Gezamenlijke opleiding UHasselt en KU Leuven

2021•2022

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Proof of concept of a hardware/Software co-design for accelerating decryption of 1-RTT QUIC packets

PROMOTOR :

dr. Robin MARX

COPROMOTOR :

dr. ing. Jo VLIEGEN

Lennert Purnal

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

2021•2022
Faculteit Industriële Ingenieurswetenschappen
master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Proof of concept of a hardware/Software co-design for accelerating decryption of 1-RTT QUIC packets

PROMOTOR :
dr. Robin MARX

COPROMOTOR :
dr. ing. Jo VLIEGEN

Lennert Purnal

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

UHASSELT
KU LEUVEN

Preface

This master's thesis is conducted at "Emerging technologies, systems and security" (ES&S) in Diepenbeek and is submitted for the degree of Master of industrial engineering technology in electronics-ICT at the KULeuven and UHasselt. ES&S is a research group of the KULeuven specialised in the design and implementation of embedded systems with a focus on secure hardware and software. I would like to thank my promotors Dr. Robin Marx and Dr. Ing. Jo Vliegen for guiding me in this thesis and providing me with the opportunity to work on an interesting topic at the cutting edge of technology. Specifically, I would like to thank Dr. Marx for providing his exceptional expertise of the QUIC protocol and Dr. Vliegen for his great amount of help in the development of the digital hardware design. Furthermore, I would like to thank my co-students Lowie Deferme for creating a test framework and the DCID lookup module, and Jonathan Butaye for creating the key memory and AXI-controller with its driver. Finally, I would like to thank everybody who helped for creating a nice and productive work environment.

Lennert Purnal

Contents

List of Tables

List of Figures

List of abbreviations and symbols

ACK	Acknowledgement
AEAD	Authenticated encryption with associated data
AES	Advanced encryption standard
ARM	Acorn RISC Machine (computer architecture)
armhf	Hard float ARM computer architecture
CPU	Central processing unit
DCID	Destination connection ID
DMA	Direct memory access
FIFO	First in, first out
GCM	Galois counter mode
HKDF	HMAC-based key derivation function
HTTPS	Hypertext transfer protocol with TLS
ID	Identifier
IETF	Internet engineering taskforce
IO	Input/output
IP	Internet protocol
LAN	Local Area Network
LSB	Least significant bit
LUT	Lookup table
MSB	Most significant bit
NAT	Network address translation
NIC	Network interface card
OS	Operating system
PL	Programmable logic
PoC	Proof of concept
PS	Processing system
RFC	Request for comments
RISC	Reduced instruction set computer
RTT	Round-trip time
TCP	Transport control protocol
TLS	Transport layer security
UDP	User datagram protocol
XOR	Exclusive OR

Abstract

QUIC is a relatively new transport layer protocol that aims to improve upon TCP with TLS. Most implementations are currently executed in user space to allow for flexibility. This makes the protocol processor intensive and currently slower than TCP because of user/kernel space data copy. This master's thesis aims to reduce this CPU cost and to accelerate the decryption of 1-RTT QUIC packets through the use of dedicated hardware.

First, opportunities for offloading QUIC are examined. A hardware/software co-design is made for receiving packets. A proof of concept (PoC) is implemented in hardware and verified on an FPGA. The resulting design performs packet parsing to detect QUIC packets, followed by a connection ID lookup and key lookup to retrieve the connection secrets. The header protection stage uses the AES_128_ECB cipher. The payload protection uses the AEAD_AES_128_GCM cryptographic suite. The decryption speed and CPU resource usage was measured for both software decryption using the picoquic stack and hardware decryption using the proof of concept design on a FPGA. Note, however, that these tests do not include overhead such as data copy.

Decrypting packets in hardware is 2 times than in software on consumer grade hardware. The maximum raw network throughput of the hardware is 2.7 Gbit/s. In future work, the PoC design should be expanded with functionality and fully integrated with a QUIC software stack to perform more thorough measurements such as maximum QUIC data throughput, network latency and other parameters.

Abstract in Dutch

QUIC is een relatief nieuw protocol in de transportlaag met als doel verbeteringen aan te brengen ten opzichte van TCP met TLS. De meeste implementaties zijn in *userspace* gemaakt. Dit maakt het protocol processorintensief en, op dit moment, trager dan TCP vanwege het kopiëren van data tussen *user- en kernelspace*. Deze masterproef analyseert mogelijke verbeteringen hiervoor door middel van toegewijde hardware voor de ontcijfering van 1-RTT QUIC pakketten.

Een hardware/software co-design werd ontworpen voor het ontvangen van 1-RTT pakketten zonder extra datakopie. Een *proof of concept* werd praktisch geïmplementeerd op een FPGA. Het ontwerp detecteert eerst QUIC pakketten door middel van *pakket-parsing*, gevolgd door een *connection ID lookup*. Die opzoeking geeft een adres waarmee de sessiesleutels opgehaald worden. Daarna wordt de header ontcijferd aan de hand van het *AES_128_ECB cipher* en vervolgens wordt de payload ontcijferd en geauthenticeerd met behulp van de *AEAD_AES_128_GCM cipher* suite. De performantie van dit ontwerp werd vergeleken met software ontcijfering in de *picoquic stack*.

Het ontcijferen van pakketten in hardware is 2 keer sneller dan in software met *consumer grade hardware*. De maximale netwerk *throughput* van de hardware is 2,7 Gbit/s. Die throughput houdt echter nog geen rekening met andere factoren zoals de datakopie. In toekomstig werk zou het proof of concept uitgebreid moeten worden en volledig geïntegreerd worden in een QUIC stack om een beter beeld van de performantiewinst te krijgen.

Chapter 1

Introduction

With the ever-increasing traffic on the Internet and the emergence of mobile users, improvements to the existing infrastructure and protocols are in high demand. 'QUIC' is one such protocol that aims to improve upon TCP with TLS and add features to accommodate for the modern needs of Internet users. This relatively new protocol has recently been formulated in a series of IETF standards[8, 9]. This now enables researching optimizations to QUIC without the optimaizations being invalidated by large updates to the protocol.

Currently, most implementations of the QUIC protocol are implemented in user space. This way, the protocol´and its implementations can more easily be updated. TCP with TLS usually has parts of its protocol implemented in kernel space, which means the operating systems (OS) on which the protocols run have to be updated when changes are made. However, updating an OS can be a cumbersome task (certainly on older machines) which leads to changes happening slowly because not all machines on the Internet are updated at the same time. Another problem that QUIC aims to solve is *protocol ossification*, which is caused by middleboxes[1] that are also often hard to replace or update[10, 11, 1]. TCP with TLS suffers from this problem as the TCP header is transmitted in plaintext and middleboxes can freely inspect it. QUIC solves this by encrypting as much of its header as possible in a seperate encryption stage to the payload encryption. This removes the possibility for middleboxes to retrieve enough information from a packet to interfere with it.

The implementation in user space in combination with the encryption by default currently makes QUIC slower and more processor intensive than TCP with TLS. Copying the data from kernel space to user space and vice versa is an expensive task for the processor, creating an important bottleneck. Another bottleneck is caused by the expensive encryption that is executed in user space. As it is favorable to keep the protocol in user space for flexibility, the second problem (encryption) will be the focus of this research. However, both problems are interlinked, as creating or using a hardware encryption/decryption core, that is used from software, will require the data to be copied from user space to kernel space and back again. This adds even more data copy overhead and possibly removes the benefit of faster decryption (or even makes it worse). This leads to the following research topics: Can the cryptography of QUIC be offloaded to hardware? Which parts of the cryptography are best suited for this? What is the benefit and what are the

[1] middleboxes are devices on a network placed between the endpoints that inspect/filter/transform passing packets such as firewalls, NATs, load balancers, etc.

drawbacks of offloading cryptography in a specific manner?

The objective is to create a hardware/software co-design that handles packet protection. The architecture should minimalize kernel/user space data copy. This has to be balanced with hardware complexity and resource usage. Parts of the design should then be implemented as a proof of concept (PoC) for decrypting 1 roundtrip time (1-RTT) packets, which make up the main bulk of QUIC traffic, using the TLS_AES_128_GCM_SHA256 cipher suite.

Firstly, in chapter 2 the theoretical background of the QUIC protocol, as well as the encryption suites that are used, are explained. Next, in chapter 3 an ideal design is constructed and simplifications are made for a proof of concept design. Furthermore, the practical implementation and its test setup is documented. In chapter 4 the design is characterised and a perfomance comparison with software-only decryption is made. The limitations of the design are also discussed in chapter 4. Finally, in chapter 5 this research is concluded and ideas for future work are proposed.

Chapter 2

Literature study

This chapter aims to provide a general understanding of the principal components of the QUIC protocol, as well as giving a deeper insight into those aspects that will be used for this thesis. It is important to mention that this thesis will focus on the IETF's standardized version of the protocol in RFC9000[8]. Furthermore, previous works concerning the acceleration of QUIC implementations using hardware are examined. Studies on the performance of the components of the protocol are used to determine where optimisations in hardware could be most beneficial. Details on the underlying protocols and cipher suites for QUIC in the context of this research are also explained.

2.1 Overview of the QUIC protocol

QUIC is a transport layer protocol aimed at improving HTTPS traffic. It is a reliable transport protocol built on top of the unreliable UDP transport protocol. This is done to avoid protocol ossification[1] by middleboxes[10, 11, 1] which is a problem the transport control protocol[12] (TCP) and its derivatives (like TCP Fast Open, multipath TCP, etc.) suffer from. QUIC takes most features from TCP, but improves upon some those, as well as adding new features that are important in the modern state of the Internet. A comparison between the HTTP/2 protocol stack and that of HTTP/3 is shown in Figure 2.1.

The remainder of this section gives an in-depth explanation of the most important QUIC protocol components in the context of this thesis.

2.1.1 Connections

A QUIC connection is shared between a client and a server. Each connection starts with a handshake in which a shared secret and the application protocol are negotiated, as well as some parameters. This confirms both parties are willing and able to communicate[8].

[1] Protocol ossification is the reduction of flexibility of protocol design due to middleboxes as these are difficult to remove or upgrade in the network.

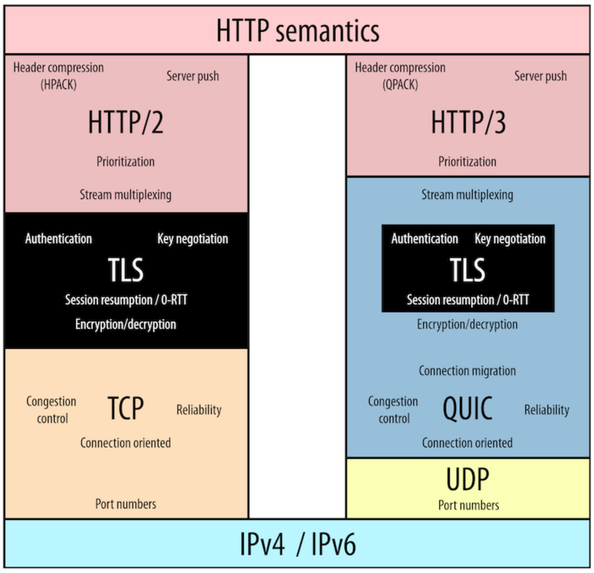


Figure 2.1: Typical HTTPS protocol stack comparison between TCP (with HTTP/2) and QUIC (with HTTP/3). From: [1]

Connection IDs

A connection is identified by a set of connection ID's. This notion of a connection ID is new in comparison to TCP. The use of an ID allows for connection migration when something changes in the underlying UDP/IP protocols, examples are when NAT rebinding occurs or a mobile user changes network during connection (for example from WiFi to cellular)[11]. It is worth mentioning that zero length connection IDs are allowed and when such packet arrives, the local address and port (4-tuple) are used to identify the connection. However, zero length connection IDs do not allow for multiplexing connections on the same IP address and port combination without failures.[8]

The connection IDs and their length are chosen by the endpoints themselves and are exchanged during and after the handshake. The length of connection IDs can vary between 0 and 20 bytes. Each endpoint accepts multiple connection IDs for a single connection and these are all sent in NEW_CONNECTION_ID frames in encrypted packets after handshake completion, so no outside observer knows this list. This prevents an attacker from knowing to which connection a connection ID belongs. When an endpoint moves and the connection migrates, it can use one of the other connection IDs that its peer accepts in order to prevent a third party from tracking the endpoint by looking at the connection ID. This has to be done because the used connection ID is not encrypted in a QUIC packet. An example of the exchange of connection IDs and their usage is shown in Figure 2.2.

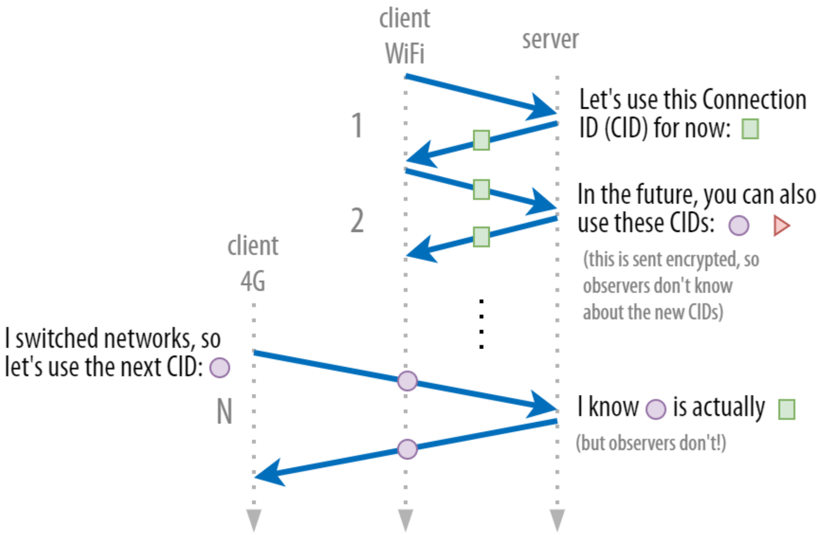

client
WiFi
server
Let's use this Connection ID (CID) for now: □
1
In the future, you can also use these CIDs: ○ ▷
2
(this is sent encrypted, so observers don't know about the new CIDs)
client
4G
I switched networks, so let's use the next CID: ○
N
I know ○ is actually □
(but observers don't!)

Figure 2.2: Example of connection ID exchange and usage. From: [1]

2.1.2 Streams

A stream is a byte-stream abstraction to an application. A QUIC connection can contain multiple unidirectional or bidirectional streams and can open/close them during its entire lifespan. In the example of loading a web-page on the Internet this can mean separate streams for HTML files, CSS files, images, etc. This multiplexing of streams within the transport protocol itself prevents head-of-line blocking[2] (which is still present in the transport layer in streams with HTTP/2 over TCP)[13, 1]. The stream data of a single stream is encapsulated in a stream frame.

2.1.3 Packets and frames

QUIC endpoints communicate by exchanging packets using UDP datagrams. Long packet headers are used for Initial, 0-RTT, Handshake and Retry packets as well as for the version independent Version-negotiation packet. To minimize overhead, short packet headers are used for 1-RTT packets after a connection is established and 1-RTT keys are available. Table 2.1 lists the packet types with a short explanation.

Table 2.1: Packet types with header form and explanation

Packet type	**Header form**	**explanation**
Version negotiation	Long header	Versionless packet for negotiating the QUIC version
Initial	Long header	First packets carrying CRYPTO frames
Handshake	Long header	Carries cryptographic handshake messages
Retry	Long header	Server uses this to validate the client's address
0-RTT	Long header	Carries limited amount of data before handshake completion
1-RTT	Short header	Carries data after handshake completion

A QUIC packet payload can consist of multiple 'frames' (for example an ACK frame, flow-control frame and stream frame). There exists a multitude of frame types, one of which is the stream

[2] Head-of-line blocking occurs when a line of packets is held up by a previous packet that was not/incorrectly received.

frame. A QUIC packet can contain multiple stream frames (from the same or from different streams). A detailed list and description of all packet types and frame types can be found in RFC9000[8]. The notion of frames will not be further used in the context of this thesis as they are part of the payload, which is encrypted in its entirety.

1-RTT packet format

As mentioned before, 1-RTT QUIC packets use a short header type. This header is made up of 3 main segments; the QUIC flags/info, the destination connection ID (DCID) and the packet number. The QUIC flags byte is made up of the following bits, where the LSB is at the bottom and the MSB at the top:

- one header form bit (0 for short header, 1 for long header)
- one fixed bit (always 1)
- one spin bit
- two reserved bits (for later additions to the protocol)
- one key phase bit (indicates the key phase, see section 2.1.6)
- two packet number length bits (indicates the amount of packet number bytes minus 1)

The destination connection ID can vary in length between 0 and 20 bytes and the packet number field can vary between 1 and 4 bytes. Parts of this header are encrypted as explained in section 2.1.5. Next, comes the payload followed by a 16-byte authentication tag. Figure 2.3 shows the format of a 1-RTT QUIC packet where the green sections are encrypted in the header protection stage and the orange parts are encrypted in the payload protection stage.

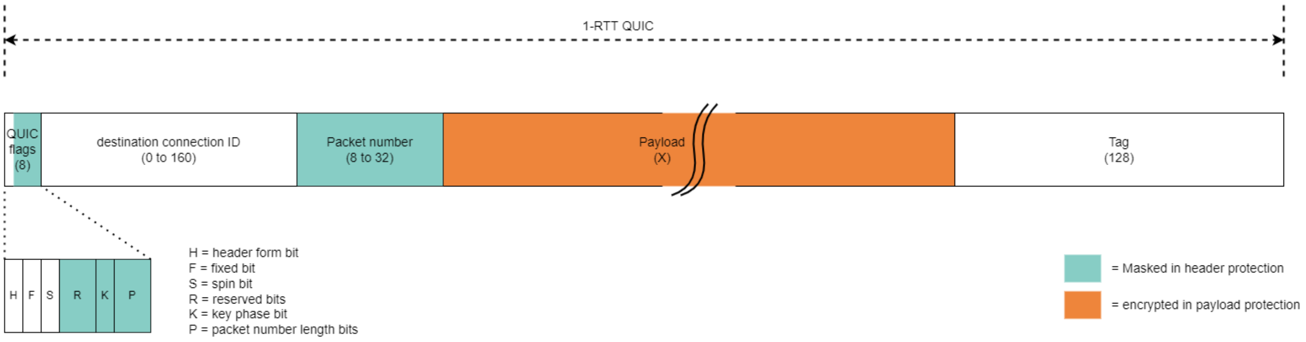


Figure 2.3: Example of a 1-RTT QUIC packet format with size ranges in bits

2.1.4 Handshake

QUIC uses a combined cryptographic and transport handshake. The cryptographic handshake is done with the CRYPTO frame. RFC9000's QUIC version uses TLS from RFC9001[9] for the cryptographic handshake, though other versions could use different handshakes. The cryptographic part of the handshake negotiates the secrets that will be used to encrypt the following QUIC packets.

TLS allows for 0-RTT and 1-RTT handshakes. 1-RTT means all cryptographic negotiation is completed in just a single round-trip, after which encrypted application data can be exchanged. 0-RTT further speeds this up by allowing application data to be sent in conjunction with the handshake. This is an improvement over TCP with TLS, which has seperate handshakes. A comparison between QUIC with TLS and TCP with TLS is shown in Figure 2.4. QUIC now combines this 0-RTT or 1-RTT handshake with the transport handshake. It is worth mentioning that the 0-RTT handshake with QUIC has limited benefits due to security risks like amplification attacks and replay attacks.

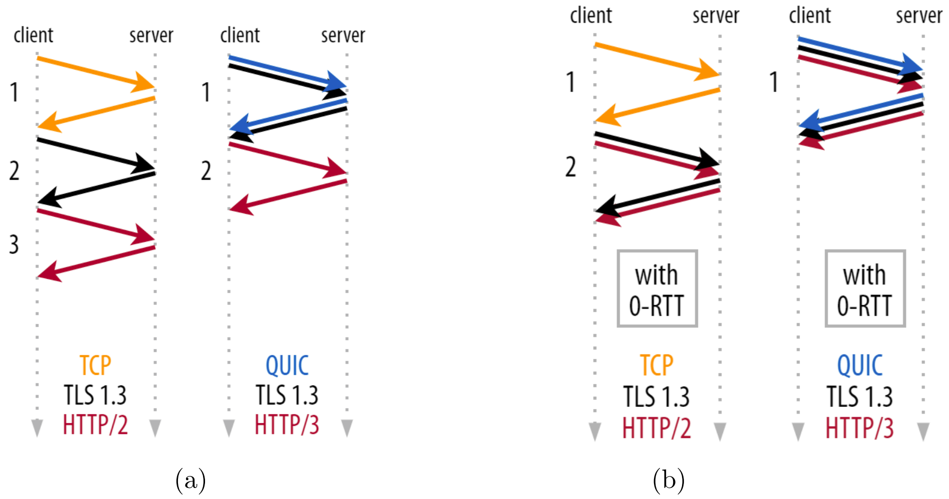


Figure 2.4: Handshake comparison between TCP and QUIC for (a) 1-RTT and (b) 0-RTT. From: [1]

2.1.5 Encryption and authentication

TLS over TCP uses *TLS records* to exchange TLS messages, providing a strict layering of both protocols. In contrast, QUIC with TLS is less layered and both protocols interact more. In TLS over TCP, packets are protected by sending the payload in TLS application data records. Contrary to that, QUIC itself provides the packet protection and authentication. Here the keys and secrets are provided by TLS in the handshake but encryption and authentication is done by QUIC.

Different QUIC packets have different levels of encryption applied to them as mentioned in Section 2.1.3. Concretely, 4 types of packets with differing encryption can be separated:

- Version negotiation packets lack any cryptographic protection
- Retry packets use AEAD_AES_128_GCM
- Initial packets use AEAD_AES_128_GCM with keys derived from the Destination Connection ID field of the first Initial packet sent by the client.
- All other packets have strong cryptographic protections using keys and algorithms negotiated in the handshake as explained in section 2.1.4

QUIC packets are encrypted in 2 stages: packet protection and header protection. In the packet protection stage, the payload is encrypted and authenticated using the AEAD function as explained in section 2.3. When packets are formed, the AEAD protection happens before header protection. When processing packets, the AEAD function can only happen after removing the header protection. This is because the plaintext header serves as additional authenticated data to the AEAD function. An example of this flow for decryption is shown in Figure 2.5.

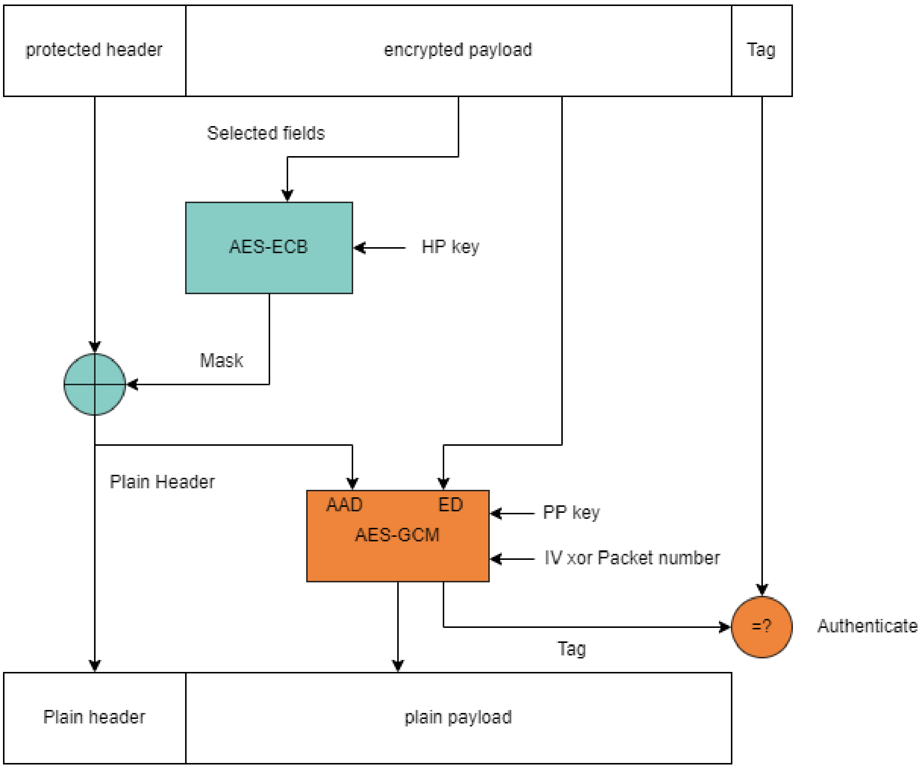


Figure 2.5: decryption flow for arriving QUIC packets with the AEAD_AES_128_GCM or AEAD_AES_256_GCM suite

Payload protection

The inputs and outputs for the AEAD function in QUIC packet protection are the following:

- Nonce N is the XOR of the IV and the padded packet number (described in section 2.1.6)
- Associated data A is the plaintext packet header starting from the first byte up to and including the plaintext packet number
- Plaintext P is the entire payload of the packet (excluding the header)
- Ciphertext C is the encrypted version of the plaintext which replaced P
- Tag T is the authentication tag which is concatenated to C

When protecting the packet, the new payload of the packet is thus the original header, followed by the ciphertext C and authentication tag T. When removing the packet protection, the output tag T' is compared to the tag in the packet to check the packet's integrity. When using the AEAD_AES_128_GCM suite with IETF's QUIC version 1, the tag is always 128 bits long.

Header protection

The application of header protection is a key difference between TCP (which lacks it) and QUIC, and is applied to all packet types except retry or version negotiation packets. The header is protected using the 'hp_key' as described in section 2.1.6 and remains the same for the duration of the connection. Header protection is applied to the 4 least significant bits of the first byte in case of a long header, the 5 least significant bits of the short header and in both cases the packet number is also protected. The protected bits of the first byte include the reserved bits, the packet number length and also the key phase bit in case of the short header.

The header protection makes use of a cryptographic algorithm that depends on the negotiated

AEAD suite in the handshake. In this research, that algorithm is AES, which is used in Electronic codebook mode (ECB). The ciphertext from the protected payload (see section 2.1.5) is sampled and serves as the input to the encryption algorithm. The 5 least significant **bytes** of the AES-ECB output are the mask. The masking operation is a bitwise XOR and is applied in the following manner for a short header:

- The 5 least significant bits of the first header byte are masked with the 5 least significant bits of first byte in the mask
- The Packet number is masked with the equivalent amount of remaining bytes in the mask

For AEAD_AES_128_GCM suite with short header packets, the input of the AES-ECB block is the 16 bytes of the payload starting from 4 bytes after the DCID. Because that input is the encrypted cyphertext, the entropy is high and same goes for the hp_key which is generated with a HKDF-function. The output of the AES-ECB block can be seen as a one-time pad. However, the hp_key is only pseudorandom, so perfect secrecy is not guaranteed. Nevertheless, the amount of bits that are XORed with this pseudorandom one-time pad is small enough for AES-ECB to be acceptable to use.

2.1.6 Keys

The secrets described in the following sections are all derived by TLS through the use of a HMAC-based key derivation function (HKDF)[14] with the negotiated hash function.

Header protection key

The header protection key 'hp_key' is the secret used in the cipher to protect the header and is derived separately from the payload protection secrets. This key doesn't change for the entire duration of a connection, allowing the key phase bit to be encrypted and always decrypted by both peers.

Payload protection keys

The payload protection secrets are used in the AEAD function for authenticating and encrypting/decrypting the payload. These secrets are the AEAD key and IV, which can change during a connection to avoid reaching the usage limits of the secrets for a specific AEAD suite. A key update is issued by toggling the 'key phase bit' in the header and using the new key and IV, allowing for two phases: 0 and 1. The receiver thus has to decrypt the packet header first to check the 'key phase bit' and packet number. If such a packet arrives with a newer packet number than the last received packet, and the key phase has changed, the receiver knows it has to use an updated key and IV to decrypt the payload. When decrypting a packet, the input for the nonce/IV of the AEAD function is not directly the IV but the XOR of the IV and the *left padded* packet number.

To calculate a new key or IV, the following function has to be used:

$$new\_secret = HKDF(current\_secret, "quicku", "", Hash\_length) \tag{2.1}$$

The only variable in this function is the *current_secret*, which allows a peer to calculate the next key before a key update is initiated. The next key should always be calculated beforehand, as it is otherwise possible to detect if a key update was issued. It is also advised to remember the previous key after a key update for at least one RTT in order to decrypt out-of-order packets with that older key.

2.2 Underlying Ethernet/IP/UDP stack

In the context of this research, QUIC packets are sent in UDP datagrams, which are in turn encapsulated in IP packets. The LAN is Ethernet based, so the IP/UDP/QUIC stack is put inside Ethernet frames. The fields of those protocols that will be important for the hardware design are outlined in this section.

2.2.1 Ethernet

Ethernet is a datalink layer protocol that uses frames for sending packets between nodes in a Local Area Network (LAN). Most modern machines use 'Ethernet II' or version 2 of this protocol, Figure 2.6 shows the header format. Note that the bytes in an Ethernet frame arrive from **most significant byte to least significant byte** (left to right in Figure 2.6, while the bits within each byte arrive from **least significant bit to most significant bit** [15].

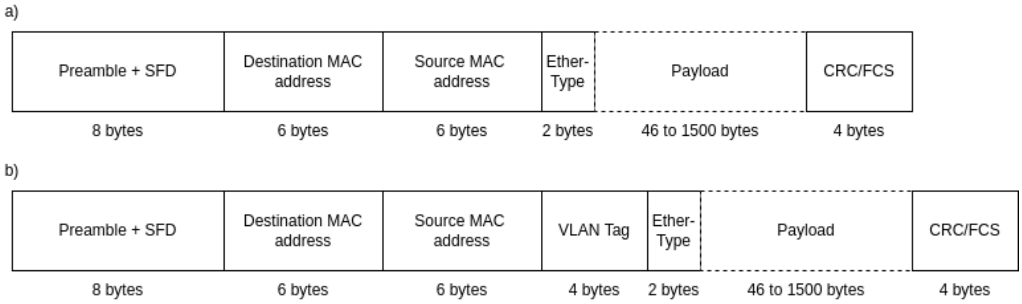


Figure 2.6: Layer 2 Ethernet frame header format without VLAN (a) and with VLAN (b)

The Preamble and start frame delimiter (SFD) are 8 bytes long and are part of the physical layer Ethernet to signal the start of a frame. In the datalink layer version of Ethernet these bytes are not present. The maximum payload length is 1500 bytes and thus the maximum frame length is for Layer 2 Ethernet is 1518 bytes.

In layer 2 Ethernet; bytes 13 and 14 indicate the EtherType, essentially meaning which protocol is carried in the payload. If these bytes equal 0x8100, then the frame is VLAN tagged and the actual EtherType bytes are at postion 17 and 18. For the Internet protocol the EtherType is:

- 0x0800 for IPv4
- 0x86DD for IPv6

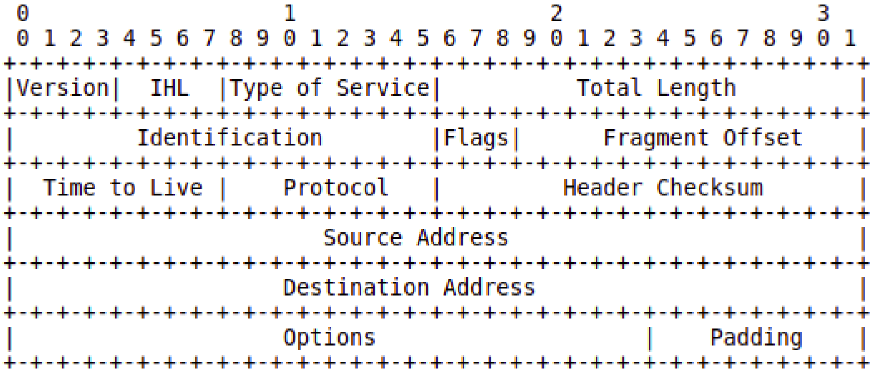
2.2.2 Internet protocol

The Internet protocol (IP) is a network layer protocol that uses fixed length IP addresses to deliver packets from one node on the Internet to another. IP is currently mainly deployed in two versions: version 4 (IPv4) and version 6 (IPv6).

IPv4

The header for an IPv4 packet is shown in Figure 2.7. Following is a list of the most important fields[2]:

- Version - IP version (4 in this case)
- IHL - Internet header length in 32-bit words
- Total Length - entire packet size in bytes, this includes header and payload
- Protocol - identifies the protocol used in the payload. For UDP this is 0x11



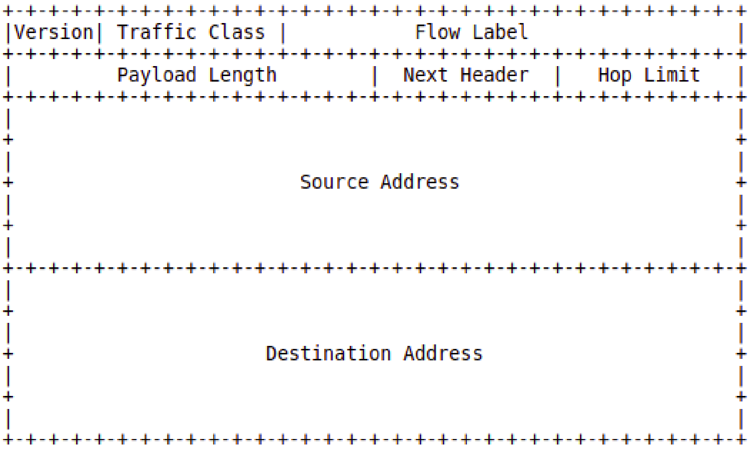
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.7: IPv4 packet header format, the numbering is in bits. From:[2, p. 10]

IPv6

The header for an IPv6 packet is shown in Figure 2.8. Following is a list of the most important fields[3]:

- Version - IP version (6 in this case)
- Payload length - length of the payload (excuding IP header) in bytes
- Next header - identifies the protocol used in the payload (same as the protocol field in IPv4). For UDP this is 0x11

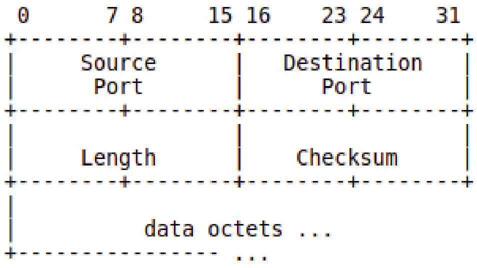


```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version| Traffic Class |           Flow Label                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Payload Length        |  Next Header  |   Hop Limit   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                         Source Address                        +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                      Destination Address                      +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.8: IPv6 packet header format, the numbering is in bits. From:[3, p. 4]

2.2.3 User datagram protocol

User datagram protocol (UDP) is an unreliable transport layer protocol for delivering datagrams from one endpoint to the other. The source and destination ports respectively signify the origin and destination process sending the data. It has a minimal header of only 8 bytes as shown in Figure 2.9. The length field is the length in bytes of the UDP-header and UDP-payload combined.



```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|     Source      |   Destination   |
|      Port       |      Port       |
+--------+--------+--------+--------+
|                 |                 |
|     Length      |    Checksum     |
+--------+--------+--------+--------+
|
|          data octets ...
+---------------- ...
```

Figure 2.9: UDP header format, the numbering is in bits. From:[4, p. 1]

2.3 Authenticated encryption with associated data (AEAD)

AEAD is an interface for providing confidentiality and integrity of a plaintext and is described in RFC5116[16]. It defines confidentiality for parts of data and also provides integrity protection for parts of data (called 'associated data'). Note that not all associated data has to be encrypted. The interface also defines the opposite operation of authenticated decryption. In case of encryption the inputs are a secret key K, a nonce N (which may be zero length), associated data A and the plaintext P and the output is the ciphertext C, with the same length as P. For decryption the inputs are K, N, A and C; and the output is P.

The confidentiality protection means that the plaintext P that needs to be protected is transformed into a seemingly random blob of data (ciphertext C) that is not readable to outsiders by using a key K. The ciphertext can only be transformed back to the plaintext using the same key, which only those who are allowed to read the plaintext should have.

The integrity protection provides a means to check if all data (plaintext P and associated data A) has not been tampered with. With the use of the key, all data is transformed into a fixed length tag T, through the use of a hash function. The tag is concatenated to the message. modifying only a single bit in the data generates an entirely different tag. This allows a holder of the key to perform the same operation on the data (without the tag) to produce their own tag. If the tag exactly matches the tag on the end of the message, the data is successfully authenticated. If the tag is different, an authentication FAIL will be returned, signalling that the data was not original.

2.3.1 AEAD_AES_128_GCM

For this research, the AEAD_AES_128_GCM suite will be used. This is explained here in the context of usage with QUIC. Following lengths apply to this algrorithm:

- length of K is 16 bytes (128 bits)
- length of T is 16 bytes (128 bits)

- length of N 12 bytes
- length of C is 16 bytes longer than plaintext P (length of P + length of T)

The cipher is AES and when used with key K, it is denoted as $CIPH_K$ and is used in Galois Counter Mode (GCM)[5]. In the following sections encryption and decryption in GCM using AES as the block cipher is explained.

Encryption

The algorithm for authenticated encryption in GCM is shown in figure 2.10. The inputs are the Nonce/IV, the plaintext P, additional authenticated data A and the key K. The output is the ciphertext C and the authentication tag T. The following steps are executed:

1. The hash subkey H is calculated using the cipher block function (AES) with its input being a 128 bits long string of 0's
2. The Pre-counter block J_0 is formed by adding the a 32-bit counter initialized with value 1 to the back of the 96 bits of IV (Nonce)
3. The plaintext P is encrypted by performing the GCTR function with the J_0 after a single increment of the counter. The output of this encryption is the ciphertext C.
4. The input for the GHASH function is formed by concatenating A (padded with 0 to a multiple of 128 bits) with P (padded with 0 to a multiple of 128 bits) and also with the 64-bit representation of the length of A followed by the length of C, also described in 64 bits
5. The GHASH function is executed using the input from step 4 and H from step 1
6. The output of the GHASH function is provided as input to the GCTR function with the original value for J_0 from step 2
7. The output of the GCTR function from step 6 is truncated to 128 bits and this is the tag T

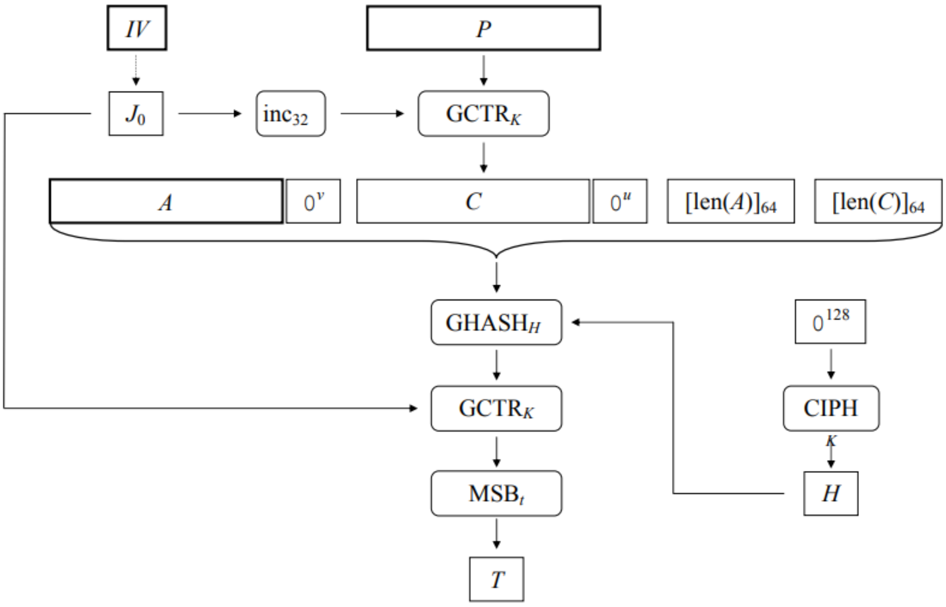


Figure 2.10: Block diagram depicting authenticated encryption in GCM. From:[5, p. 16]

Decryption

The algorithm for authenticated decryption in GCM is shown in figure 2.11. The inputs are the Nonce/IV, the ciphertext C, additional authenticated data A and the key K. The output is the plaintext P or an indication that the authentication failed. The following steps are executed:

1. The hash subkey H is calculated using the cipher block function (AES) with its input being a 128 bits long string of 0's
2. The Pre-counter block J_0 is formed by adding the a 32-bit counter initialized with value 1 to the back of the 96 bits of IV (Nonce)
3. The ciphertext C is decrypted by performing the GCTR function with the J_0 after a single increment of the counter. The output of this operation is the decrypted plaintext.
4. The input for the GHASH function is formed by concatenating A (padded with 0 to a multiple of 128 bits) with C (excluding the Tag T and padded with 0 to a multiple of 128 bits) and also with the 64 bit representation of the length of A followed by the length of C, also described in 64 bits
5. The GHASH function is executed using the input from step 4 and H from step 1
6. The output of the GHASH function is provided as input to the GCTR function with the original value for J_0 from step 2
7. The output of the GCTR function from step 6 is truncated to 128 bits and this is the tag T'
8. Compare the T' with T and, if they are equal, the authentication is successful. Otherwise the authentication failed

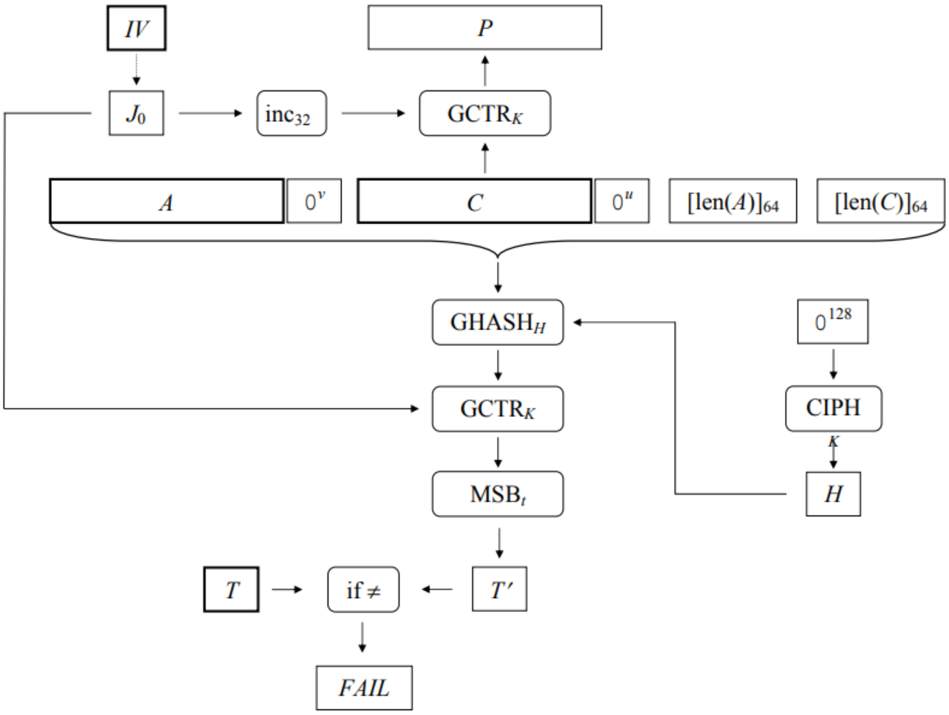


Figure 2.11: Algorithm for authenticated decryption in GCM. From:[5, p. 18]

GCTR function

The GCTR function is used to encrypt the plaintext or decrypt the ciphertext and also to produce the tag from the output of the GHASH function. This is done by splitting the input in blocks of 128 bits. The initial counter block ICB (= IV||0x00000002) is encrypted with the chosen encryption algorithm (for example AES) with the provided key and is XORed with the first input block. The result of this is also the first block of output data. The 32 least significant bits of the ICB are modular incremented by 1 and encrypted again. The result is XORed with the second block of input data. This continues up to and including the last block of input data. Figure 2.12 shows this process.

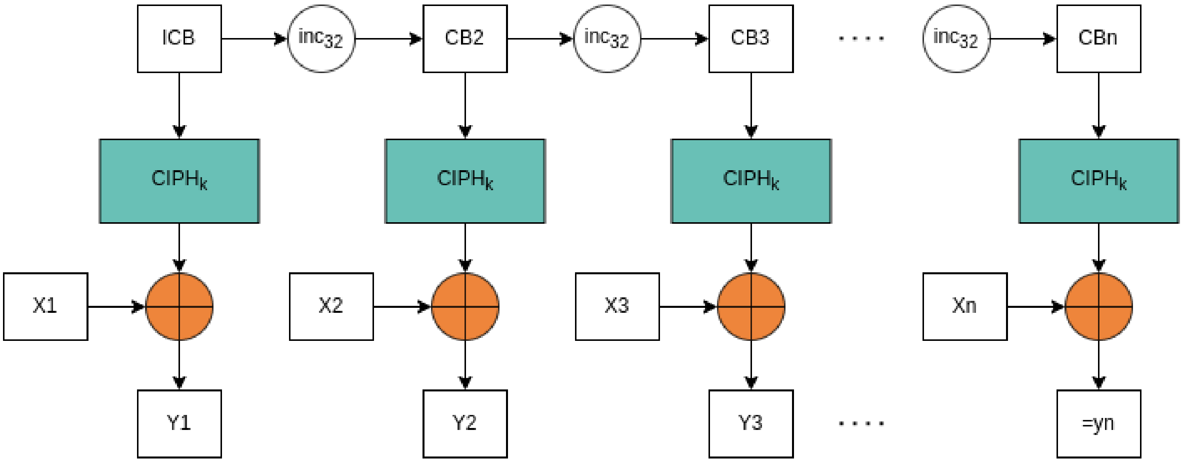


Figure 2.12: Block diagram of the GCTR function

GHASH function

The GHASH function is used to create a state that is dependant on all the input data. This is done by splitting the input (associated data, cipher/plaintext and concatenation of lengths) in blocks of 128 bits. Starting from the first block, a modular multiplication is done with the value of H ($GCTR_k$ of 128 zeros). The output of this is XORed with the next block and the result is again multiplied with H. This continues until the last block is XORed and multiplied. The result of the final multiplication is the result of the GHASH function. Figure 2.13 shows this process.

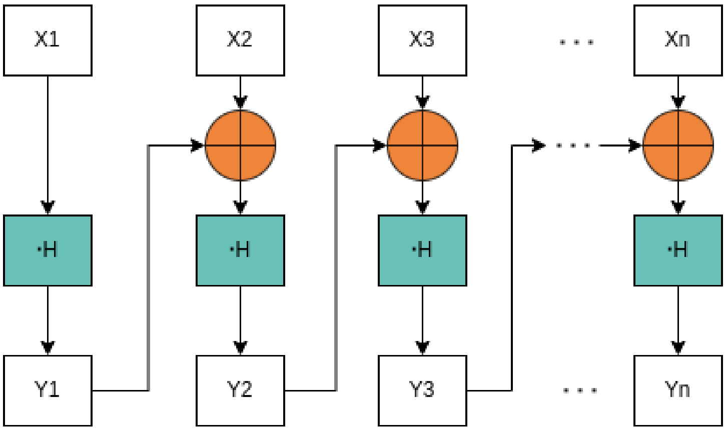
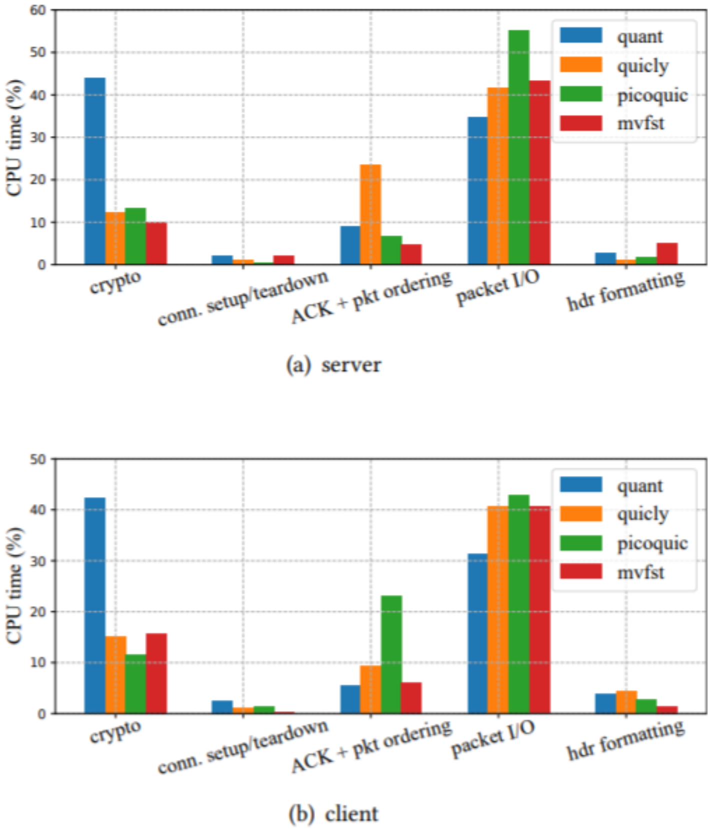


Figure 2.13: Block diagram of the GHASH function

2.4 Related work in QUIC hardware offloading

A closely related work to QUIC offloading on hardware was proposed by Yang et al. [6]. Here, the researchers did extensive performance tests on 4 C/C++ implementations (Quant, Quicly, Picoquic and Facebook's Mvfst). In these tests: CPU time for cryptography, connection setup and teardown, ACK and packet ordering, packet IO and header formatting were measured. The results for a single connection are shown in Figure 2.14.



(a) server

(b) client

Figure 2.14: CPU usage breakdown in % by both server(a) and client(b) for the different QUIC implementations in a single connection scenario. From: [6, p. 22]

These results show that CPU time is mostly spent on packet IO, followed by crypto operations and ACK + packet ordering (with an exception for quant which uses a different IO engine). The packet reordering is dependant on the amount of packets that arrive out-of-order. As such, the paper lists the effect of packet reordering at different out-of-order packet rates, where Quant, quickly and mvfst showed significant throughput drop-off at increasing reordering rate. Picoquic was able to keep its throughput constant even at a reordering rate up to 10%. The reason the researchers give for this is; when the implementation does not handle the reordering fast enough, the packet loss timeout may expire and the packet is seen as 'lost'. This shrinks the congestion window and decreases throughput.

Another factor in packet reordering is the algorithm that is used to do so. The researchers tested this on the picoquic implementation with a linear and a splay algorithm at different packet loss rates. The linear algorithm has a higher throughput at low packet loss rates but drops of more quickly, making the splay algorithm more efficient with high packet loss rates and the linear algorithm at low loss rates.

In the multiple connections scenario, picoquic and mvfst show an increase in throughput when the number of connections increases (up to a certain point). In case of out-of-order packet arrival or packet loss the results are similar to the single connection scenario (e.g. picoquic's throughput remains constant, the throuhgput of mvfst and quickly drops).

The three main takeaways from [6] are:

- packet-IO uses the most CPU resources (around 50%)
- crypto uses the second most CPU resources
- the timely handling of packet reordering increases throughput.

A practical implementation of hardware offloading was done at Intel, where researchers offloaded segmentation of the packets and also the cryptography for payload protection[17]. The offloading was done on server grade hardware (Intel Ethernet Controller XL710 40Gbe backplane Adapter, connected to an Intel Arria 10 FPGA). The offloaded cryptographic algorithm was AES-GCM (AEAD_AES_128_GCM, AEAD_AES_128_CCM, AEAD_AES_256_GCM, AEAD_AES_256_CCM). This offloading was compared with a software implementation that was already accelerated through dedicated hardware instructions. The offloads were only done on short header packets as they are most commonly used for the actual application data transfer. Furthermore, the used implementation was based on draft 11 of the IETF standard[18], which means no packet header encryption was done and the offloading is only used for payload protection.

A number of tests were conducted in which a number of clients send requests to a QUIC server. The number of requests and size of data served by the request were kept constant between experiments. The amount of CPU cycles for completing the task as well as the time to complete all the requests and the average and maximum throughput achieved were measured. On average, the segmentation offloading showed a 10% reduction in CPU utilization and 12% higher throughput. Similarly, crypto-offloads showed a 13% reduction in CPU utilization and a 13% increase in throughput. Combining both segmentation and crypto-offloading resulted in a 16% CPU utilization improvement and throughput increased by 32% on average.

The researchers from [17] also propose an interface for the passing of cryptographic parameters and segmentation information from user-space to the offloading hardware.

Chapter 3

Design and implementation

In this chapter a hardware/software co-design for decrypting QUIC packets is constructed using a top-down design approach. First an ideal design is imagined with near unlimited design resources and time available. Next, a proof of concept design is discussed where parts of the ideal design are left out or simplified in exchange for hardware resources and development time.

3.1 Ideal design

In this section an ideal design for offloading the decryption of QUIC packets is outlined and analysed. The main goal of this 'ideal' design is to decrypt QUIC packets immediately after their arrival on the hardware Ethernet interface. The reason for this is because the QUIC stack is situated in user space and the data copy between user space and kernel space is expensive. A drawback of this ideal design is that all processing steps that happen before the offloading task when receiving a packet (or after sending) have to be done in hardware as well. Figure 3.1 shows this ideal workflow for decryption.

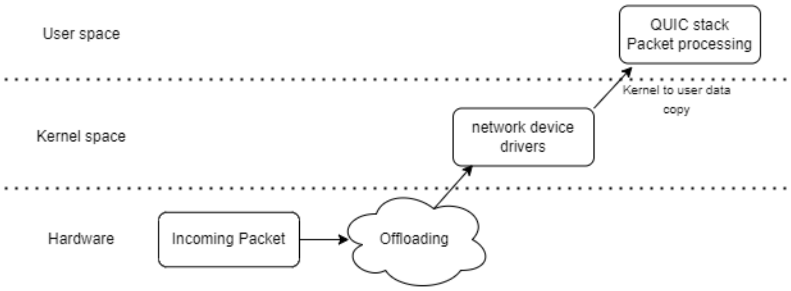


Figure 3.1: Ideal hardware/software co-design for decryption with hardware offload before data copy to user space

A less efficient solution would thus be; sending the encrypted packet to the QUIC stack on arrival and then checking in software which connection it belongs to and what the session keys are, etc. The message, keys and nonce would then be sent through a device driver to the offloading hardware, which decrypts the payload and sends the plaintext back to user space. Now there is the added overhead of not only copying the entire payload from user to kernel space and back, but also the sending of data to and from the offloading hardware, over a bus interface. This way, most of the benefits from the hardware decryption are cancelled out by the data copy. A graphical representation of this workflow is shown in Figure 3.2.

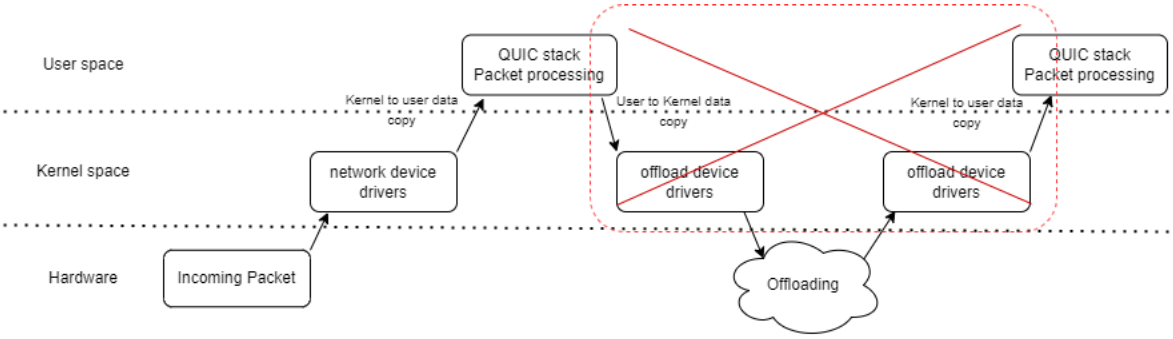


Figure 3.2: Less performant hardware/software co-design with hardware offload between user space processing

The ideal design comes with some added complexity in hardware. Before any offloading can be done, the incoming Ethernet frame has to be parsed to check if it contains a QUIC packet. If so, a check has to be done if the packet is from an existing connection for which keys are available. This can be done by parsing the destination connection ID and doing a lookup in a connection table for this ID. If the lookup has a match, an address is returned, with which all the keys can be retrieved from memory. Once the keys are retrieved, the header protection can be removed. When the plaintext header is available, the key phase can be determined and the correct key/IV combination can be selected for removing the payload protection. In the payload protection removal stage the packet is decrypted and it is here that the data flowing in the pipeline has to be replaced with the decrypted bytes. At the end of the payload protection removal, a tag will be returned from the hardware AEAD function, which will be compared to the last 128 bits of the QUIC payload. The message is now decrypted and authenticated. If the authentication failed, the packet is discarded before it reaches the software, saving it from needless processing and data copy. Ideally, a counter of failed authentications should be kept so the software can use it to identify possible problems or attacks. If no QUIC packet was detected, the packet simply passes through the pipeline without any replacement of bytes.

Finally, an important detail still has to be mentioned: How does the QUIC stack know if the incoming message has already been decrypted? An option would be to explicitly add a bit/byte as a flag in front of the Ethernet frame and notify the the kernel drivers of what this flag is, so it can be propagated up to the QUIC stack to indicate if the packet was decrypted. However, this is a cumbersome task as it requires modification of the underlying protocol stacks like Ethernet, IP and UDP. Adding this byte in front of the QUIC packet payload (so within the UDP/IP/Ethernet payload) also adds complexity because this would falsify the checksums in the underlying protocols and invalidate their length fields, requiring logic to update them.

Another option is the implicit option. Here, the QUIC stack saves a shadow copy' of the DCID's for which it has passed the keys to the hardware. This way it knows that all incoming packets from a DCID that is present in the hardware LUT have already been decrypted on arrival. Figure 3.3 shows a general design flowchart for this decryption offload

3.2 Proof of concept design

To test the viability of the ideal design with limited resources and time, a proof of concept (PoC) is made that aims to follow the general design flow of the ideal design, while making compromises

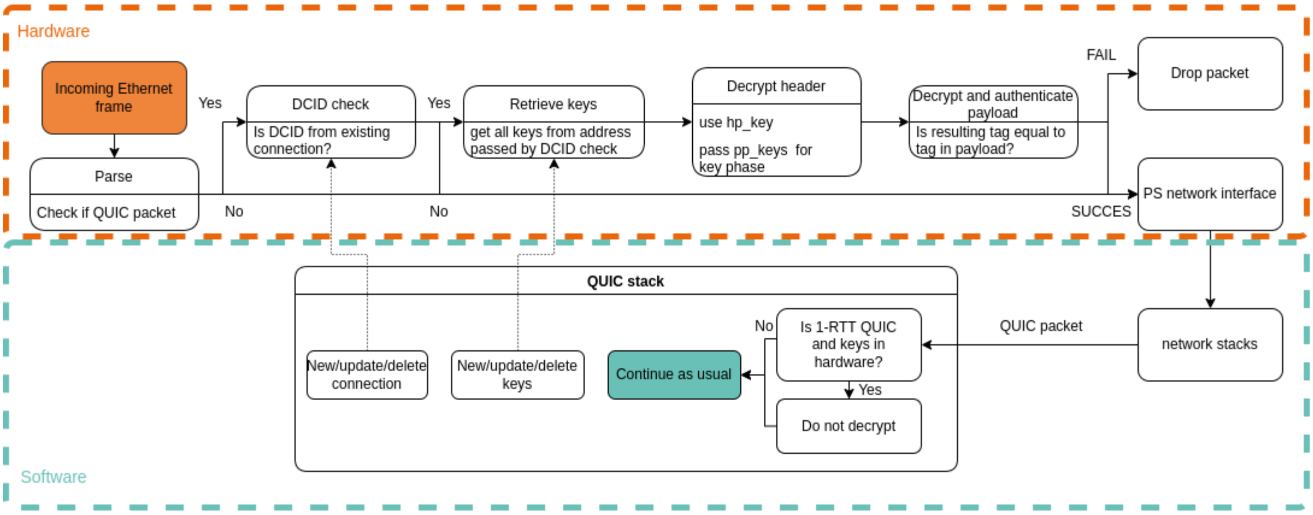


Figure 3.3: ideal hardware/software co-design for QUIC packet decryption offloading

in the completeness. This is possible because for tests, certain parameters can be chosen, like which combination of network stacks are used for sending the data. Figure 3.4 shows the design flow of the proof of concept for decryption of 1-RTT QUIC packets.

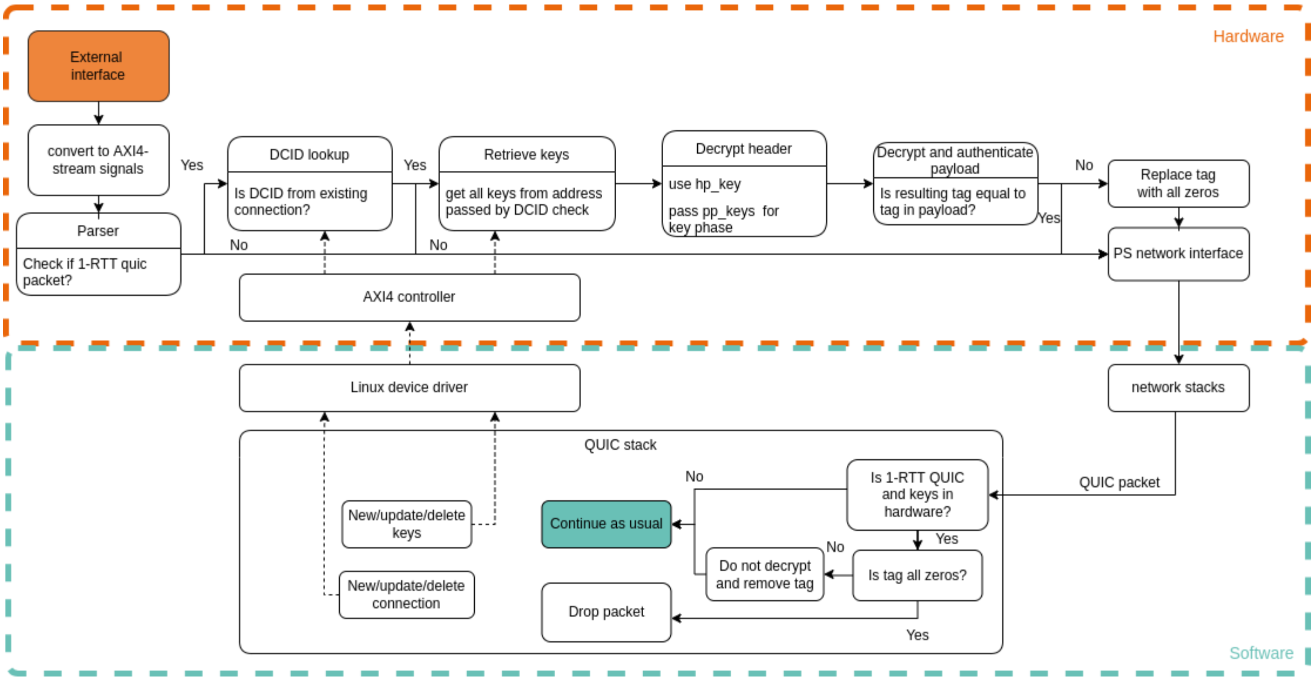

Hardware
External interface
convert to AXI4-stream signals
Parser
Check if 1-RTT quic packet?
Yes
No
DCID lookup
Is DCID from existing connection?
Yes
No
Retrieve keys
get all keys from address passed by DCID check
Decrypt header
use hp_key
pass pp_keys for key phase
Decrypt and authenticate payload
Is resulting tag equal to tag in payload?
No
Yes
Replace tag with all zeros
PS network interface
AXI4 controller
Linux device driver
network stacks
QUIC stack
New/update/delete keys
New/update/delete connection
Continue as usual
No
Is 1-RTT QUIC and keys in hardware?
Yes
QUIC packet
Do not decrypt and remove tag
No
Is tag all zeros?
Yes
Drop packet
Software

Figure 3.4: proof of concept hardware/software co-design for QUIC packet decryption offloading

Firstly, the parser for the proof of concept will only detect possible 1-RTT QUIC packets in an Ethernet/IPv4/UDP stack. All downstream modules will use metadata which is generated by this parser. The parser works independent of the stacks underneath UDP. A commercial design would only need to add support for the other possible underlying stacks.

Next, the DCID lookup stage only takes into account DCID's of length 20. This is because the lookup in hardware is always done for a certain length. A solution would be to have 20 lookup tables for all the possible DCID lengths, because a shorter DCID could be a prefix of a longer one. If this happens, the longest match should be taken. These lookups would be done in parallel so the result is found in fixed time. Another option would be do the lookup repetitively for the different lengths. However, this way the lookup is not fixed time which is in conflict with that design principle.

The next step is to retrieve the keys of a connection when a DCID match was found. The DCID lookup returns an address with which the keys can be retrieved. In this manner, multiple DCID's belonging to the same connection can point to the same keys. For every connection, the keys for the current key phase are saved in memory as well as the keys for the next key phase (which are calculated in advance). Ideally, when a key update occurs, the previous key phase should be remembered for one round-trip-time to accommodate for out-of-order packets. This would also require additional logic to check if the packet is older than the packet that initiated the key update (by checking the packet numbers). This "previous key memory" is not supported in the proof of concept design. Consequently, authentication will fail (and the packet will be dropped) if an out-of-order packet arrives that was encrypted with a key that was replaced with a newer one. Nevertheless, if the key updates do not happen too frequently, the amount of packets dropped for this reason will be negligible.

The fastest way to retrieve the keys would be to have 5 different memories for: the header protection key, payload protection keys for both key phases and IV's for both key phases. However, for the proof of concept a single memory is used that is accessed in 5 consecutive clock cycles (one for each of the keys), where every cycle the address is increased by one, starting from the base address. The advantage of the latter approach is that the individual address can be used to change a single key, while in the former approach all keys would have to be rewritten simultaneously or extra logic would be needed to select which single key would need to be changed. The changing of single keys is needed because the header protection key never changes and with a key phase update, only one pair of the secrets changes.

Once the keys are available, the header protection can be removed. For the proof of concept, the AES-ECB cryptographic function is supported. Once the header protection is removed, the key phase can be determined. The correct key and IV pair can be passed to the payload protection stage. In the payload protection stage, the AES_128_GCM AEAD function is supported. In the proof of concept design, the discarding of a packet does not happen in hardware. The reason for this is that the tag is located at the end of the packet, so long packets might already be largely in the output FIFO or even in the PS by the time the tag is compared. Subsequently, those bytes can not be dropped anymore. The solution is to replace the tag with all zeros if the authentication failed. In software, the QUIC stack has to check the tag first. If the tag is all zeros, the packet can be dropped, otherwise processing can continue and no further decryption has to be done. This approach has the possibility of dropping a correct packet if the packets actual tag is all zeros. However, the chance of that happening is 1 in 2^{128}, which is negligible. Furthermore, the packet would be retransmitted with a different packet number which changes the tag.

3.3 Practical implementation

3.3.1 The pipeline

The system is designed to be placed between the hardware Ethernet interface and the networking device on the PS side. For arriving packets, the data flows from the Ethernet interface to the networking device. The networking device is for example a FIFO or DMA device that is read by the PS. On the Ethernet interface side, there has to be a module that converts the signals from the Ethernet PHY to the used internal interface. The chosen interface for data transfer between those devices is the AMBA AXI4-Stream interface. This is mainly because of it is a ubiquitous protocol and the Ethernet PHY to AXI4-Stream conversion intellectual property (IP) blocks are available from Xilinx.

The AXI4-Stream interface is 4 bytes (32 bits) wide. The signals that are used in the decryption system are TVALID, TSTROBE, TLAST and TREADY, where TREADY is driven by the slave (PS side FIFO) and the others are driven by the master (Ethernet interface side). TVALID indicates that the 4 bytes of data (TDATA) contains valid data. TSTROBE is 4 bits wide and indicates which bytes of TDATA are valid. TLAST indicates the last byte of a packet. TREADY indicates if the slave can take data.

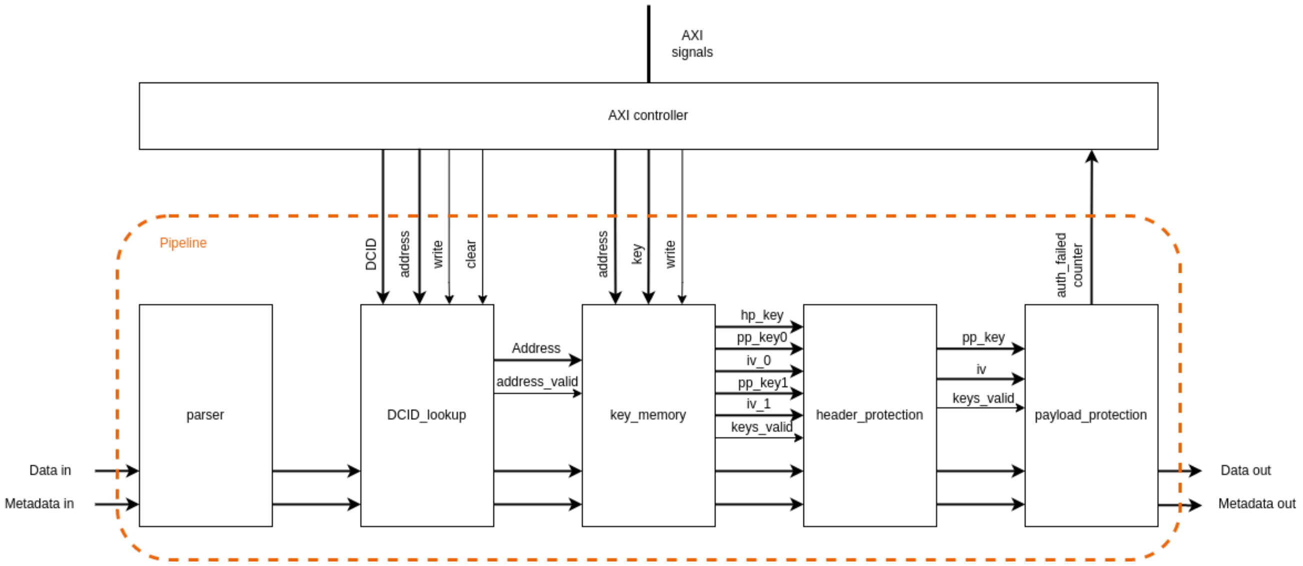


Figure 3.5: Modular representation of the proof of concept hardware pipeline with interfaces

The decryption pipeline consists of two side-by-side lines of *parallel in parallel out* shift registers between the input and output; the data path and the metadata path. The data path is 32 bits wide and always contains byte aligned data. The metadata path starts at 6 bits wide (TVALID, TSTROBE and TLAST) but expands with new metadata after the parser.

The pipeline is split up in to 5 different modules:

- the parser module
- the DCID lookup module
- the key memory module
- the header protection module
- the payload protection module

This compartmentalisation allows for the modules to be exchanged with other versions or more efficient implementations in the future, without affecting the other stages. For example, if the in- and outputs of the pipeline use a different protocol than AXI4-stream, only the parser has to be replaced with one that generates the correct metadata based on that other protocol. Another benefit is that the modules can be implemented in parallel by the researchers. To control the modules and update the keys, an AXI4-Lite control module also supervises the pipeline. Figure 3.5 shows the system with these modules and their interfaces.

3.3.2 The parser

The parsing module provides metadata on a per byte basis so the following modules can determine of which protocol each byte is. In the case of a 1-RTT QUIC packet, the possible DCID bytes are also flagged as such. The parser has to do this using the TVALID, TSTROBE and TLAST metadata signals and the data. It detects the start of an Ethernet frame when TVALID first goes high or if TVALID is high and TLAST has toggled from high to low. The locations of the fields and header lengths are fixed for an Ethernet/IPv4/UDP stack as explained in section 2.2. This allows the parser to count the byte offset to flag the bytes with their protocol. By checking the protocol field in the Ethernet frame the next header can be determined. If the next protocol is IP, the next header field can be checked to determine if the included payload is UDP. If so, the UDP bytes can be flagged as such and the payload could possibly be a QUIC packet.

In the metadata, the protocol field is 3 bits wide and is present for every byte of data. The DCID field is 1 bit per byte. The encoding of the protocols are shown in table 3.1.

Table 3.1: Encoding of the protocol fields in the metadata registers

protocol	**encoding**
Ethernet	”000”
IPv4	”001”
UDP	”010”
1-RTT QUIC	”011”
Unknown	”100”
Padding	”101”

3.3.3 The DCID lookup module

In the DCID_lookup module the passing packet is checked if it is part of an active connection for which keys are available. The potential DCID from the packet is copied on the basis of the metadata from the parser module. Those 20 bytes, that are flagged as DCID bytes, are the input of a *content addressable memory* (CAM). The CAM works by running the DCID through a hash function (Xoodoo)[19] which results in an address. That address is the address from which a memory is read or written in the CAM. The value inside the memory at that location is another address which serves as the base address where the keys of the connection belonging to that DCID are located in the key_memory module.

When performing a lookup, the DCID could possibly not be from a valid connection. Nevertheless, the hash function always has an output and thus an address. To indicate that the lookup does not belong to a connection, the value zero is returned from the memory in the CAM. Because the memory is initialized to all zeros, there are no valid connections at the start. The DCIDs can be assigned to a connection. The address at which the corresponding keys will be stored in the key_memory module, will be written to the memory of the CAM at the location of the DCID. The resulting address is thus valid, provided that this address is not zero.

If a DCID is valid, the address_valid signal at the output goes high for one clock cycle while the value for the address is also set at the output. No changes to the metadata happen if the DCID is valid. If there was no DCID match, the protocol fields in the metadata are changed from "1-RTT QUIC" to "unknown". This way the downstream modules do not perform any tasks with the packet. A more in depth explanation of how this module works can be found in [20].

3.3.4 The key memory module

When the address_valid signal from the DCID_lookup module goes high, the key_memory module starts collecting the keys. This is done by copying the 10-bit address value into a register. This address register is the input to a memory block. The output of the memory block is a connection secret. When fetching the keys, the key at the presented address is first copied into a 128-bit register. Next, the address register is incremented by 1 and the output is again copied to another register. This copying is repeated 3 more times to retrieve all the keys. The location and offsets of the keys in memory are shown in table 3.2. A more detailed explanation of this module can be found in [21].

Table 3.2: Key locations in the key memory with offsets

address	**secret**
Base address + 0	header protection key *(hp_key)*
Base address + 1	payload protection key for key phase 0 *(pp_key0)*
Base address + 2	initialization vector for key phase 0 *(iv0)*
Base address + 3	payload protection key for key phase 1 *(pp_key1)*
Base address + 4	initialization vector for key phase 1 *(iv1)*

Once all these secrets are copied into their respective registers, the keys_valid signal is driven high to signal the header protection module that the keys are available.

3.3.5 The header protection module

The header protection module does the decryption of the masked header fields as described in section 2.1.5.

Copying the keys

Firstly, if the keys_valid signal at the input is high, the header protection secrets as well as the payload protection secrets for both key phases are copied into registers. The previous modules are built in a way that this always occurs on the same clock cycle as when the first QUIC byte enters the header protection module. These keys stay in their registers as long as the QUIC packet, to which they belong, is in the hardware.

Header protection sample

The input to the AES_128_ECB module is the 128 bits sample after the packet number, under the assumption that the packet number length (PNR_len) is 4 bytes. With the constraint of 20 byte DCIDs, the offset of the first sample byte is at $1 + 20 + 4 = 25$ bytes from the start of the

QUIC packet. The start of the QUIC packet is always at the third byte in a 32-bit shift register in a Ethernet/IPv4/UDP stack, which means the last DCID byte is always in the fourth byte in a register. Figure 3.6 shows the location of the sample in an Ethernet frame in the pipeline in green. The skipped bytes are shown in orange.

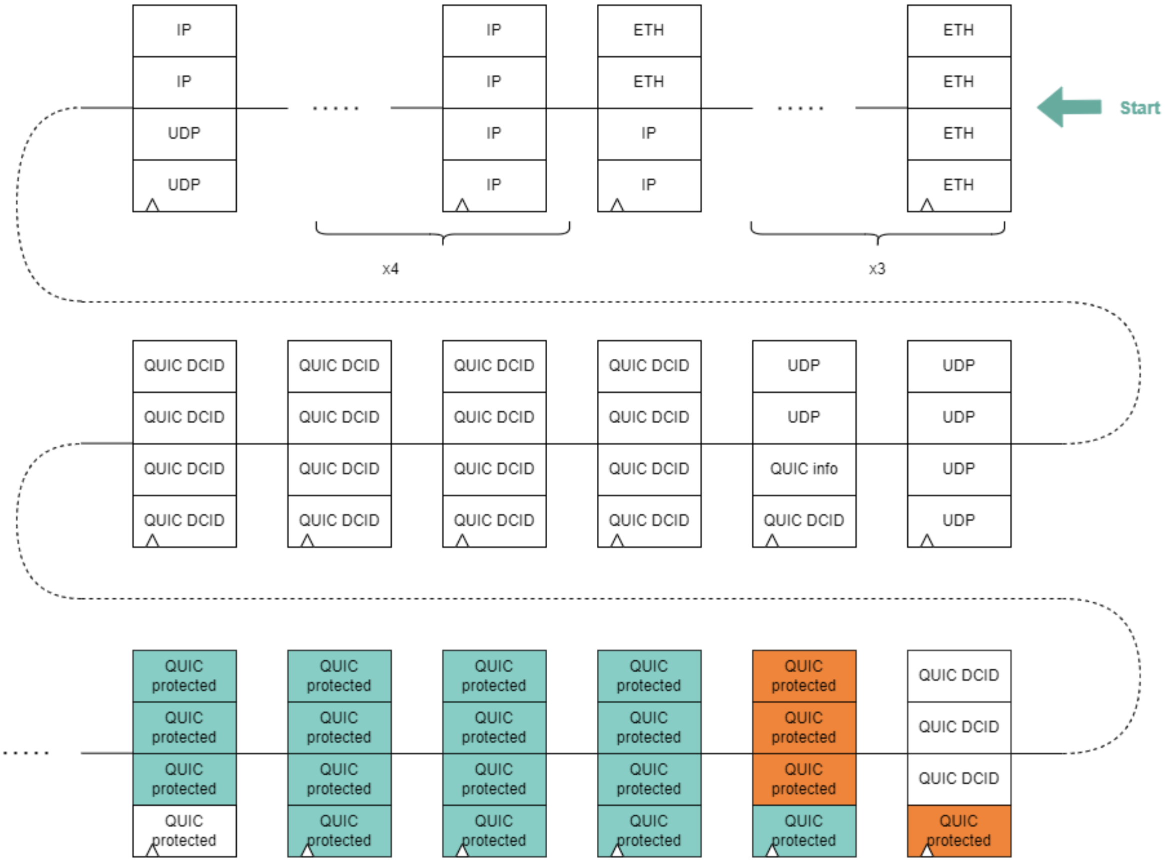


Figure 3.6: Ethernet frame containing a QUIC packet in the pipeline. The location of the sample input for AES-ECB is shown in green.

Calculating the mask

The mask is calculated using an iterative AES module for FPGAs. This means only a single AES round is implemented in hardware with an input register that is filled with the round-output on consecutive rounds. This results in relatively low FPGA resource usage. The implementation of this module was provided by ES&S.

A simple check can be done for the last DCID byte by inspecting if the DCID metadata changes from high to low. At this point, the 4 bytes after the last DCID byte are skipped and the 16 bytes after that serve as the input to the AES_128_ECB module. Furthermore, when this occurs, the AES-ECB goes to the *RUNNING* state. In this state, the AES_128_ECB module is enabled and the mask is being generated. The AES_128_ECB module takes 10 cycles to complete. This is also the amount of registers that are used as delay, to guarantee synchronization. When the AES module is done, the AES_done signal resets the state to *INIT* and indicates that the output of the AES_128_ECB module is valid. This output is the mask and can now be used to decrypt the header.

Applying the mask

When the AES_done signal is high, the 5 least significant bits of the QUIC info byte (first QUIC byte) are XORed with the 5 least significant bits of the least significant byte in the mask. The PNR_len is now decrypted (2 least significant bits of QUIC info byte) and determines how many bytes after the last DCID are XORed with the mask. All bits that were XORed are replaced with their respective output of the XOR.

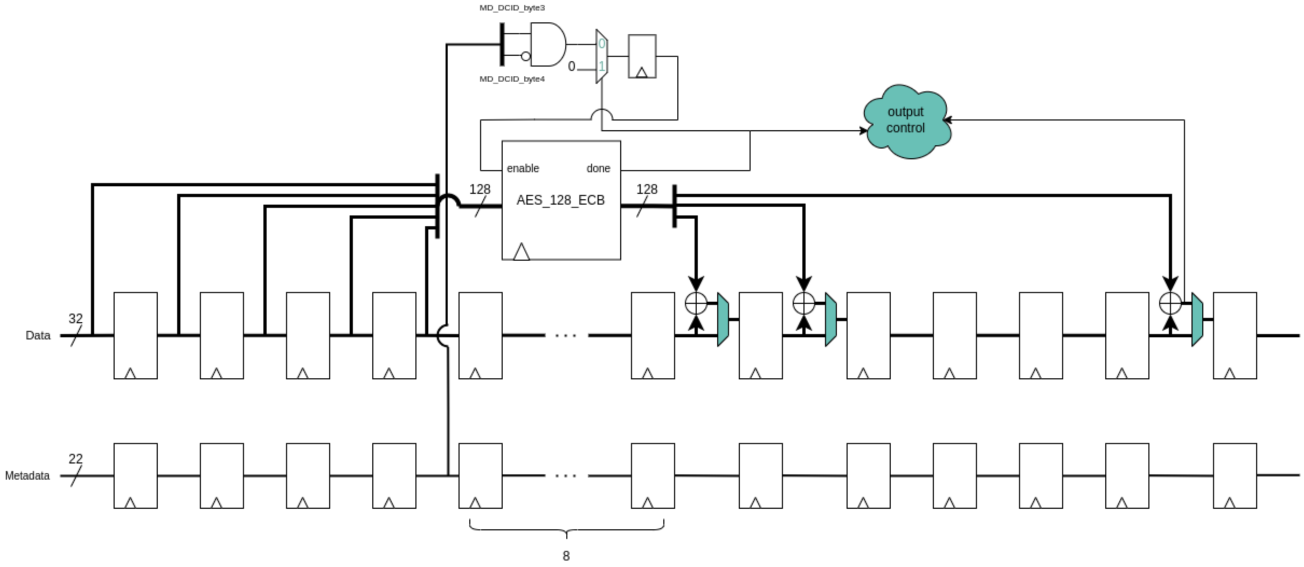


Figure 3.7: Hardware schematic of the header protection module (without key copy and key passthrough

Passing keys to the payload protection module

After applying the mask, the key phase is now the third least significant bit of the plain QUIC info byte. The pp_key and IV for that key phase are passed to the output. Next, a keys_valid signal goes high to notify the payload protection module that the correct secrets are available.

3.3.6 The payload protection module

The payload protection module decrypts the payload and authenticates the QUIC packet. If it receives a 1-RTT QUIC packet, the header is already decrypted by the header decryption (see section 3.3.5).

Copying the keys

This step is done analogous to section 3.3.5, with the exception that only the pp_key and IV belonging to the detected key phase are copied.

Note that the IV is not directly the input to the nonce/IV of the AES-GCM module but rather the XOR of the IV and the packet number. The packet number is parsed from the incoming packet by saving the packet number length in a register when it arrives in the module and copying that amount of bytes after the last DCID bit when it arrives.

Input stage

The input stage is built as a state machine to keep track of which type of data is being fed to the AES-GCM module. A flowchart for this state machine is shown in figure 3.8. Count4 denotes the value of a 2-bit counter that keeps track of the clock cycle the state machine is in (as a multiple of 4) since the *START* state was entered. As long as no QUIC packet is detected, the input is in the *WAIT* state. When the first QUIC byte arrives, the input enters the *START* state where the *Start* (S) command is sent to the AES-GCM module. After 4 clock cycles, the *AUTHENTICATE* state is entered where the additional authenticated data (QUIC header bytes) is sent to the AES-GCM module in blocks of 128 bits. When all header bytes were added as input, the *DECRYPT* state is entered. The payload is now fed to the AES-GCM module in blocks of 128 bits until the first QUIC tag byte arrives. After that, the *FINISHED* state is entered and the concatenation of the header length and payload length is the input of the AES-GCM module. These lengths are parsed from the UDP header length field in combination with the fixed DCID length and the parsed packet number length (PNR_len). Equations 3.1 and 3.2 are used to calculate the length of associated data (A) and ciphertext (C) respectively in bits.

$$len(A) = (21 << 3) + (PNR\_len << 3) \tag{3.1}$$

$$len(C) = (UDP\_length << 3) - (21 << 3) - (PNR\_len << 3) - (16 << 3) \tag{3.2}$$

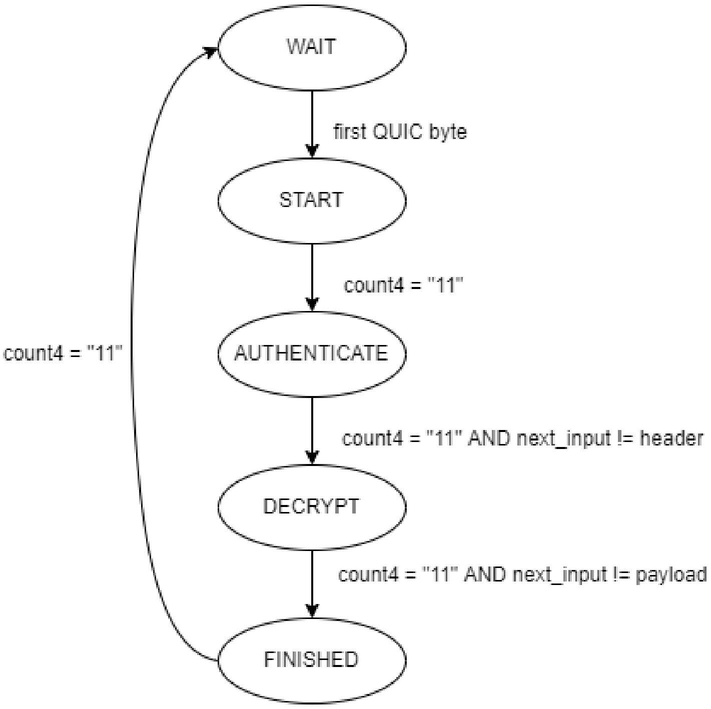
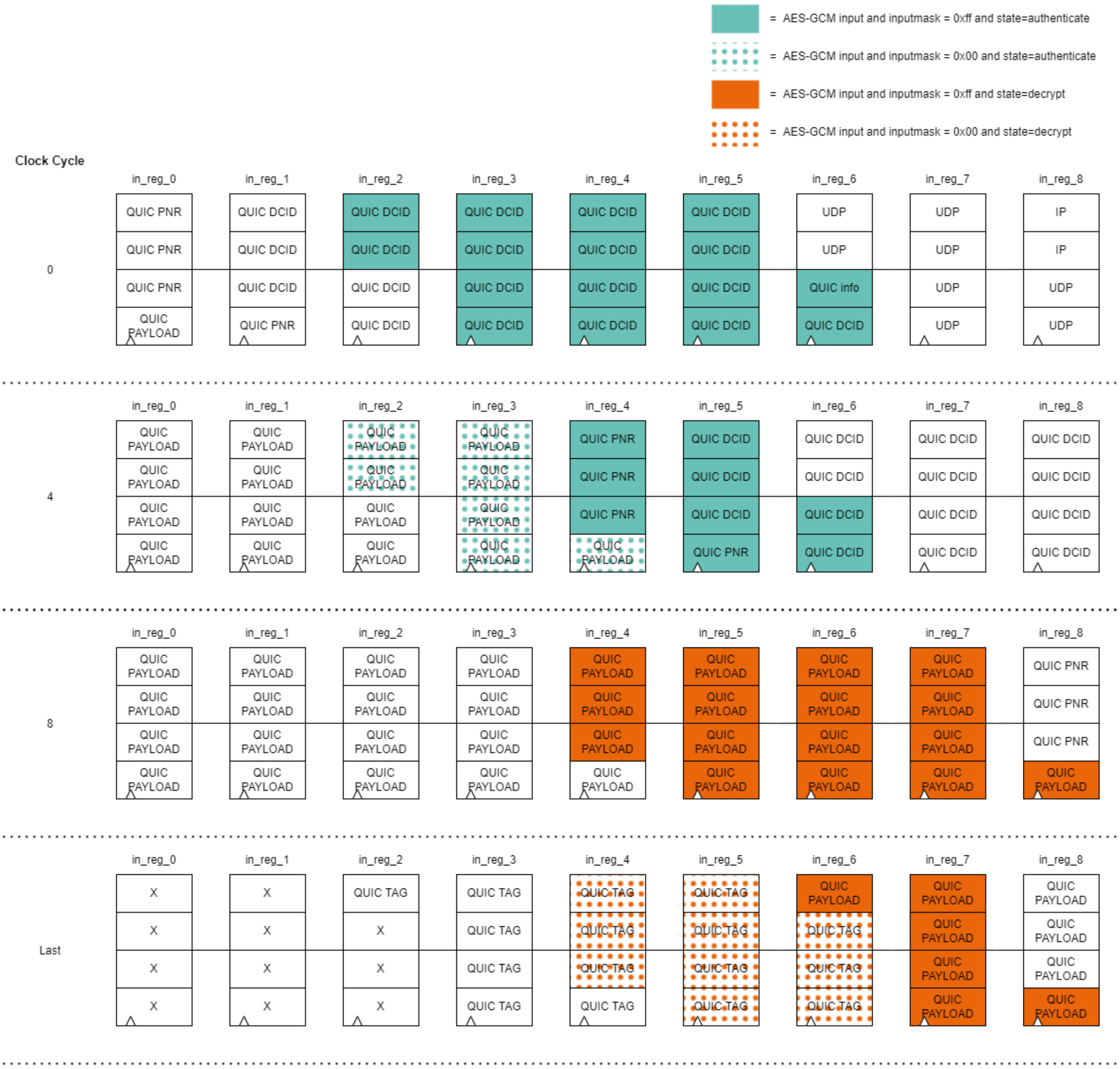

WAIT
first QUIC byte
START
count4 = "11"
count4 = "11"
AUTHENTICATE
count4 = "11" AND next_input != header
DECRYPT
count4 = "11" AND next_input != payload
FINISHED

Figure 3.8: Flowchart for the payload protection module input stage

The input to the AEAD_AES_128_GCM function is 128 bits (16 bytes) wide. This is the amount of bits that have to be available in the module at a minimum. This means it takes 4 clock cycles to shift in enough data with 32 bit registers. The fastest speed at which the AES-GCM module can take this input is once every 4 clock cycles. With this in mind, and the fact that the QUIC header for 1-RTT packets with 20 byte DCIDs varies between 22 and 25 bytes in length, the header can be added as additional authenticated data in 2 blocks. However, the second block will only be between 6 and 9 bytes, so the input mask is used to indicate that the remaining bytes are padding. As a consequence of the 6 to 9 byte input and maximum input rate of the AES-GCM module, the first payload input for decryption will have shifted two registers further. Finally, when the QUIC tag arrives, the last input bytes of the payload are sent to the AES-GCM module. However, because the last payload input can be 1 to 16 bytes, the remaining bytes (which are tag bytes) are again masked with zeros. The input selection is shown in figure 3.9.



🟩 = AES-GCM input and inputmask = 0xff and state=authenticate

▩ = AES-GCM input and inputmask = 0x00 and state=authenticate

🟧 = AES-GCM input and inputmask = 0xff and state=decrypt

▨ = AES-GCM input and inputmask = 0x00 and state=decrypt

Clock Cycle	in_reg_0	in_reg_1	in_reg_2	in_reg_3	in_reg_4	in_reg_5	in_reg_6	in_reg_7	in_reg_8
0	QUIC PNR	QUIC DCID	🟩 QUIC DCID	🟩 QUIC DCID	🟩 QUIC DCID	🟩 QUIC DCID	UDP	UDP	IP
	QUIC PNR	QUIC DCID	🟩 QUIC DCID	🟩 QUIC DCID	🟩 QUIC DCID	🟩 QUIC DCID	UDP	UDP	IP
	QUIC PNR	QUIC DCID	QUIC DCID	🟩 QUIC DCID	🟩 QUIC DCID	🟩 QUIC DCID	🟩 QUIC info	UDP	UDP
	QUIC PAYLOAD	QUIC PNR	QUIC DCID	🟩 QUIC DCID	🟩 QUIC DCID	🟩 QUIC DCID	🟩 QUIC DCID	UDP	UDP
4	QUIC PAYLOAD	QUIC PAYLOAD	▩ QUIC PAYLOAD	▩ QUIC PAYLOAD	🟩 QUIC PNR	🟩 QUIC DCID	QUIC DCID	QUIC DCID	QUIC DCID
	QUIC PAYLOAD	QUIC PAYLOAD	▩ QUIC PAYLOAD	▩ QUIC PAYLOAD	🟩 QUIC PNR	🟩 QUIC DCID	QUIC DCID	QUIC DCID	QUIC DCID
	QUIC PAYLOAD	QUIC PAYLOAD	QUIC PAYLOAD	▩ QUIC PAYLOAD	🟩 QUIC PNR	🟩 QUIC DCID	🟩 QUIC DCID	QUIC DCID	QUIC DCID
	QUIC PAYLOAD	QUIC PAYLOAD	QUIC PAYLOAD	▩ QUIC PAYLOAD	▩ QUIC PAYLOAD	🟩 QUIC PNR	🟩 QUIC DCID	QUIC DCID	QUIC DCID
8	QUIC PAYLOAD	QUIC PAYLOAD	QUIC PAYLOAD	QUIC PAYLOAD	🟧 QUIC PAYLOAD	🟧 QUIC PAYLOAD	🟧 QUIC PAYLOAD	🟧 QUIC PAYLOAD	QUIC PNR
	QUIC PAYLOAD	QUIC PAYLOAD	QUIC PAYLOAD	QUIC PAYLOAD	🟧 QUIC PAYLOAD	🟧 QUIC PAYLOAD	🟧 QUIC PAYLOAD	🟧 QUIC PAYLOAD	QUIC PNR
	QUIC PAYLOAD	QUIC PAYLOAD	QUIC PAYLOAD	QUIC PAYLOAD	🟧 QUIC PAYLOAD	🟧 QUIC PAYLOAD	🟧 QUIC PAYLOAD	🟧 QUIC PAYLOAD	QUIC PNR
	QUIC PAYLOAD	QUIC PAYLOAD	QUIC PAYLOAD	QUIC PAYLOAD	QUIC PAYLOAD	🟧 QUIC PAYLOAD	🟧 QUIC PAYLOAD	🟧 QUIC PAYLOAD	🟧 QUIC PAYLOAD
Last	X	X	QUIC TAG	QUIC TAG	▨ QUIC TAG	▨ QUIC TAG	🟧 QUIC PAYLOAD	🟧 QUIC PAYLOAD	QUIC PAYLOAD
	X	X	X	QUIC TAG	▨ QUIC TAG	▨ QUIC TAG	▨ QUIC TAG	🟧 QUIC PAYLOAD	QUIC PAYLOAD
	X	X	X	QUIC TAG	▨ QUIC TAG	▨ QUIC TAG	▨ QUIC TAG	🟧 QUIC PAYLOAD	QUIC PAYLOAD
	X	X	X	QUIC TAG	QUIC TAG	▨ QUIC TAG	▨ QUIC TAG	🟧 QUIC PAYLOAD	🟧 QUIC PAYLOAD

Figure 3.9: Input selection for the to be authenticated and decrypted data in function of the clock cycle from the start of authentication

Decryption stage

Contrary to the header protection module, the payload protection uses a pipelined AES implementation for the AES_128_GCM function. This was necessary to achieve the same speed as the pipeline, at the cost of FPGA resources. If the AES_128_GCM module was made with an iterative AES module, the fastest rate at which data could be processed is 128 bits every 10 clock cycles. With the pipelined AES-GCM, the maximum data rate is 128 bits every 4 clock cycles. This is the same data rate as the rest of the pipeline, which avoids the need to stop the pipeline while decrypting. The pipelined AES_128_GCM module is described in section 3.3.7. In total, the delay of the AES-GCM module is 13 clock cycles. The data and metadata pipelines are delayed with the same amount of clock cycles using registers.

Output stage

The output stage uses a copy of the state machine of the input stage to determine which type of data is passing. The decryption stage is used to replace the encrypted payload data with the decrypted data at the output of the AES-GCM module, using some control logic in combination with a multitude of multiplexers. The finished state is finally used to compare the last 16 bytes of the QUIC packet with the output of the AES-GCM module at that point (which is the calculated tag). If both tags are equal, no further action is taken. If the tags are not the same, a failed authentication counter is increased by 1. The failed authentication counter is also accessible for reading by the AXI4-stream interface.

3.3.7 The AEAD_AES_128_GCM pipelined module

The pipelined AES_128_GCM module is made up of two main stages: the encryption stage and the authentication stage. A pipelined AES module is the workhorse of the encryption stage which takes 9 clock cycles to complete. Parallel to the AES_pipelined module are 9 registers of 128 bits wide to delay the input with that same amount of cycles. Similarly, the command that was given at the input is delayed by 9 cycles using registers. The authentication stage performs the GHASH function which takes 4 cycles to complete. Again, the input data and command, as well as the AES output, are delayed for 4 cycles using registers.

At the start of an AEAD operation, 128 zeroes are encrypted using the AES module with the key to calculate the value for H that will be used by the GHASH module. Next, additional associated data can be fed to the module in blocks of 128 bits using the input and inputmask[1]. When feeding associated data, the AES_pipelined module is not used, the input is delayed for 9 cycles after which it is sent unmodified to the GHASH function. In this case, the output is the unmodified associated data. Following this, the ciphertext can be decypted in blocks of 128 bits, again by using the input and inputmask signals. When decrypting, the 96-bit nonce is concatenated with the value of a 32-bit counter that is initiated with the value '1'. The counter (CTR) is increased by 1 with every authenticated decrypt (AD) command. The input ciphertext is again propagated up to the GHASH, where it is fed to the GHASH function for authentication. When decrypting (or encrypting), the output is the XOR of the output of AES_pipelined module and the input data. Finally, when all data has been decrypted/encrypted, the tag can be calculated by encrypting the concatenation of the nonce and 32 zeroes with the AES_pipelined module and feeding the concatenation of the 64-bit representation of the length of associated data and the length of the ciphertext/plaintext in bits to the input. In that case, the output is the XOR of the outputs of the AES_pipelined and GHASH modules.

The module is controlled using commands. These commands are eplained in Table 3.3. The fastest speed a which commands can be fed to the module is once every 4 clock cycles, with the exception of the first authenticated data (A) which can be fed immediately after the start (S) command. This is due to the GHASH module needing 4 cycles to authenticate a block of data.

[1] The inputmask is used to add padding to an input block when the associated or decrypted data is not a multiple of 128 bits and does this by doing a bitwise AND with the input.

Thus, the speed of the AES_128_GCM module is limited by the GHASH module. If a single cycle implementation of the GHASH function would be available, the data rate can by 128 bits per clock cycle.

Table 3.3: AESGCM_pipelined module commands with explanation

Command	explanation
S	start command (calculates H for the GHASH block)
A	Authenticate additional data (add 128 bits of associated data)
AD	Authenticated decrypt (add 128 bits of data to be decrypted and authenticated)
AE	Authenticated encrypt (add 128 bits of data to be encrypted and authenticated)
F	Finish (add length data and produce the authentication tag)

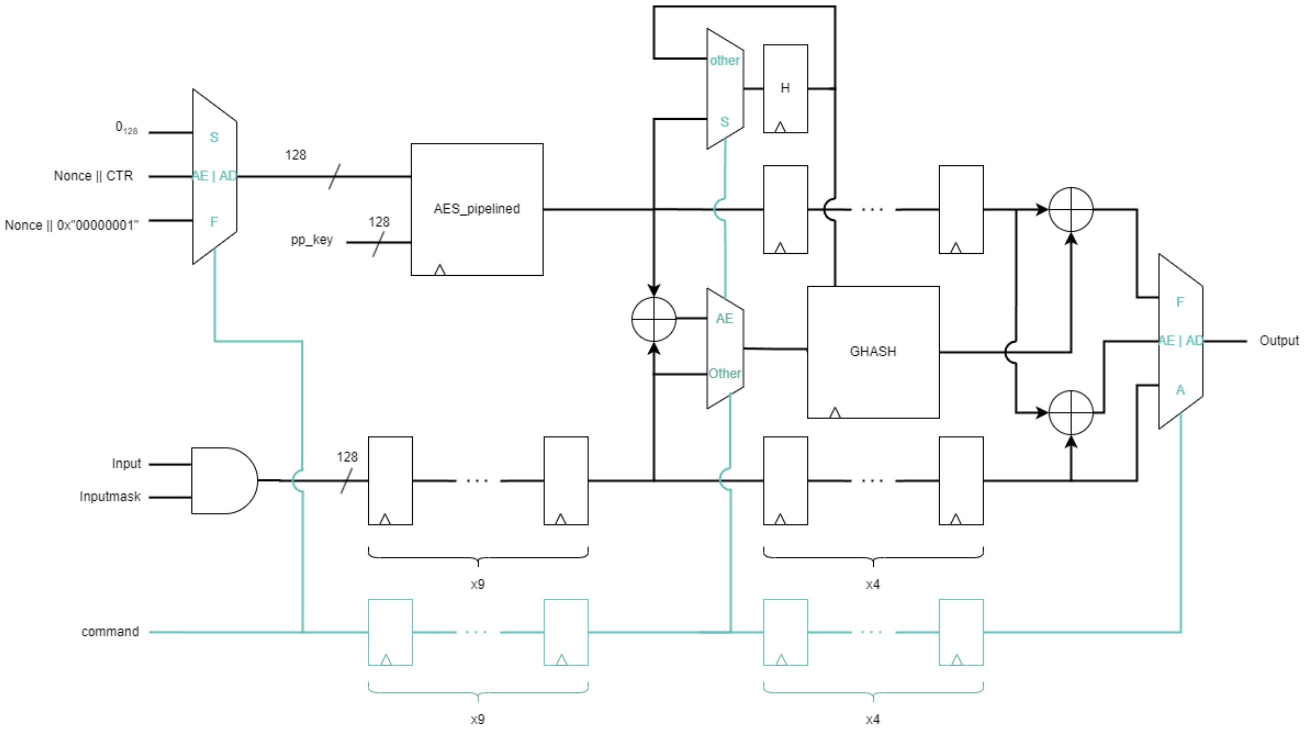


Figure 3.10: Hardware schematic of the AESGCM_pipelined module

AES_pipelined

The pipelined AES module unrolls the AES algorithm into its 10 rounds with registers between each round for the data and roundkeys. More precisely, first the unmodified key is added followed by 9 normal rounds of *subsitute bytes*, *shift rows*, *mix columns* and *add roundkey* followed by a final round without *mix columns*. In each round, the roundkey is calculated using the previous key in a *keyGenerator* step. In hardware, the subBytes, shiftRows and mixColumns modules were provided by ES&S. The normal AES_round modules calculate their roundkey using a round constant, which is hard coded into the modules based on which round they are. A representation of an AES_round module and the final AES_lastround module are shown in Figure 3.11. The hardware schematic of the AES_pipelined module is shown in Figure 3.12.

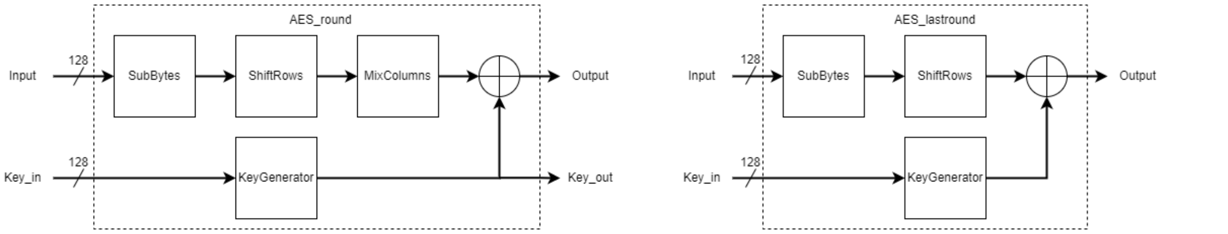


Figure 3.11: Hardware schematic of the AES_round (left) and AES_lastround (right) modules

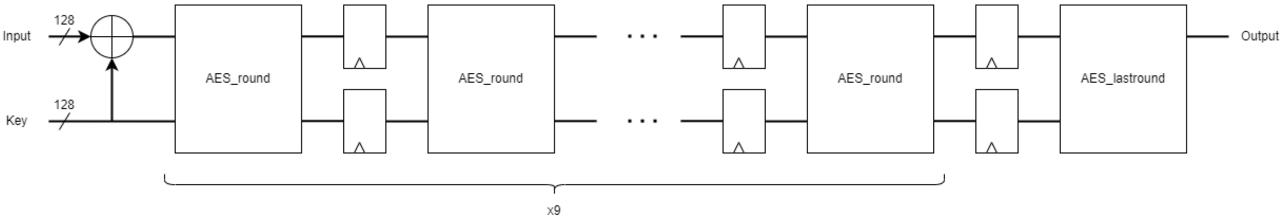


Figure 3.12: Hardware schematic of the AES_pipelined module

GHASH

The GHASH module uses a state machine to identify in which stage of authentication the module is. The state machine starts in the *INIT* state, where the data at the input is loaded into the input register for the MALU (which performs the modular multiplication with H). When a data_valid signal[2] goes high and the chip_enable (ce)[3] is high, the *CALCULATE* state is entered. The MALU takes two cycles to complete its calculation, where the second cycle the input data has to be shifted 64 bits to the left. When the MALU has finished calculating, it pulses a MALU_done signal after which the state machine enters the *WAITING* state, where it holds the output of the MALU in a register. In the *WAITING* state, the next input to the input register is the XOR of the output and input. Thus, after the first block of authenticated data, the state switches between calculating and waiting for as long as there is data to authenticate, and goes to the *INIT* state after data valid goes low.

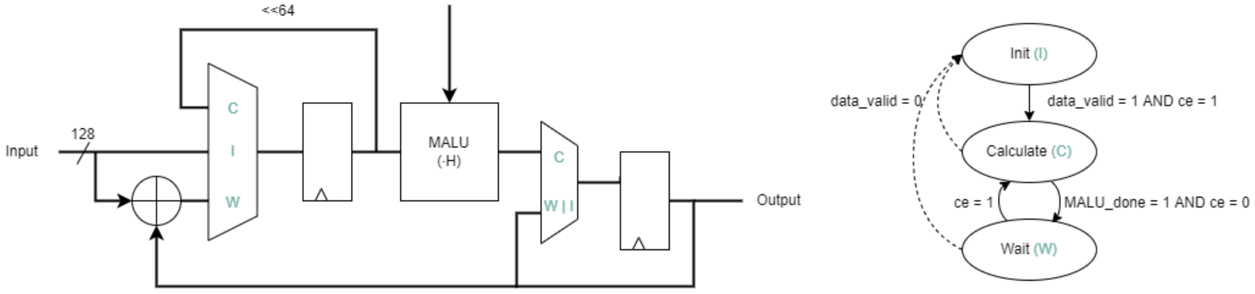


Figure 3.13: Hardware schematic of the GHASH module (left) and its state machine (right)

[2] data_valid in the GHASH module has to stay high as long as there is data to be authenticated, otherwise the authentication state is discarded

[3] chip_enable (ce) in the GHASH module is a pulse that indicates a new block of data to be authenticated has arrived

3.4 Control

3.4.1 AXI4-lite controller

The decryption hardware can only perform its task if the key_memory has keys for a connection and the DCID_lookup module has the corresponding addresses. Consequently, a memory-mapped controller is made with a accessory linux device driver. The AXI4-lite protocol is used for writing to the controller's registers. The PS is connected through a AXI interconnect to the controller. The memory-map of the controller registers is shown in Figure 3.14.

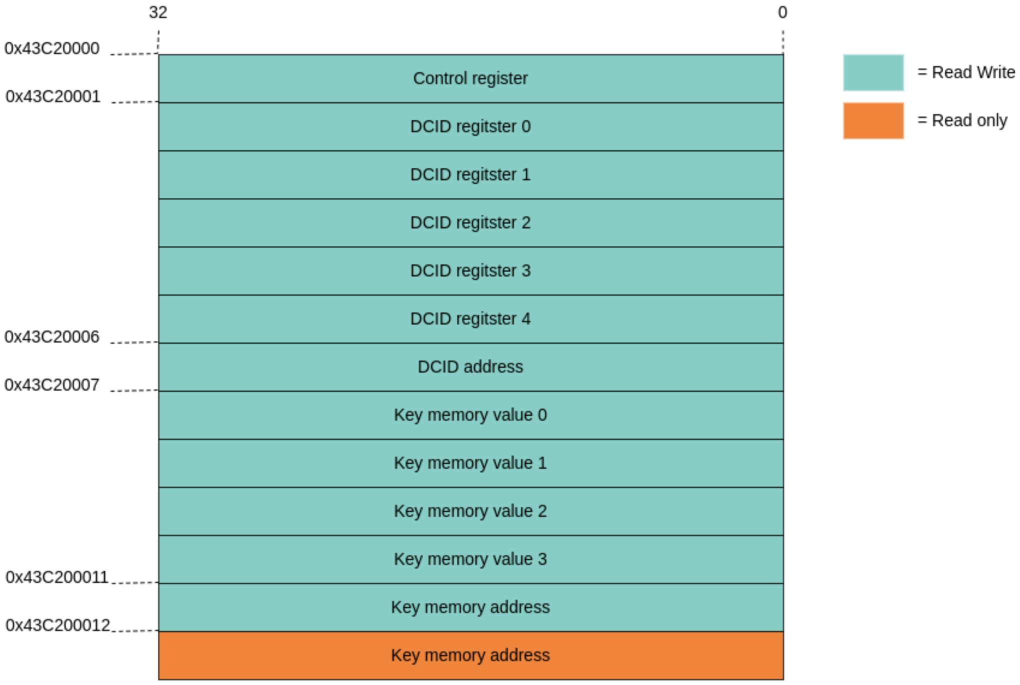


Figure 3.14: Memory map of the controller registers

Usage

When writing connection secrets, the value of the secret is written to *key memory value* 0 to 3, totalling 128 bits. Next, the address in memory, to which the secret has to be written, is set in *key memory address*. Now, to write the secret to the given address, the third least significant bit of the control register CR has to be set by writing 0x4 to the CR.

When writing a connection ID, the value of the DCID has to be written to *DCID register* 0 to 4, totalling 160 bits. Next, the base address for the connection secrets corresponding to that DCID are written to *DCID address*. Now, to complete the transaction, the least significant bit of the CR has to be set by writing 0x1 to the CR.

To invalidate/discard a connection ID, the DCID has to be loaded into the DCID registers and the second bit of the CR has to be set (by writing 0x2 to the CR).

3.4.2 linux kernel driver

The accessory driver is a linux character device driver. Firstly, the physical hardware addresses (see Figure 3.14) are remapped to virtual addresses. *IOCTL* functions are used to set the write state and perform control operations. The IOCTL functions are shown in Table 3.4. Some of the IOCTL functions are used to set the drivers write state. The write state determines which physical registers are written to, when a write to the device file is performed. The write states are explained in Table 3.5.

Table 3.4: IOCTL function of the PoC control driver

IOCTL name	**explanation**
IOCTL_CLEAR_DCID	Discards the DCID written to the registers (CR=0x2)
IOCTL_WRITE_DCID	Writes the DCID written to the registers (CR=0x1)
IOCTL_WRITE_KEY	Writes the secret written to the registers (CR=0x4)
IOCTL_RESET _CONTROLLER	Resets the controller
IOCTL_SET_DCID	Sets the write state to DCID_VALUE_WRITE
IOCTL_SET_DCID _ADDRESS	Sets the write state to DCID_ADDRESS_WRITE
IOCTL_SET_KEY	Sets the write state to KEY_VALUE_WRITE
IOCTL_SET_KEY _ADDRESS	Sets the write state to KEY_ADDRESS_WRITE

Table 3.5: Write states of the PoC controller driver. Determines which registers are written upon a write to the device file

Write state	**explanation**
DCID_VALUE_WRITE	Write to *DCID registers*
DCID_ADDRESS_WRITE	Write to *DCID address* registers
KEY_VALUE_WRITE	Write to *key memory value* registers
KEY_ADDRESS_WRITE	Write to *key memory address* register

Usage

An example python program that writes keys to the PoC with the kernel driver from this section is shown in appendix A.

3.5 Test setup

To verify the design, a test setup uses input and output FIFOs serving as the in- and outputs of the decryption system. A FIFO at the input is filled with packet data and metadata to simulate the AXI4-stream protocol using a test setup linux device driver. The DCIDs and keys are written to their respective modules using the AXI4-stream control block and the proof of concept device driver. Using an enable signal, the packet is passed through the PoC system and the output FIFO is filled with the decrypted packets. In software, the output FIFO is read and the packets are compared with verification packets. A semi-automated test program compares the packets and gives info of where differences are detected. The input packets and verification packets are generated using a python program that makes files with both the plain and encrypted versions of the packets in a binary format. Figure 3.15 shows a block diagram of this test setup.

To measure the performance benefit of the POC design, the decryption speed and CPU resource usage were measured for both the software decryption using the picoquic stack with openSSL, and the hardware decryption using the POC design. The packet size was varied to determine its influence on decryption speed in both cases.

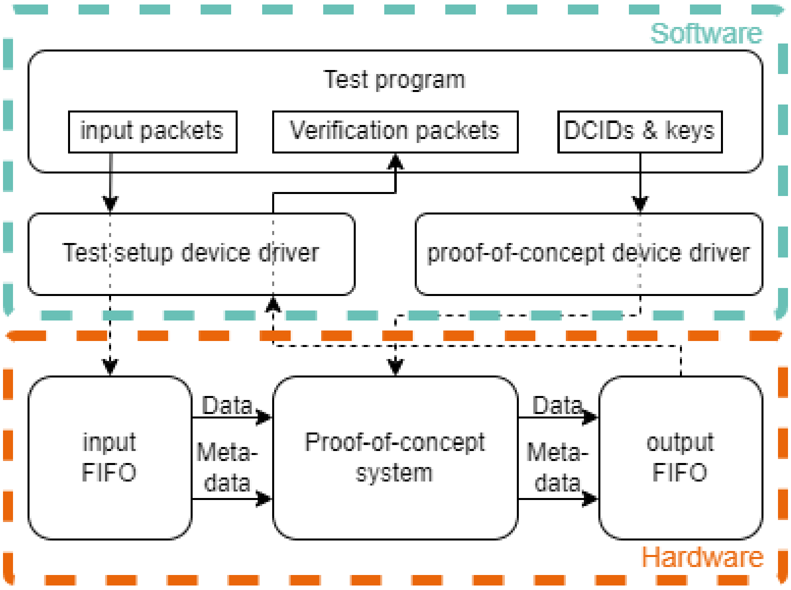


Figure 3.15: Test setup for verifying the functionality of the proof of concept hardware

3.5.1 Platform choice

The hardware from section 3.3 is designed to be used on an FPGA platform. The QUIC stack, with which the hardware should be combined, is run on a processor with an operating system (OS). Thus, the chosen hardware platform must have a processor capable of running a (preferably linux) OS and be able to communicate with hardware on an FPGA. A ZYNQ7000 series FPGA with a dual-core ARM cortex-A9 processor on a PYNQ-Z2 board was chosen based on this requirement and availability.

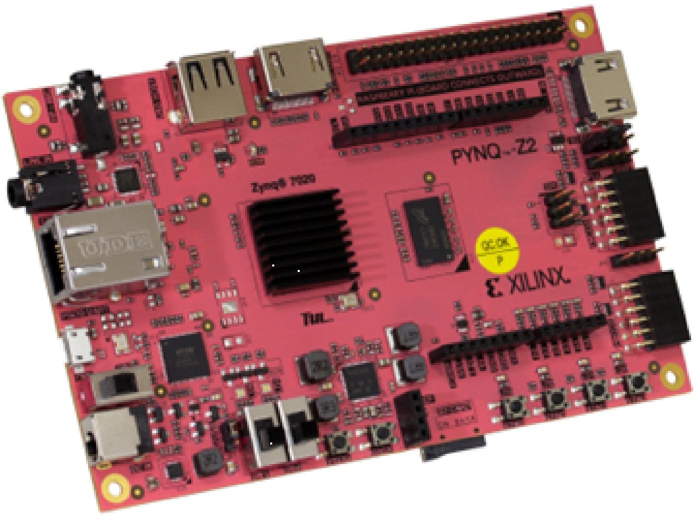


Figure 3.16: PYNQ-Z2 board with a ZYNQ7000 series FPGA. From: [7]

3.5.2 QUIC implementation choice

The choice for a QUIC implementation in the context of this research is guided by the following requirements:

- compliance with the IETF standard
- minimal overhead induced by programming language abstractions
- readability and comprehensibility of the code
- ease of use

An actively updated list of QUIC implementations can be found at [22]. To comply with the requirements, only the section 'IETF QUIC Transport' from the list is taken into account. Minimal programming language induced overhead is achieved by narrowing the list down to the C/C++ implementations. An argument could be made for implementations in the 'Rust' programming language as it is 'low level' as well. However, lack of experience with that language makes it a less suited candidate for this Master's thesis. Picoquic[23] was then chosen for its relative simplicity in comparison to implementations like mvfst[24] by Facebook, Chromium[25] by Google and Quant[26]. Moreover, Picoquic has good integration with picoTLS. PicoTLS is a cryptographic library that is also required to get the implementation running. PicoTLS has already been proven usable on ARM-based systems[27] in combination with Quant.

Chapter 4

Results and discussion

This chapter aims to characterise the PoC design and give some initial performance comparison with software decryption. Section 3.5 explains how the performance tests were performed. Firstly, the timing constraints of the design after synthesis are given. Next, the resource usage is analysed and the raw throughput is calculated. The decryption speed measurements for both hardware and software decryption are listed and compared. Finally, the limitations and possible solutions for them are explained.

4.1 FPGA synthesis results

4.1.1 Timing results

Table 4.1 shows the timing results after synthesizing the total PoC system for the PYNQ-Z2 board with the Vivado design suite. The maximum clock frequency is the frequency at which the PL can run without exceeding the critical path's delay in the PoC design. The ARM processor has a separate clock that runs at a frequency much higher than that but communication can take place due to the AXI4-Lite adapter. The maximum frequency for the PoC is found by iteratively increasing the clock frequency until the worst negative slack (WNS) is as small as possible without becoming negative. This was found at 83.33 MHz.

Table 4.1: Timing results after synthesis of the total PoC for the PYNQ-Z2 board

Frequency	**worst negative slack (WNS)**
76,923080 MHz	0.136 ns
83.333336 MHz	0.054 ns
90.909088 MHz	-0.220 ns

After analysing the timing summary in Vivado it was found that the critical delay path was located in the MALU hardware (in the GHASH module). This was expected as a the calculation involves a very wide (order of 1000's of lines) XOR is performed. Improving this implementation is the first step to increasing the maximum clock frequency of the PoC.

4.1.2 Resource usage

Table 4.2 shows the resource usage of the total PoC and its submodules for the PYNQ-Z2 board. The slice LUT and register usage is the value after running 'optimize design'. The percentage value for the slice LUTs and registers denotes the used fraction of their respective total available resources on the FPGA. Note that all results are platform specific and might be better with more performant hardware.

Table 4.2: Hardware resource usage after synthesis of the total PoC and submodules for the PYNQ-Z2 board

Module	Slice LUTs	Slice registers
Parser	20 (0.04%)	10 (<0.01%)
DCID lookup	1147 (2.16%)	427 (0.40%)
Key memory	3375 (6.34%)	871 (0.82%)
Header protection	1548 (2.91%)	1622 (1.52%)
Payload protection	17146 (32.23%)	4959 (4.66%)
Total	23315 (43.83%)	7995 (7.51%)

The Payload protection module is clearly the largest module. This is a result of the pipelined AES_128_GCM module, which has the pipelined AES module with 10 AES rounds and the two cycle MALU.

4.1.3 Throughput

The raw throughput of the PoC hardware can be calculated by multiplying the maximum clock frequency with the amount of bits that are processed every clock cycle:

$$Throughput_{raw} = f_{max} \cdot data\_width = 83.33\ MHz \cdot 32\ bits = 2.67\ Gbit/s$$

Note that this the maximum raw throughput, which includes headers and does not take into account the speed at which the software can handle the incoming data.

4.1.4 Latency

The latency of the PoC denotes the amount of time a byte takes between the input of the PoC and the output. This can be found by multiplying the amount of cycles it takes to propagate from input to output with the clock period (from the maximum frequency) as shown in equation 4.1. Table 4.3 shows the amount of register stage and their resulting delay.

$$t_{delay} = \#_{cycles} \cdot T_{clock} \tag{4.1}$$

where:

- t_{delay} is the delay in seconds
- $\#_{registers}$ is the amount of cycles of delay a module introduces
- T_{clock} is the clock period

Table 4.3: Delay introduced by the pipeline registers

Module	$\#_{registers}$	t_{delay}**(ns)**
DCID lookup	11	132
Key memory	6	72
Header protection	14	168
Payload protection	32	384
Total	63	756

4.2 Decryption speed

Figure 4.1 shows a the results for the decryption time in function of QUIC packet size for software-only decryption with picoquic and for decryption in hardware with the PoC design. The decryption in hardware can be calculated in nanoseconds with equation 4.2. The software version runs on a OS and the decryption time is thus variable. The software decryption was measured on a HP elitebook.

$$t_{decrypt}(s_{QUIC}) = t_{latency} + \lceil \frac{s_{overhead} + s_{QUIC}}{4} \rceil \cdot T_{clock} \tag{4.2}$$

Where:

- $t_{decrypt}$ is the time needed to decrypt a QUIC packet in ns
- $t_{latency}$ is the latency of the proof of concept design in ns (see section 4.1.4)
- $s_{overhead}$ is the amount of non-QUIC bytes in the packet (Ethernet, IP and UDP headers)
- s_{QUIC} is the amount of QUIC bytes (header and payload)
- T_{clock} is the clock period in ns

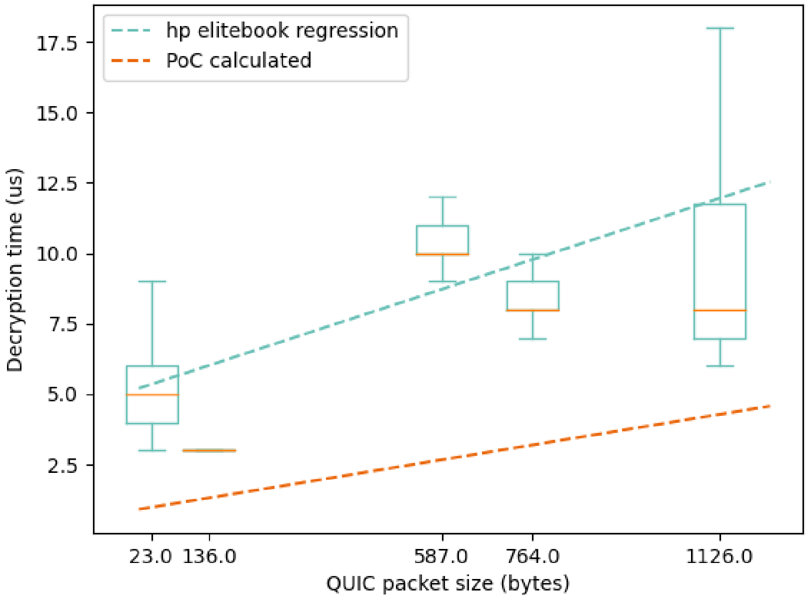


Figure 4.1: Plot of the decryption time needed in function of packet size in software-only and for the PoC

After performing linear regression, the following relation was found between decryption time and QUIC packet size for software-only decryption. Note that $t_{decrypt}$ is in μs:

$$t_{decrypt}(s_{QUIC}) = 0.006\ \mu s \cdot s_{QUIC} + 3.2\ \mu s$$

From equation 4.2 the slope of the decryption time of the PoC hardware can be found as 0.003. From this, we can find that the decryption time in hardware is at least 2 times faster than with picoquic on a hp elitebook laptop over the entire range of packet sizes. However, this is not the main benefit of the hardware decryption. The advantage is that none of the decryption time spent in hardware is performed on the CPU. Essentially, the total amount of CPU resources and time that would normally be spent on decrypting a 1-RTT QUIC packet when it arrives is freed up. These resources can than be spent on other tasks or on increased throughput. Note that the increase in throughput is still dependant on a large amount of other factors such as the user/kernel space data copy.

4.3 Limitations

This section will discuss some of the limitation of the PoC. These are limitations in the design as well as the practical implementation. Where possible, potential solutions are proposed.

4.3.1 Only 1-RTT packets (short header packets)

The PoC is made for 1-RTT packets only. Offloading decryption of this packet type should give the largest benefit as after completion of the handshake, all data is exchanged with this packet type. In most cases this will means that most of the packets will be 1-RTT. However, for smaller requests, the proportion of other packet types will become higher. In that case, also offloading other packet types might become beneficial. However, doing so would require more hardware resources which is a trade-off that has to be considered.

For example, supporting hardware decryption of initial packets upon arrival would require the secrets to be calculated in hardware as well (because the DCID is used in the HKDF function). Thus, the HKDF function and its hashing algorithm would also need to be implemented in hardware. The question remains if the performance gain of doing this is worth the extra hardware resources.

4.3.2 Only 20 byte DCIDs

The PoC only supports 20 byte long DCIDs and uses this as a fixed offset for detecting where the QUIC header stops, etc. However, in a future design it would be desirable to support all DCID lengths and even zero length DCIDs in combination with the 4-tuple. This would require a substantial rework of the design.

4.3.3 Incomplete parser

The parser in the practical implementation of the PoC is currently highly oversimplified. It marks all packets as a Ethernet/IP/UDP/1-RTT-QUIC packet. It is the DCID lookup module that eventually detects if it actually is a valid 1-RTT QUIC packet. Granted, there is a very low probability that a DCID match is found if a packet is not actually of the 1-RTT type.

4.3.4 No packet dropping in hardware

In the PoC, packets are signalled to have failed authentication by replacing the tag at the end of the packet with all zeros. This means the entire packet is still copied to software, where the check for a null tag is done and subsequently, the packet is dropped. This still creates some unnecessary overhead that could even be exploited by an attacker to perform a denial of service attack. Consequently, dropping the packets in hardware would prevent these needless steps in software.

A possible way to do this practically would be to pass the packets to DRAM instead of a simple FIFO at the processing system side. Upon failed authentication, the hardware could then 'fail' to notify the PS that a packet has arrived, preventing the packet to ever reach the software. The authentication failure counter can be used to detect an attack by measuring how fast it is increasing.

4.3.5 Out-of-order packets after key update

As explained in section 3.2, out-of-order packets that arrive with an older key than is available in the lookup table are dropped because their authentication will fail. This will have an impact on the performance of the protocol because retransmissions have to be done. How large of an impact this has will be dependant on how frequent key updates occur in combination with how high the rate of out-of-order packet arrivals is.

A possible solution to this is to save the previous key and IV in hardware for about 1 round trip time and also save the last highest acknowledged packet number for that connection. The hardware can then use the packet number of the arriving packet to check if it has arrived out-of-order and subsequently if it has to use the older key and IV.

4.3.6 Single RAM block for keys and IVs

By using a single 128-bit wide RAM block for the all the secrets, some space is wasted because the IV is just 96 bits wide and the remaining 32 bits are filled with zeros. A better solution would be to implement 5 different RAM block for each secret (hp_key, pp_key0, iv_0, pp_key1, iv_1).

4.3.7 Fully separated modules

As explained in section 3.3, the practical design was split up into sections for easier development. This is at the cost of hardware resources and latency because there is no overlap in the modules. For example, the header protection module has 5 register stages before the AES-ECB module to gather the input for the AES-ECB module. These 5 registers could be taken from the previous module (key lookup). This could be done similarly in the payload protection module. Subsequently, less registers would be used and the latency would be reduced.

Chapter 5

Conclusion

The proof of concept design shows it is possible to decrypt QUIC packets in hardware without needing extra data copy between user space and kernel space. This is done by placing the decryption system directly after the Ethernet interface and decrypting the packet before it is accessed by the CPU. A linux kernel driver was created to write connection information, such as secrets, to the PoC using an AXI4 interface.

The PoC hardware decryption frees up CPU time and resources which can be used on other task or increasing throughput. The maximum raw data throughput of the PoC is 2.67 Gbit/s. The initial design makes a number of simplifications such as; only decrypting 1-RTT QUIC packets, only allowing 20-byte DCIDs and only supporting the AEAD_AES_128_GCM cipher suite. Moreover, there is no integration with a QUIC implementation yet. Instead, an artificial test setup was used.

The decryption speed of the PoC hardware was measured and compared with software-only decryption (picoquic) on a HP elitebook laptop. This was done for a multitude of QUIC packet sizes ranging from 23 bytes to 1126 bytes. Over the entire range of packet sizes, the hardware decryption is at least 2 times faster than software decryption. However, this number is heavily dependant on the other tasks running on the OS. Furthermore, the used CPU as well as FPGA have an effect on the performance of both software decryption and hardware decryption respectively. Furthermore, a number of improvements could be made to increase the performance of the hardware decryption such as decreasing the critical delay path which increases the maximum clock frequency.

In future work, a comparison should be made with server grade hardware to give a more realistic view of the performance characteristics. Furthermore, the hardware should be fully integrated with a QUIC software stack and the functionality should be expanded to perform more thorough measurements such as maximum QUIC data throughput, network latency and other parameters, while including overheads such as data copy.

Bibliography

[1] R. Marx, "Http/3 from a to z: Core concepts (part 1)," 2021. [Online]. Available: https://www.smashingmagazine.com/2021/08/http3-core-concepts-part1/

[2] "Internet Protocol," RFC 791, Sep. 1981. [Online]. Available: https://www.rfc-editor.org/info/rfc791

[3] B. Hinden and D. S. E. Deering, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460, Dec. 1998. [Online]. Available: https://www.rfc-editor.org/info/rfc2460

[4] "User Datagram Protocol," RFC 768, Aug. 1980. [Online]. Available: https://www.rfc-editor.org/info/rfc768

[5] M. Dworkin, *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*, 2007. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf

[6] X. Yang, L. Eggert, J. Ott, S. Uhlig, Z. Sun, and G. Antichi, "Making quic quicker with nic offload," 08 2020, pp. 21–27.

[7] Tulembedded.com. (2022) Products - pynq-z2". Accessed Jul. 15, 2022). [Online]. Available: https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html

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

[9] M. Thomson and S. Turner, "Using TLS to Secure QUIC," RFC 9001, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9001

[10] "quic as a solution to protocol ossification," 2018. [Online]. Available: https://lwn.net/Articles/745590/

[11] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. B. Krasic, C. Shi, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. C. Dorfman, J. Roskind, J. Kulik, P. G. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, and W.-T. Chang, "The quic transport protocol: Design and internet-scale deployment," 2017.

[12] "Transmission Control Protocol," RFC 793, Sep. 1981. [Online]. Available: https://www.rfc-editor.org/info/rfc793

[13] A. Ghedini, "The road to quic," Jul 2018. [Online]. Available: https://blog.cloudflare.com/the-road-to-quic/

[14] D. H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)," RFC 5869, May 2010. [Online]. Available: https://www.rfc-editor.org/info/rfc5869

[15] "Ieee standard for ethernet," *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, pp. 1–5600, 2018.

[16] D. McGrew, "An Interface and Algorithms for Authenticated Encryption," RFC 5116, Jan. 2008. [Online]. Available: https://www.rfc-editor.org/info/rfc5116

[17] J. Hay, M. Machnikowski, G. Bowers, N. Wochtman, J. Muniak, and M. Deval, "Accelerating quic via hardware offloads through a socket interface," pp. 1–6, 2019.

[18] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-transport-11, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-transport/11/

[19] J. Daemen, S. Hoffert, M. Peeters, G. Assche, and R. Keer, *Xoodoo cookbook*, 2019. [Online]. Available: https://eprint.iacr.org/2018/767.pdf

[20] L. Deferme, J. Vliegen, and R. Marx, "Hardware offload van decryptie in het quic-protocol," 2022.

[21] J. Butaye, J. Vliegen, and R. Marx, "Optimaliseren van een quic-implementatie door middel van hardware offloading," 2022.

[22] quicwg, "Implementations," 2022. [Online]. Available: https://github.com/quicwg/base-drafts/wiki/Implementations

[23] private octopus, "Github - private-octopus/picoquic: Minimal implementation of the quic protocol," 2022. [Online]. Available: https://github.com/private-octopus/picoquic

[24] facebookincubator, "Github - facebookincubator/mvfst: An implementation of the quic transport protocol." Aug 2022. [Online]. Available: https://github.com/facebookincubator/mvfst

[25] "Quic, a multiplexed transport over udp," 2017. [Online]. Available: https://www.chromium.org/quic/

[26] NTAP, "Github - ntap/quant: Quic implementation for posix and iot platforms," 2016. [Online]. Available: https://github.com/NTAP/quant

[27] L. Eggert, "Towards securing the internet of things with quic," EasyChair Preprint no. 2434, EasyChair, 2020.

List of appendices

Appendix A

Python program for writing keys to hardware

```
from sys import argv
from os import path
from fcntl import ioctl

# conversion of C ioctl's to python ioctl's
IOCTL_SET_DCID = 101 << (4*2) | 0x0
IOCTL_SET_DCID_ADDRESS = 101 << (4*2) | 0x1
IOCTL_SET_KEY = 101 << (4*2) | 0x2
IOCTL_SET_KEY_ADDRESS = 101 << (4*2) | 0x3
IOCTL_CLEAR_DCID = 101 << (4*2) | 0x4
IOCTL_WRITE_DCID = 101 << (4*2) | 0x5
IOCTL_WRITE_KEY = 101 << (4*2) | 0x6
IOCTL_RESET_CONTROLLER = 101 << (4*2) | 0x7

try:
    filename_dcids = argv[1]
    filename_keys = argv[2]
    filename_out = argv[3]
except (IndexError, ValueError):
    quit()

#read dcid's
with open(filename_dcids) as fd_dcids:
    dcid_inlines = fd_dcids.readlines()
dcid_inlines = [line.rstrip('\n') for line in dcid_inlines]

#read keys
with open(filename_keys) as fd_keys:
    key_inlines = fd_keys.readlines()
key_inlines = [line.rstrip('\n') for line in key_inlines]

for x in range(len(dcid_inlines)):
    dcid_line = dcid_inlines[x]
    dcid_address = [[] for i in range(4)]
    dcid_address[0] = 0
    dcid_address[1] = 0
    dcid_address[2] = int(dcid_line[:2], 2)
    dcid_address[3] = int(dcid_line[2:10], 2)
    dcid = [[] for i in range(20)]
    for i in range(20):
        dcid[i] = int(dcid_line[10+i*8:18+i*8], 2)

    driver_fd = open(filename_out, 'wb')
    ioctl(driver_fd, IOCTL_SET_DCID)
    driver_fd.write(bytearray(dcid[::-1]))
    ioctl(driver_fd, IOCTL_SET_DCID_ADDRESS)
    driver_fd.write(bytearray(dcid_address[::-1]))
    ioctl(driver_fd, IOCTL_WRITE_DCID)
    driver_fd.close()

for y in range(len(key_inlines)):
    key_line = key_inlines[y]
    key_address = [[] for i in range(4)]
    key_address[0] = 0
    key_address[1] = 0
    key_address[2] = int(key_line[:2], 2)
    key_address[3] = int(key_line[2:10], 2)
    key = [[] for i in range(16)]
    for i in range(16):
        key[i] = int(key_line[10+i*8:18+i*8], 2)

    driver_fd = open(filename_out, 'wb')
    ioctl(driver_fd, IOCTL_SET_KEY_ADDRESS)
    driver_fd.write(bytearray(key_address[::-1]))
    ioctl(driver_fd, IOCTL_SET_KEY)
    driver_fd.write(bytearray(key[::-1]))
    ioctl(driver_fd, IOCTL_WRITE_KEY)
    driver_fd.close()
```

Listing A.1: Example python program for writing keys and DCIDs to hardware from an input file in a test setup