_t iv[] = "\xd4\x48\x04\x1a\xfa\x37\xc0\x32\x2f\xe1\x8e\
    ↪ xee";

int buffer_size = PICOQUIC_MAX_PACKET_SIZE;
uint8_t *bytes_in;
uint8_t *bytes_out;

int main(int argc, char **argv)
{
  printf("[INFO] Function \"%s\" in \"%s\" has started\n",
     ↪ __FUNCTION__, __FILE__);

  int exit_code = 0;

  bytes_in = malloc(buffer_size);
  memset(bytes_in, 0, buffer_size);

  memcpy(bytes_in, &msg, msg_len);

  printf("\n[INFO] Parse header and decrypt:\n");
  printf("------ ------------------------------------------------\n");
  printf("[DEBUG] Bytes in: ");
  print_byte_array(bytes_in, msg_len);
  printf("%d,\n", msg_len);
  for (int test = 0; test < 50; test++)
  {
    picoquic_stream_data_node_t *decrypted_data = NULL;

    /*****************************************************
     *                Create picoquic context            *
     *****************************************************/

    picoquic_quic_t *quic = picoquic_create(
        8,
        NULL,
        NULL,
        NULL,
        NULL,
```

```
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    picoquic_current_time(),
    NULL,
    NULL,
    NULL,
    0
);

if (quic != NULL)
{
  quic->local_cnxid_length = 20;

  /*****************************************************
   *             Create picoquic connection            *
   *****************************************************/

  struct sockaddr_in sa;
  memset(&sa, 0, sizeof sa);
  sa.sin_family = AF_INET;
  inet_pton(AF_INET, "10.25.135.108", &(sa.sin_addr));
  sa.sin_port = 4434;

  picoquic_cnx_t *connection = picoquic_create_cnx(
      quic,
      picoquic_null_connection_id,
      picoquic_null_connection_id,
      (struct sockaddr *)&sa,
      0,
      0,
      NULL,
      NULL,
      0
  );

  picoquic_connection_id_t dcid = *((picoquic_connection_id_t *)(
    ↪ malloc(sizeof(picoquic_connection_id_t))));
  dcid.id_len = dcid_len;
  for (int i = 0; i < dcid.id_len; i++)
    dcid.id[i] = dcid_id[i];
  connection->path[0]->p_local_cnxid =
    ↪ picoquic_create_local_cnxid(connection, &dcid, 0);

  ptls_cipher_suite_t *cipher = picoquic_get_aes128gcm_sha256_v
    ↪ (1);
  ptls_cipher_algorithm_t *hp_algo = cipher->aead->ctr_cipher;
  ptls_aead_algorithm_t *pp_algo = cipher->aead;

  const void *hp_h = malloc(hp_algo->key_size);
  const void *pp_h = malloc(pp_algo->key_size);
```

```
const void *iv_h = malloc(pp_algo->iv_size);
for (int i = 0; i < hp_algo->key_size; i++)
  ((uint8_t *)hp_h)[i] = hp[i];
for (int i = 0; i < pp_algo->key_size; i++)
  ((uint8_t *)pp_h)[i] = pp[i];
for (int i = 0; i < pp_algo->iv_size; i++)
  ((uint8_t *)iv_h)[i] = iv[i];

ptls_cipher_context_t *pn_context = ptls_cipher_new(hp_algo, 1,
   ↪ hp_h);
ptls_aead_context_t *aead_context = ptls_aead_new_direct(
   ↪ pp_algo, 0, pp_h, iv_h);

connection->crypto_context[3].pn_dec = pn_context;
connection->crypto_context[3].aead_decrypt = aead_context;

/*****************************************************
 *      Parse & decrypt bytes_in into bytes_out      *
 *****************************************************/

decrypted_data = picoquic_stream_data_node_alloc(quic);
if (decrypted_data != NULL)
{
  int length = msg_len;
  int packet_length = length;

  picoquic_packet_header ph;
  picoquic_cnx_t *cnx = NULL;
  int new_context_created = 0;
  size_t consumed = 0;

  uint64_t current_time = picoquic_current_time();

  clock_t t_s = clock();
  int result = picoquic_parse_header_and_decrypt(
      quic,
      bytes_in,
      length,
      packet_length,
      NULL,
      current_time,
      decrypted_data,
      &ph,
      &cnx,
      &consumed,
      &new_context_created
  );
  bytes_out = decrypted_data->data;
  clock_t t_e = clock();
  double dt = (double)(t_e - t_s) * 1000.0 / CLOCKS_PER_SEC;
  printf("%f,\n", dt);
}
else
```

```
      {
        printf("[ERROR] picoquic_stream_data_node_alloc for
           ↪ decrypted_data failed\n");
      }
    }
    else
    {
      printf("[ERROR] picoquic_create failed\n");
    }

    if (decrypted_data != NULL)
    {
      picoquic_stream_data_node_recycle(decrypted_data);
    }
    if (quic != NULL)
    {
      picoquic_free(quic);
    }
  }
  printf("[DEBUG] Bytes out: ");
  print_byte_array(bytes_out, msg_len);

  free(bytes_in);

  printf("[INFO] Function \"%s\" in \"%s\" is finished\n",
     ↪ __FUNCTION__, __FILE__);
  exit(exit_code);
}

/**
 * @brief Print a array's value and end with newline
 *
 * @param array Buffer to print
 * @param array_len Length of array to print
 * @return int Zero if success
 */
int print_byte_array(uint8_t *array, int array_len)
{
  for (int i = 0; i < array_len; i++)
  {
    printf("%02x", array[i]);
  }
  printf("\n");
  return 0;
}
```

Code C.1: Brondcode van de picoquic test