PSP over QUIC: Securing Internet Traffic While Supporting Hardware Offloading

Dermott Boylan

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2024

Abstract

QUIC and PSP are both new protocols to be used instead of existing protocols (TCP and TLS respectively). QUIC was created as a lightweight alternative to TCP with a faster handshake and greater support for more options such as sending unreliable datagrams alongside reliable data. PSP on the other hand was created to be able to secure a large number of connections in intra and inter data centre connections, while supporting hardware offloading and not requiring a receiver to store all data in memory, allowing a larger number of supported connections. This project looks to implement PSP within QUIC in order to provide a fast, reliable, secure connection that supports hardware offloading. Since QUIC currently demands encryption using TLS from the standard defined by the IETF, PSP was added to early packets that were not encrypted to avoid encrypting the same data twice. In doing so, I found a roughly 20% overhead for creating a PSP/QUIC packet compared to an ordinary packet. However, the time to handle a PSP packet is quite small, with the largest average decryption time for a PSP packet being roughly 1ms.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Dermott Boylan*)

Acknowledgements

First, I would like to thank my supervisor, Professor Michio Honda, for giving me support with PSP and helping me gain a greater understanding.

I would also like to thank my friends, particularly those who have also been in Appleton Tower at all hours of the day, providing a sense of community that has made my time at university a much better experience, and my flatmates, who have helped me more times than I can count by allowing me to ramble about issues with code in the kitchen.

I could not have finished this without the support of my partner, Nicole, whose love, care, and encouragement has given me the strength needed to continue.

Last but not least, I would like to thank my family, who have shown me nothing but love and support throughout my time at university, and who I know have always had me in their hearts, even when they had no clue what I was talking about.

Table of Contents

Chapter 1

Introduction

The nature of the internet has led to competing standards for communication across the layers of the OSI model. For the transport layer, UDP and TCP have emerged as the main protocols used - UDP for fast communication where reliability is not a necessity, and TCP for when all packets are required to reach their destination. Therefore, whenever a developer is deciding on the protocol used they must decide between speed and reliability. Because of this, a new protocol has arisen, QUIC.

1.1 Transport Protocols

QUIC is a protocol made to be both lightweight and reliable, built on top of UDP while also implementing connections and mechanisms to resend lost packets. It implements a handshake to initiate a connection that is combined with the TLS handshake in order to reduce the overhead of initiating a connection.

QUIC has rapidly taken off, accounting for over half of the connections between clients and Google's servers [4], and is implemented in Microsoft Edge [13], Firefox [8], and Safari [6].

1.2 Security Protocols

Similarly, there have been many enhancements in encryption protocols used. IPsec was developed in the 1990s as a way to encrypt traffic at the internet protocol layer [12]. Nowadays, IPsec is mainly used for VPNs since it only encrypts traffic between private networks and not within, allowing devices on those private networks to observe information that should only be known by the two devices the connection is between.

Shortly after, Secure Socket Layer (SSL) [17] was developed as a way to have privacy and security for communication in a client server model. SSL required that servers authenticated themselves with certificates that also contained their private keys, allowing key negotiation to be completed securely. Unfortunately, due to multiple major security concerns, SSL was eventually depreciated and replaced with Transport Layer Security (TLS), the most popular encryption protocol used today by a wide variety of applications

[21]. TLS is still being developed for, with the latest standard, TLS 1.3, being released in August 2018.

1.3 Data Centre Transmission

As the internet has grown, the amount of data that needs to be stored and transported has also grown. Much of this is stored in data centres, with 1.1 zettabytes ($1.1e21$ bytes) of data stored there in 2017. With data constantly needing to be moved and updated, data centre providers require their data to be moved in a manner that is reliable, efficient, secure, and makes full use of all available hardware.

While previously TCP was used to ensure reliability, QUIC now has better performance in data centre networks than TCP [35]. In addition, Network Interface Cards (NIC) have advanced to the point where data centres CPUs can now offload a variety of tasks on to new SmartNICs [31].

While RFC 9001 [30] outlines how TLS should be implemented with QUIC in a way that combines the QUIC and TLS handshake, decreasing the network traffic required to establish a secure connection, TLS does not have support for hardware offloading, and so this cannot take advantage of the computing power of SmartNICs. While IPsec does support hardware offloading, it also stores the entire encryption state, meaning that the storage required to store all connections would be far too much to be worth it.

1.4 Motivation

With all this in mind, Google have developed the PSP Security Protocol (PSP), a transport-independent, hardware offloading friendly encryption protocol that allows one-way per-connection security [32]. When it was made open-source, the offloading has saved 0.5% of Google's processing power.

Therefore, I have decided to create an implementation of PSP within QUIC to combine the increase in speed that both provide. While unfortunately I am unable to test PSP's hardware offloading due to the lack of a SmartNIC, PSP supports a software implementation called SoftPSP to support traffic between new and legacy equipment that does not have a SmartNIC.

1.5 Project Outline

The project has the following structure:

- Background and Relevant Work: This chapter will cover TCP and TLS, the protocols that QUIC and PSP are aiming to replace.
- QUIC and PSP Protocols: This chapter will examine the architecture of the QUIC and PSP protocols and how they can work together.

- Implementation: This chapter will look at specifics of how I implemented PSP into QUIC.
- Methodology: This chapter examines how PSP/QUIC was tested.
- Results: This chapter looks at the data generated.
- Evaluation and Conclusion: This chapter goes into details of the implications of the results and future work that can be done with PSP/QUIC.

1.6 Project Aims

The aims of this project are as follows:

- Implement PSP in QUIC
- Test PSP to calculate the overhead compared to an ordinary QUIC packet
- Evaluate PSP/QUIC

I will evaluate PSP by calculating the overhead it creates for QUIC packets, and the time taken to create PSP packets, and also to process QUIC packets.

1.7 Results

PSP was found to add a 20.67% overhead to a QUIC packet, with the distribution of packet creation time shown in figure 1.1. Further, we can see the time it takes to create any PSP packet in figure 1.2.

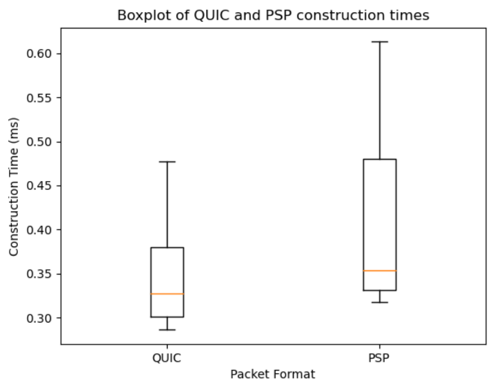

Figure 1.1: The distribution of time for packet construction of PSP and QUIC packets.

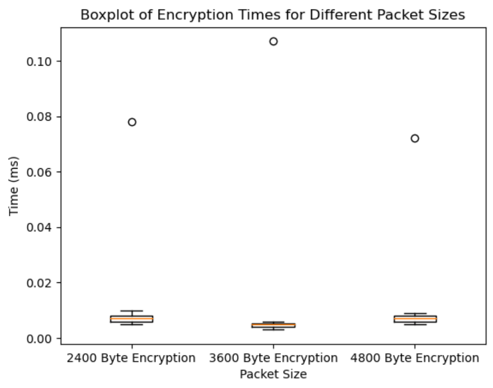

Figure 1.2: The distribution of time to create a PSP packet based on packet size.

Chapter 2

Background and Relevant Work

This chapter will go into detail about TCP and SSL/TLS, the protocols that QUIC and PSP are aiming to replace in certain contexts. Additionally, I will explore relevant works around the transport and encryption of data in environments similar to data centres.

2.1 TCP

TCP is the main protocol used on the internet to provide reliable transmission [24]. TCP has multiple mechanisms to ensure that traffic is received, packets are dealt with sequentially, and to handle congestion control.

2.1.1 Handshake

Whenever a TCP connection is established, both endpoints must complete what is called the TCP handshake [10]. The handshake is made up of three parts, and is initiated by the client in a client-server model.

The client first sends a synchronisation packet (SYN), letting the server know it wishes to create a connection, and the beginning sequence number that the client wishes to use. For the acknowledgement field (ACK), it uses 0 to allow the server to decide.

The server, upon receipt, generates a combined synchronisation and acknowledgement packet (SYN+ACK) to send back to the client. In this packet, the server generates its own starting sequence number to use for the connection, and for the ACK field increments the client's sequence number by 1.

Finally, the client sends an ACK packet to confirm the set up of the connection, allowing data to be passed between the two. A diagram of the handshake can be seen in figure 2.1.

2.1.2 Acknowledgments (Again)

Once a connection has been established, TCP makes sure that data has been received through the use of acknowledgement packets, known as ACKs. When a sender receives

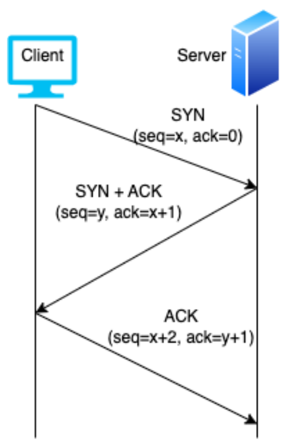

Figure 2.1: The TCP three-way handshake

an ACK, it knows that the packet corresponding to the ACK number has been received. A TCP connection will only send a select amount of data at a time, corresponding to a window size defined by the receiver [3]. When a TCP receiver receives a series of packets, it sends what is called a cumulative ACK, which tells the sender the last packet it received in order. Using this, the sender can now only retransmit packets that are higher than the last received ACK. Similarly, if a receiver receives a packet out of order, it sends an ACK for the last in order packet received. This allows the sender to know that at least one packet has been lost, therefore allowing it to resend all sent packets after the last acknowledged one.

To reduce the overall traffic on the network, the client will only instantly send ACKs for every other packet it receives. It will then only ACK the first packet if it does not receive the second within a timeout that is no greater than half a second.

2.1.3 Window

In order to reduce congestion of the network, the client advertises to the server the amount of data it can handle at any time. This is known as the window. The server can then only send packets that total the size of the window. The window can then be moved once it receives an ACK for a packet within the window [3].

2.1.4 Ending Connection

When an endpoint decides it wants to close the connection, it sends a finishing packet (FIN). The other endpoint, upon receipt, acknowledges the FIN packet, and sends its own FIN packet. Finally, the original endpoint sends an ACK, closing the connection on both ends. This three part end of the connection allows both endpoints to close, even if the last packet is lost [3]. This end phase of the connection can be seen in figure 2.2.

2.1.5 Implementations

Nowadays, TCP is implemented in the operating system kernel such as in Linux [7] and in Windows through Winsock [29].

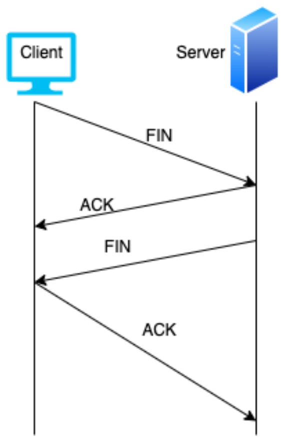

Figure 2.2: The closure of a TCP connection.

2.2 SSL/TLS

Secure Socket Layer and Transport Layer Security (SSL/TLS) have been the main protocols used to secure web traffic since they were developed. The development of the protocol has focused on vulnerabilities found in previous versions, with SSL 1.0 not being released, SSL 2.0 instead being the first standard for encrypting traffic [34]. The protocol is stateful, encrypting each connection separately. Therefore, it must be built on top of a reliable connection, using a transport protocol like TCP. TLS 1.3 is the latest version in the series of protocol, improving forward security compared to TLS 1.2, and reducing the time required to perform the handshake [22].

2.2.1 Handshake

The SSL/TLS handshake is used in order to perform key derivation. In order to start a secure connection, TLS makes use of public key infrastructure, with at least one endpoint (the server) required to have a certificate that allows it to both authenticate its traffic, and that has a public key to be used for asymmetrical encryption.

In a TCP connection, once the TCP handshake has taken place, the client sends a 'ClientHello' message, informing the server it wishes to establish a secure connection, and informing the server of the supported versions of TLS, the available cipher suites, and a string of random bytes [5]. In response, the server sends a 'ServerHello', including the agreed upon TLS version and cipher suite (this should be the latest TLS version that both support and the strongest cipher suite that both support), the server's SSL certificate, along with another string of random bytes.

Upon receipt, the client authenticates the server using its certificate. Knowing that the server is who it says it is, the client encrypts another string of random bytes known as the 'premaster key' using the server's public key. Both client and server can now generate a symmetric encryption key using the premaster key, and the previously sent random bytes. This should give both endpoints the same encryption key.

Finally, the client sends a message informing the server that it is finished, encrypted with the generated key. Once the server receives this, it sends its own finished message,

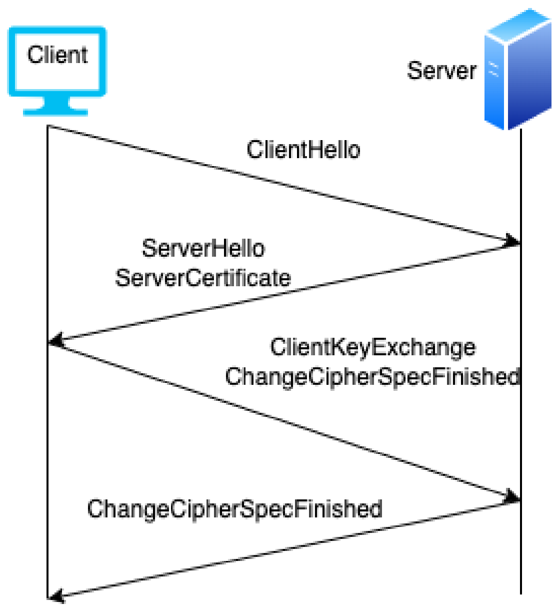

Figure 2.3: The TLS handshake

encrypted with the key. Secure messages can now be sent between the two endpoints. The TLS handshake can be seen in figure 2.3.

2.3 Hardware Offloading

As the data processed in data centres has increased, the Network Interface Card (NIC) has had more and more responsibility, resulting in the rise of the SmartNIC [9]. These are NICs with programmable capabilities, and with more abilities than before. This allows the main CPU to handle fewer processes associated with sending data, freeing them to handle more intensive processes and increasing the efficiency of the data centre.

While there is the concept of TLS offloading [33], instead of offloading encryption onto the SmartNIC, it instead offloads TLS encryption on to a separate server. While this does free up the main server to deal with other processes, it still requires the use of more hardware that could be better used to expand the abilities of the data centre.

2.4 Related Work

In 2023, Liu and Crowley [23] compared the security and performance differences between TCP and QUIC. While QUIC sees diminishing gains when compared to TCP's performance, the authors note that data centres could see gains not seen in smaller contexts. Megyesi et al. 2016 explored QUIC when it was first released and found even then that QUIC was extremely beneficial in areas with high round trip time, a possibility when sending data between data centres across the world.

Pismenny et al. 2016 [28] examines the possibility of TLS offloading, and finds a significant speed increase when offloading TLS to the kernel. Cui et al. similarly find that offloading TLS on to a SmartNIC using the built in crypto accelerator speeds up the encryption drastically, however this requires offloading higher OSI layers onto the SmartNIC, instead of up to the transport layer. Pismenny et al. 2021 [27] explains how NIC offloading is normally reserved for layer 4 and below, and that processes such as TLS which are layer 5 are trickier to offload, requiring a new design in order to work.

Chapter 3

QUIC and PSP Protocols

3.1 QUIC Specification

This section will go into the requirements of a QUIC packet and connection as outlined in RFC 9000 [18].

3.1.1 Streams

QUIC connections send data over streams, which are abstractions used by applications to handle incoming and outgoing data. A stream can be unidirectional or bidirectional, with the two least significant bits used to identify who initiated the connection and whether it is unidirectional or bidirectional. The other 60 bits are unique, starting at the minimum value for each type of connection and increasing consecutively, with any out of order streams created also creating all previous streams as well.

Applications have full control over what streams have priority over others, with QUIC having no control with regards to prioritisation. Instead, QUIC has defined operations that an application can use for streams. For applications sending data, they can:

- Write data to be sent.
- End the stream for a clean end, with a frame sent with the FIN bit.
- Reset the stream for an abrupt end, sending a RESET_STREAM frame if the stream was not already ending.

For an application receiving data, it can:

- Read data sent.
- Abort reading the stream and request the stream to be closed, with the possibility of sending a STOP_SENDING frame.

3.1.2 Stream States

Reliable QUIC data is sent over streams, a connection that can either be unidirectional or bidirectional. When sending, both endpoints use states to know what to expect, and what to do next.

3.1.2.1 Sending

A sending application has multiple states to describe its current and next actions, from which it can move to other states as needed:

- Ready - The endpoint is put in the ”Ready” state after the creation of the stream and before any data has been sent. It stays in this state until it sends its first frame or it sends a RESET_STREAM frame.
- Send - The endpoint is put in the ”Send” state after it sends its first frame until it sends a FIN or a RESET_STREAM frame. After this, it sends or resends data as necessary.
- Data Sent - The endpoint then goes into the ”Data Sent” state once it has sent all frames but hasn’t received an acknowledgement for all of them yet. While in this state the endpoint will retransmit any frames required.
- Reset Sent - The endpoint enters this state if it wants to stop transmitting data early, or if it receives a STOP_SENDING frame from the other endpoint. The endpoint will then send a RESET_STREAM frame and enter this state.
- Data Recvd - The endpoint enters this state when all acknowledgements have been received and it can close the stream.
- Reset Recvd - The endpoint enters this state when the reset has been acknowledged, allowing the endpoint to close the stream.

The state machine for a sending stream is displayed in figure 3.1.

3.1.2.2 Receiving

The receiving application mirrors many of the states that the sending application can be in, however some are not since the receiving application cannot know about them (such as the ”Ready” state).

- Recv - The endpoint enters this state after it receives its first packet or, if it is a bidirectional stream, the sending part enters the ”Ready” state. It stays in this state until it receives a packet with the FIN bit set or a RESET_STREAM frame.
- Size Known - The endpoint enters this state once it receives a packet with the FIN bit sent. At this point, it knows that all data required has been sent, and so it is just waiting for retransmissions.
- Data Recvd - The endpoint enters this state once all data has been received but before it has been processed by the application. If the endpoint receives a

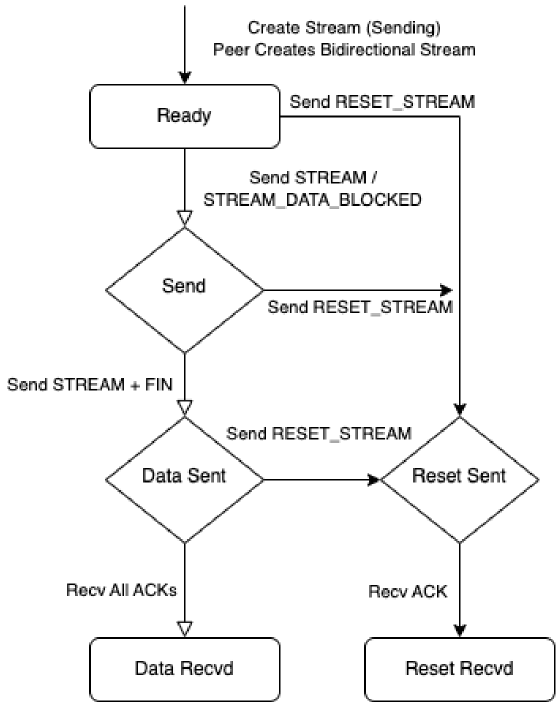

Figure 3.1: The state machine for a sending QUIC stream [18].

RESET_STREAM frame in this state, it is up to the implementation whether to continue processing the data for the application or whether to cancel the process.

- Reset Recvd - The endpoint enters this state if it receives a RESET_STREAM frame. If it was previously in the ”Data Recvd” state the implementation may decide to return to that state instead of moving to the ”Reset Read” state.
- Data Read - The endpoint enters this state once all data has been processed by the application. This is a terminal state.
- Reset Read - The endpoint enters this state once the application has received the notice that the stream was reset. This is a terminal state.

The state machine for a receiving stream is displayed in figure 3.2.

3.1.3 Packet Headers

QUIC packets come in two different forms: long headers are used before symmetric key agreement and version negotiation so that the QUIC header is fixed length, while short headers are used after. In the short header, all the header fields after the reserved bits and before the payload are protected using header protection [30] to protect against replay attacks. The long header fields are described in table 3.1 while the short header fields are described in table 3.2.

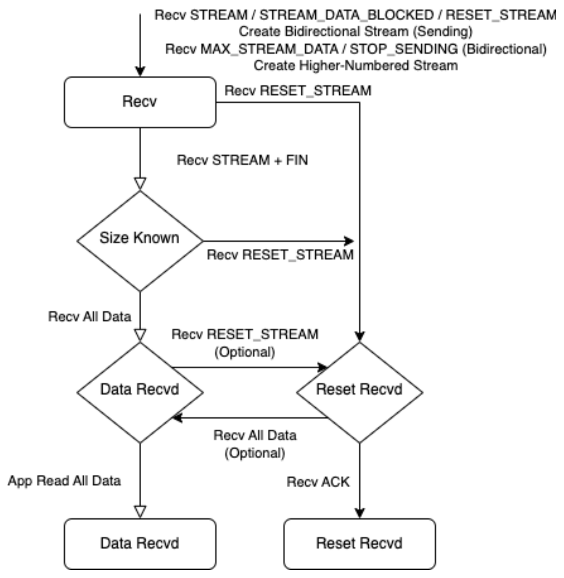

Figure 3.2: The state machine for a receiving QUIC stream [18].

Packet Field	Description
Header Form	Which header form the QUIC packet is using (the most significant bit of the first byte is set to 1 for a long header).
Fixed Bit	The next bit is set to 1 unless the packet is performing Version Negotiation. If a packet has a 0 in this bit then the packet must be discarded.
Long Packet Type	The next two bits describe whether the packet is an initial packet, part of the TLS handshake, the QUIC handshake, or a retry packet.
Type-Specific Bits	The last four bits of the first byte are determined by the type of packet transmitted.
Version	A 32-bit field to indicate the QUIC version being used, which effects how the protocol fields are interpreted.
Destination Connection ID Length	The length in bytes of the Destination Connection ID, the next field in the header.
Destination Connection ID	The destination connection ID, following the length from the previous header field
Source Connection ID Length	The length in bytes of the Source Connection ID, the next field in the header.
Source Connection ID	The source connection ID
Type-Specific Payload	Depending on the type of packet sent, there may be a payload following from the header

Table 3.1: The header fields of a QUIC long header.

Packet Field	Description
Header Form	This bit is set to 0 for a short header.
Fixed Bit	The next bit is set to 1, and packets with a 0 in this field must be discarded.
Spin Bit	The latency spin bit, enabling passive latency monitoring from an observation point on the network. This is an optional field, and must be disabled if an endpoint does not allow it.
Reserved Bits	Two bits that are reserved and given header protection, and after removing protection from both the packet and header must be dropped if non-zero.
Key Phase	Indicates the key phase in order to allow the recipient to tell which packet protection key is being used.
Packet Number Length	The length of the packet number represented as a two-bit unsigned integer. This value is always one less than the length of the packet number field.
Destination Connection ID	The connection ID chosen by the intended recipient.
Packet Number	This value is 1 to 4 bytes long and protected using header protection.
Packet Payload	A 1-RTT packet must include a protected payload.

Table 3.2: The header fields of a QUIC short header

3.2 PSP

PSP uses Security Associations (SAs) as an abstraction for a connection between endpoints. A PSP SA can only be unidirectional, and a bidirectional connection in the transport layer, either through QUIC or TCP, must be represented by at least two separate SAs.

Since PSP was created to handle a large number of connections at once, the cost of storing all SA states on a NIC would be far too great to be worth it. Therefore, the protocol is designed in such a way that the data corresponding to an SA (including the key) can be derived with the data in the packet and a secret stored only in the NIC. This allows support of a far greater number of SAs than a protocol like IPsec.

The protocol also allows for different states for each SA on the hardware for an implementation. While this decreases the number of SAs that can be handled at any one time, the rate that packets are handled at is increased, so this decision is left to the implementation. An implementation can also choose to only lookup the key based on the received packet, however this choice would remove a key advantage of the protocol.

PSP can be used to either encrypt and authenticate a packet, or simply authenticate the packet. This is indicated in the PSP header. PSP authenticates its own header and the payload, and does not authenticate the outer headers. If encryption is enabled, it starts encrypting from the end of the PSP header plus the ”crypt-offset”, which is given in the header.

3.2.1 Architecture

A PSP SA is made up of the following attributes:

- Security Parameters Index (SPI)
- Encryption key (this is stored by the sender and derived by the receiver)
- Crypt Offset (how much of the packet should be encrypted after the initialisation vector (IV))
- Traffic flow confidentiality configuration (this is done the same way as it is done in IPsec [19].)
- Lifetime requirement (after which it must be destroyed)

The method for key exchange is not defined by the PSP protocol. The receiver must create an encryption key and securely send it to the sender. This key is then remembered by the sender, and can be re-derived each time the receiver gets a packet. Each NIC has two master keys which it uses to create the PSP key. At any one time, one of the master keys is ”active”.

An SA creating a new key uses the active key to create its encryption key. Once all SPIs for a key have been used, or after a set period, the other master key takes over. The old key remains until the related connections finish or the SAs’ lifetimes expire, at which point it is replaced and can again become an active key. Since only the receiver is aware of which master key is in use, and the master key is indicated in the SPI, the receiver must decide both the key and the SPI.

A NIC must not use the same SPI until key rotation since the SPI and master key decide the encryption key, and keys should not be used for different SAs. When implemented for a SmartNIC, the SPI must be created by the NIC. PSP's supported encryption algorithms are AES-GCM-128 and AES-GCM-256, the selected algorithm signalled in the PSP version header field.

3.2.2 Headers and Trailer

The PSP header encapsulates an inner transport header such as TCP or UDP, and is then encapsulated by an outer UDP header. The outer UDP header uses a flow hash of the inner header fields as the UDP source port, a destination port of 1000, and a checksum value of 0, or possibly a valid checksum.

The header fields of the PSP header are described in table 3.3.

Header Field	Description
Next Header	The IP protocol number to identify the next header.
Hdr Ext Len	The length of the header in 8-octet units, without the first 8 octets. When non-zero, header extension fields may be present.
R	Reserved bits which are set to zero on transmit.
Crypt Offset	The offset from the end of the Initialisation Vector to the start of the payload. The payload can only be empty if this points to the last octet. A negative value is handled differently depending on the implementation. When authentication, this should be set to zero.
S	Sample at Receiver, used to trigger packet sampling.
D	Drop after sampling, used by packet processing to drop the packet after it has been sampled.
Version	Identifies which version of PSP is being used, which indicates the encryption/authentication algorithm. Values with an unsupported version should be dropped.
V	Virtualisation-Cookie-Present bit, set if and only if the Virtualisation Cookie field is set.
Security Parameter Index	Used to identify the SA the packet belongs to.
Initialisation Vector	A unique value for each packet sent on an SA.
Virtualisation Cookie	Only present if V is set, contains information that is defined by the implementation.

Table 3.3: The header fields of PSP.

The format of the PSP header can be found in figure 3.3.

The PSP trailer contains only the integrity checksum value which authenticates the packet from the start of the PSP header to the end of the payload.

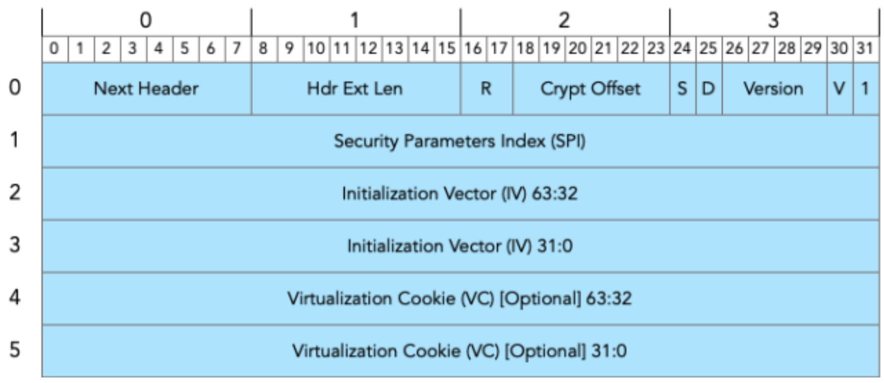
0
1
2
3
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
0
Next Header
Hdr Ext Len
R
Crypt Offset
S
D
Version
V
1
1
Security Parameters Index (SPI)
2
Initialization Vector (IV) 63:32
3
Initialization Vector (IV) 31:0
4
Virtualization Cookie (VC) [Optional] 63:32
5
Virtualization Cookie (VC) [Optional] 31:0

Figure 3.3: The format of a PSP header [2].

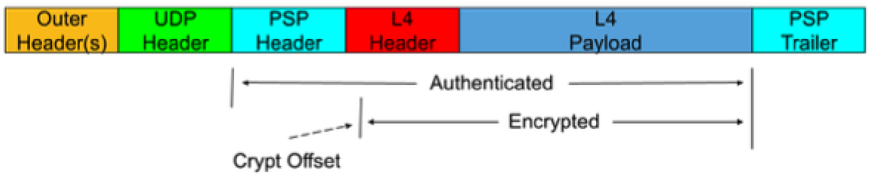

Figure 3.4: A PSP transport packet [2].

3.2.3 Packet Format

PSP packets come in two forms - transport mode and tunnel mode. For the purposes of this project I will solely be examining transport mode. The format of a PSP transport packet is shown in figure 3.4.

Chapter 4

Implementation

This chapter will go into details around my PSP implementation and the chosen QUIC implementation. My implementation can be found here: `https://github.com/DermottB/quiche_psp/tree/master`.

4.1 QUIC Implementation

The QUIC implementation used to test is quiche, created by Cloudflare [14]. The choices considered were quiche and ngtcp2 [25], a QUIC implementation created to work with nghttp3.

Multiple pros and cons for both implementations were taken into account for the decision between the two. While ngtcp2 is written in C, which I am more familiar with than Rust, which is what quiche is written in, quiche benefits from more public use - quiche is used in [16], can be integrated into cURL, and is used for Android's DNS resolver [11].

Additionally, while attempting to work with both, I found that quiche gave better debugging statements, making it easier to work with personally. ngtcp2 was also giving segmentation faults without explanation.

4.2 PSP Implementation

My implementation handles packets after quiche has added the QUIC frame to them but before they are handled by a UNIX kernel's socket abstraction when sending, and the reverse when receiving. This allows for the cryptographic algorithms to be offloaded to a SmartNIC when possible as quiche creates the frame in the application layer.

Each QUIC connection, called an `quiche_conn`, is associated with two separate PSP security associations, one for sending and one for receiving, since security associations are unidirectional [2]. Without access to a Hardware Security Module (HSM) or a SmartNIC to store the keys on RAM, the keys are stored in a struct, however this should not be done in a live deployment of the implementation, which should use more secure storage and follow the encryption key policy of the organisation using the implementation.

4.2.1 Key Exchange

Key exchange is outside the scope of the PSP protocol and is only defined as to be a secure method of transporting a symmetric encryption key. Since both sender and receiver are on the same machine and both have access to the master keys, I assume for a testing environment that key exchange has taken place at the same time as the QUIC handshake similar to TLS. Therefore, when timing the testing, I ignore the handshake packets before 1 round trip, since TLS is using an asymmetric encryption algorithm, which is much slower than symmetric encryption, to agree a symmetric key.

4.2.2 Key Derivation and Handling

In order to take advantage of one of the benefits provided by PSP, I implemented a proper key derivation function that is based on the relevant master key and the SPI of the SA.

This is done through the function `psp_key_derivation`, which accepts an SPI as input, which can be used to find the master key. The master key and SPI can then be used to derive the encryption key using a cipher based message authentication code (CMAC) function. The other input to the CMAC function is made up of 4 32-bit fields that are concatenated together, and are each stored in network-order (big-endian). The fields for the data provided are as follows:

- The counter for which bits of the key are being generated.
- The label for which version of PSP is being used. This can also be constructed by OR'ing the PSP version number in a packet with the hexadecimal bytes 50 76 30 00.
- The context field, which is the SPI, either used during packet generation, or found from the relevant header field when receiving.
- The length field, which represents the length of the key to be generated - 128 bits or 256 bits.

The CMAC function is generated twice to support AES-256-GCM encryption since it only generates 128 bits, using values of 00 00 00 01 and 00 00 00 02 for the counter so as to give different values from the CMAC. These two values are then concatenated together to give the final encryption/decryption key.

4.2.3 Encrypting and Sending

Applications that handle PSP/QUIC packets should use the header files ”quiche.h” and ”quiche_psp.h”, which expose the necessary functions to first create QUIC frames, and to then encrypt using PSP. A QUIC frame should first be added to a packet using quiche's `quiche_conn_send`, before then encrypting the packet using `quiche_psp_encrypt`.

The function first creates a the inner UDP packet and writes it to a new buffer. It then encrypts the packet and writes it to a buffer to be used for the final packet. Afterwards, it creates the PSP header using a struct defined in the same file. Since the implementation is made only for transport mode with QUIC, the next header and version numbers are hard coded, as are the reserved bits since these should be set to zero. The sample and sample drop bits are also set to zero for the purposes of testing the implementation, especially since the implementation is being tested on a virtual network. The virtualisation cookie bit is set to zero since no virtualisation cookie is needed for testing.

The crypt offset of the PSP header is set to zero so that PSP encrypts the whole packet. There were two options for encrypting the QUIC header - using the header protection as described in RFC 9001 [30] or encrypting the whole inner packet using PSP.

I made the decision to do the encryption solely with PSP, rather than use QUIC's header protection. Normally, QUIC uses header protection on certain QUIC header fields in order to protect the connection against certain attacks. However, since PSP encrypts the entirety of all inner headers, I decided encrypting all data and leaving header protection was the best choice.

This keeps an observer from knowing the type of data being transferred from the UDP destination port, and from keeping track of bidirectional streams. Since SPIs are unidirectional, if two endpoints have more than one stream between them and any of them are bidirectional, an observer would be unable to correlate two SPIs to a bidirectional stream.

Additionally, this stops anything along the network path other than the endpoints from knowing that the packet is a QUIC packet. Therefore, I made the design decision to encrypt the entire packet from the end of the PSP header to the start of the PSP trailer. On the other hand, firewalls can cause the number of UDP packets dropped to increase dramatically [20].

Since PSP uses AES-256-GCM, an authentication tag is also generated during encryption. This is then appended to the rest of the packet, and the function returns the size of the PSP packet. The application must then send the packet encapsulated in a UDP and IP header, either IPv4 or IPv6. The format of the final packet is show in figure 4.1.

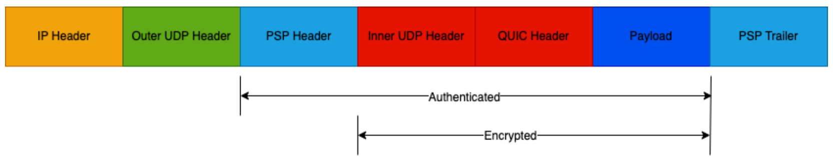

Figure 4.1: A packet using QUIC and PSP.

4.2.4 Receiving/Decryption

The process for handling a received packet is similar to that of a sending packet. If a packet is received on the recommended PSP port (50000), the application calls `quiche_psp_decrypt`.

The function first splits the packet in to three parts - the PSP header, the payload, and the PSP trailer. The SPI is read from the header, which is used to derive the encryption key. The key is then used alongside the authentication tag in the trailer to decrypt the payload.

The inner UDP header is then stripped from the packet, and the function returns the size of the decrypted QUIC packet in bytes.

4.2.5 Statistics

According the PSP architecture [2], an endpoint that sends or receives PSP packets must keep track of certain attributes. A sender should keep track of the following attributes:

- The number of packets processed for PSP transmission
- A count of the bytes processed for transmission, not including the PSP header or trailer
- The number of packets that had errors before transmission with PSP

A receiver should keep track of the following attributes of received PSP packets:

- The number of PSP packets that are processed and authenticated correctly
- A count of the bytes that have been authenticated using PSP, not including the PSP header or trailer
- The number of received packets that have failed authentication
- The number of received packets that have had length or framing issues
- The number of received packets with any other issues with PSP

While a hardware implementation should keep track of these counters on the NIC, since I am testing SoftPSP, these statistics should be tracked using the struct `quiche_psp_stats` in the header file ”quiche_psp_stat.h”.

4.2.6 Mechanisms

There are additional requirements that an endpoint must have as defined in the specification. The SPI and IV can be returned as an optional value to a higher layer application. The implementation returns a pointer to the buffer if properly authenticated and decrypted, or -1 if not, noting in the database the failed packets and why they have failed. The packets still exist so they can be inspected by the application. Control over rate limiting is done by the application and by the kernel/firewall.

A sender that fails to encrypt a packet does not send that packet and instead can inspect it. Support can be added in the future for packets with all required headers to be authenticated/encrypted appropriately and given the proper trailer.

4.3 Combined

In order to combine PSP and QUIC, I create security associations, and also associate them with a QUIC connection. Since security associations are unidirectional, a security association is created for each direction, with a note for the direction.

Chapter 5

Methodology

5.1 Testing Environment

In order to keep the results as fair as possible, the testing of the protocols was done in the same environment. The main choices for the environment were a virtual machine, or a docker container. While a virtual machine has hooks for syscalls in the host system, a docker container uses the kernel space of the host machine. On the other hand, a docker container allows for easier reproducibility with the testing environment. Therefore, I made the decision to use a Vagrant box running an Ubuntu virtual machine. This gives the hooks for the syscalls, allows a virtual network to be created on the machine, and the Vagrant environment allows for the reproducibility that is needed for testing and comparing the different protocols.

5.1.1 Host Machine

The hardware specifications of the host machine can be found in table 5.1. I used my personal laptop, a 2019 Macbook Pro as the host device since it is a portable UNIX environment and what I use for development.

System Information	Specs
Operating System	macOS Sonoma 14.3.1
Processor	2.4 GHz Quad-Core Intel Core i5
Memory	8 GB 2133 MHz LPDDR3

Table 5.1: The hardware of the host machine.

5.1.2 Virtual Machine

The technical specifications of the virtual machine can be found in table 5.2. For the virtual machine, I decided to use a Vagrant box running a long time service version of Ubuntu. This is because Ubuntu is commonly used in the virtual machines of a data centre [1], giving a somewhat similar environment to that of a production data centre. I have given the virtual machine the minimum requirements for an Ubuntu machine

[5]. This is to allow the proper running of the virtual machine without slowing down the host machine too much, especially since I require the socket interface provided in kernel space.

System Information	Specs
Vagrant Box	Focal 64
Operating System	Ubuntu 20.04.6 (LTS)
Processor Cores	2 Cores
Memory	4 GB

Table 5.2: The specifications of the virtual machine.

5.2 Testing

The PSP implementation was tested by calculating overhead associated with creating a QUIC packet. This involves three separate categories of packet:

- Plaintext QUIC packet
- PSP encrypted QUIC packet

The unencrypted QUIC packet creation time is take as a baseline, and the PSP packet encryption overhead is then calculated. This allows the average percentage overhead to be calculated for PSP packets.

Since the network speed is not being tested, with the additional bytes from the PSP header and trailer being negligible, the packets are not sent. They are instead handled using only the functions from quiche and the PSP implementation. This stops any I/O delay, and focuses only on the cost of encrypting and decrypting, as well as adding/removing headers.

The size of the packets is differed in order to better understand how this affects the overhead. The packet sizes used were (in bytes):

- 1200
- 2400
- 3600
- 4800

This will give a better understanding of optimal packet size for PSP.

Chapter 6

Results

From table 6.1, we can see that the average time to construct a plain QUIC packet is 0.342533ms, while the average time to construct a PSP packet is 0.413333ms. This works out to an overhead cost of 20.67% to create a PSP packet over an ordinary QUIC packet.

Further, looking at figure 6.1, we can see that while the average packet construction time is similar, the maximum time is much higher for a PSP packet. Additionally, we can see from the plot that nearly a quarter of the PSP packets constructed took over half a millisecond to construct. We can also see that the construction times for the ordinary QUIC packets are far more consistent than PSP packets.

In addition, there is a far greater difference between the fastest and slowest packet encrypted with PSP than with the fastest and slowest created solely with QUIC.

Table 6.2 shows the time to encrypt and decrypt larger packets. Further, the variance in the time to encrypt and decrypt packets can be seen in figure 6.2 and figure 6.3. From these, we can see that the time to encrypt or decrypt packets is not dependent on the packet size.

The time to encrypt is fairly consistent across all three chosen packet examples. The average times to encrypt were calculated to be 0.0113750ms for 2400 byte packets, 0.0111250ms for 3600 packets, and 0.010938ms for 4800 byte packets. Similarly, the average decryption times were 1.282800ms for 2400 bytes, 0.865933ms for 3600 bytes, and 1.033000ms for 4800 bytes. From these and from the box plot, we see that the decryption time varies a lot more than encryption, along with taking more time, however there does not appear to be a correlation with packet size.

6.1 Evaluation

Based on the results, PSP can be implemented with QUIC, with potentially small overhead. Looking forward, this can be reduced even further if the full advantage of PSP is used by encrypting the packets with a SmartNIC, instead of using SoftPSP. We can see that

QUIC	PSP/QUIC
0.385000ms	0.437000ms
0.304000ms	0.602000ms
0.393000ms	0.354000ms
0.327000ms	0.331000ms
0.314000ms	0.323000ms
0.397000ms	0.613000ms
0.286000ms	0.338000ms
0.298000ms	0.318000ms
0.330000ms	0.393000ms
0.374000ms	0.332000ms
0.292000ms	0.523000ms
0.477000ms	0.374000ms
0.291000ms	0.593000ms
0.325000ms	0.322000ms
0.345000ms	0.347000ms

Table 6.1: The time taken to create a 1200 byte packet

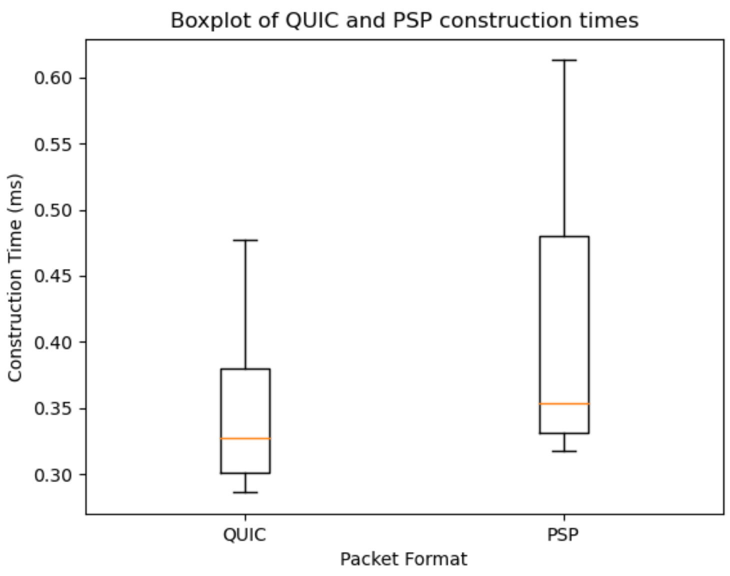

Figure 6.1: A box plot of the packet construction times for QUIC and PSP/QUIC

2400 byte packet		3600 byte packet		4800 byte packet	
Encryption	Decryption	Encryption	Decryption	Encryption	Decryption
0.078000ms	1.638000ms	0.107000ms	0.929000ms	0.072000ms	1.096000ms
0.008000ms	1.365000ms	0.005000ms	0.902000ms	0.007000ms	1.077000ms
0.010000ms	1.308000ms	0.005000ms	0.905000ms	0.007000ms	1.085000ms
0.007000ms	1.339000ms	0.004000ms	0.886000ms	0.007000ms	1.198000ms
0.007000ms	1.207000ms	0.006000ms	0.885000ms	0.008000ms	1.056000ms
0.008000ms	1.225000ms	0.005000ms	0.850000ms	0.009000ms	1.005000ms
0.006000ms	1.240000ms	0.004000ms	0.870000ms	0.007000ms	1.204000ms
0.008000ms	1.172000ms	0.005000ms	0.825000ms	0.006000ms	0.920000ms
0.005000ms	1.148000ms	0.004000ms	0.927000ms	0.006000ms	0.955000ms
0.006000ms	1.166000ms	0.006000ms	0.854000ms	0.007000ms	0.920000ms
0.006000ms	1.166000ms	0.004000ms	0.842000ms	0.006000ms	0.915000ms
0.005000ms	1.177000ms	0.005000ms	0.824000ms	0.006000ms	0.912000ms
0.006000ms	1.104000ms	0.004000ms	0.812000ms	0.012000ms	0.920000ms
0.006000ms	1.771000ms	0.005000ms	0.858000ms	0.008000ms	1.070000ms
0.008000ms	1.216000ms	0.003000ms	0.820000ms	0.005000ms	1.125000ms

Table 6.2: The time to encrypt and decrypt PSP packets of given sizes.

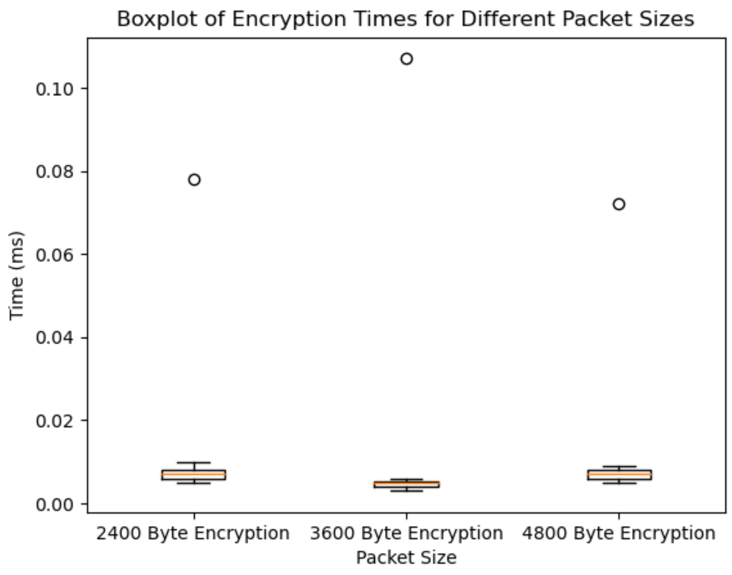

Figure 6.2: A box plot of the time to encrypt packets.

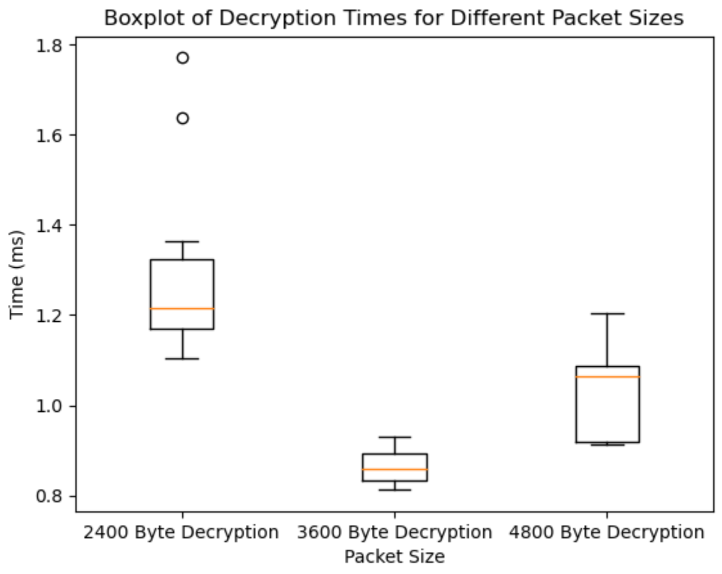

Figure 6.3: A box plot of the time to decrypt packets.

6.1.1 Use Cases

As PSP was designed for use in data centres, the obvious use case for PSP/QUIC is to reliably send data within and between data centres. QUIC can be useful in this scenario since once a connection has been created, QUIC allows 0-RTT resumption, meaning that the endpoints can send application data from the first packet without the need to repeat the handshake, unlike TCP [15].

PSP also provides users with greater privacy, since it encrypts the inner UDP header. While HTTP/3 data will be sent on ports 443 or 8443, other applications use other ports which can be observed by network observers. Therefore, with the reliability provided by QUIC and the security and privacy provided by PSP, a QUIC/PSP connection can be used, with SoftPSP used at the client end, and PSP used by the server on a SmartNIC.

Additionally, qiuche provides support for QUIC datagrams, packets that do not require retransmission [26]. This can be used in situations such as when an application wants to make use of both reliable streams alongside unreliable data, as both are sent over the same QUIC connection.

Chapter 7

Conclusion

PSP over QUIC provides a way to encrypt reliable traffic while supporting hardware offloading, and the ability to support a much larger number of incoming connections through the lack of keeping them in memory.

The overhead created by PSP is manageable, and with the support of hardware offloading this could provide a much lower overhead. The use of early data sending in the handshake packets also allows endpoints to determine an encryption key, since PSP makes use of master keys and security association specific information.

Since the time to create and decrypt a PSP packet do not seem to be directly correlated with packet size, and other than the encryption and decryption operations, PSP packets are independent of packet length, the protocol should hopefully scale well and deal with a larger amount of data being sent in any of the potential use cases.

7.1 Challenges

There were various challenges faced while implementing PSP with QUIC. These were mainly centred on working with QUIC implementations.

One of the common errors found was with creating QUIC packets to then be encrypted with PSP. Since the RFC for QUIC and the standardisatio from the IETF demands TLS is used, early data had to be enabled, and the PSP packets encrypted were early packets to avoid encrypting the same packet and information twice.

Further, while experimenting with ngtcp2 as an option for the QUIC implementation, I found that I could not construct packets without causing a segmentation fault. This was a main reason for switching to quiche, especially since quiche provides better mechanisms for logging errors.

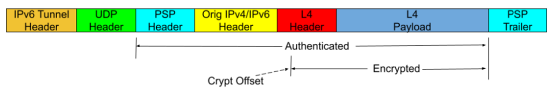

Figure 7.1: The headers for a PSP tunnel mode packet [2].

7.2 Future Work

7.2.1 Hardware Offloading

The obvious next step is to test PSP with a SmartNIC, offloading encryption and keeping relevant information in memory. This should provide a clear and obvious speedup, especially when dealing with lots of data at once.

With access to two endpoints equipped with SmartNICs, there is the potential to properly test the hardware offloading capabilities, and to test the benefits at scale, since a large number of connections are needed to take advantage of the fact that PSP does not store security associations that it receives.

7.2.2 QUIC Agnostic

While this implementation is focused on quiche, I believe there is potential for a future version that is able to be used with different QUIC implementations. QUIC implementations use connections to handle traffic between endpoints. My implementation adds quiche connections to the security association to allow for easier correlation. Therefore, there exists the potential to instead find a way to not rely on a single QUIC implementation, and work with all instead. This is furthered by the fact that PSP as first described is agnostic to the transport protocol that it encapsulates.

7.2.3 Tunneling Mode

The PSP architecture specification also describes tunnel mode, a packet that encapsulates an IP packet with a PSP header, followed by a UDP header, then an IP tunnel header (as seen in figure 7.1). This is supported further by QUIC datagrams, which can be used to supply data for a Virtual Private Network (VPN) after reliably sending data for the handshake to establish the connection [26].

Bibliography

[1] Jessica Agorye. *Operating Systems Common in Cloud Data Centers*. en. Feb. 2024. URL: https://verpex.com/blog/cloud-hosting/operating-systems-common-in-cloud-data-centers (visited on 03/31/2024).

[2] Cedell Alexander and Lance Richardson. *psp*. https://github.com/google/psp/tree/main. 2022.

[3] Mark Allman and Aaron Falk. "On the effective evaluation of TCP". In: *ACM SIGCOMM Computer Communication Review* 29.5 (Oct. 1999), pp. 59–70. ISSN: 0146-4833. DOI: 10.1145/505696.505703. URL: https://dl.acm.org/doi/10.1145/505696.505703 (visited on 04/12/2024).

[4] Ryan Hamilton Alyssa Wilk and Ian Swett. *A QUIC update on Google's experimental transport*. 2015. URL: https://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html (visited on 03/25/2024).

[5] *Basic installation*. en. URL: https://ubuntu.com/server/docs/installation (visited on 04/01/2024).

[6] David Belson and Lucas Pardue. *Examining HTTP/3 usage one year on*. 2023. URL: https://blog.cloudflare.com/http3-usage-one-year-on/ (visited on 03/26/2024).

[7] Helali Bhuiyan et al. "TCP Implementation in Linux: A Brief Tutorial". en. In: ().

[8] Dragana Damjanovic. *QUIC and HTTP/3 Support now in Firefox Nightly and Beta*. 2021. URL: https://hacks.mozilla.org/2021/04/quic-and-http-3-support-now-in-firefox-nightly-and-beta/ (visited on 03/26/2024).

[9] Kevin Deierling. *What Is a SmartNIC?* en-US. Oct. 2021. URL: https://blogs.nvidia.com/blog/what-is-a-smartnic/ (visited on 04/22/2024).

[10] Deland-Han. *The three-way handshake via TCP/IP - Windows Server*. en-us. Dec. 2023. URL: https://learn.microsoft.com/en-us/troubleshoot/windows-server/networking/three-way-handshake-via-tcpip (visited on 04/12/2024).

[11] *DNS-over-HTTP/3 in Android*. en. URL: https://security.googleblog.com/2022/07/dns-over-http3-in-android.html (visited on 04/22/2024).

[12] Neil Dunbar. "IPsec Networking Standards—An Overview". In: *Information Security Technical Report* 6.1 (2001), pp. 35–48.

[13] Christopher Fernandes. *Microsoft to add support for Google's QUIC fast internet protocol in Windows 10 Redstone 5*. 2018. URL: https://www.windowslatest.com/2018/04/03/microsoft-to-add-support-for-googles-quic-fast-internet-protocol-in-windows-10-redstone-5/ (visited on 03/26/2024).

[14] Alessandro Ghedini. *Enjoy a slice of QUIC, and Rust!* en. Jan. 2019. URL: `https://blog.cloudflare.com/enjoy-a-slice-of-quic-and-rust` (visited on 03/29/2024).

[15] Alessandro Ghedini. *Even faster connection establishment with QUIC 0-RTT resumption*. en. Nov. 2019. URL: `https://blog.cloudflare.com/even-faster-connection-establishment-with-quic-0-rtt-resumption` (visited on 04/22/2024).

[16] Alessandro Ghedini and Rustam Lalkaka. *HTTP/3: the past, the present, and the future*. en. Sept. 2019. URL: `https://blog.cloudflare.com/http3-the-past-present-and-future` (visited on 04/22/2024).

[17] Kipp Hickman and Taher Elgamal. "The SSL protocol". In: (1995).

[18] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. RFC Editor, May 2021. URL: `https://datatracker.ietf.org/doc/html/rfc9000`.

[19] Stephen Kent. *IP Encapsulating Security Payload (ESP)*. RFC 4303. Dec. 2005. DOI: `10.17487/RFC4303`. URL: `https://www.rfc-editor.org/info/rfc4303`.

[20] Mutaz Hamed Hussien Khairi et al. "The impact of firewall on TCP and UDP throughput in an openflow software defined network". en. In: *Indonesian Journal of Electrical Engineering and Computer Science* 20.1 (Oct. 2020), p. 256. ISSN: 2502-4760, 2502-4752. DOI: `10.11591/ijeecs.v20.i1.pp256-263`. URL: `http://ijeecs.iaescore.com/index.php/IJEECS/article/view/21863` (visited on 04/22/2024).

[21] Hugo Krawczyk, Kenneth G Paterson, and Hoeteck Wee. "On the security of the TLS protocol: A systematic analysis". In: *Annual Cryptology Conference*. Springer. 2013, pp. 429–448.

[22] Hyunwoo Lee, Doowon Kim, and Yonghwi Kwon. "TLS 1.3 in Practice:How TLS 1.3 Contributes to the Internet". en. In: *Proceedings of the Web Conference 2021*. Ljubljana Slovenia: ACM, Apr. 2021, pp. 70–79. ISBN: 978-1-4503-8312-7. DOI: `10.1145/3442381.3450057`. URL: `https://dl.acm.org/doi/10.1145/3442381.3450057` (visited on 04/13/2024).

[23] Fan Liu and Patrick Crowley. "Security and Performance Characteristics of QUIC and HTTP/3". In: *Proceedings of the 10th ACM Conference on Information-Centric Networking*. 2023, pp. 124–126.

[24] Josip Lorincz, Zvonimir Klarin, and Julije Ožegović. "A Comprehensive Overview of TCP Congestion Control in 5G Networks: Research Challenges and Future Perspectives". en. In: *Sensors* 21.13 (Jan. 2021). Number: 13 Publisher: Multidisciplinary Digital Publishing Institute, p. 4510. ISSN: 1424-8220. DOI: `10.3390/s21134510`. URL: `https://www.mdpi.com/1424-8220/21/13/4510` (visited on 04/12/2024).

[25] *ngtcp2/ngtcp2*. original-date: 2017-06-25T08:28:58Z. Apr. 2024. URL: `https://github.com/ngtcp2/ngtcp2` (visited on 04/14/2024).

[26] Tommy Pauly, Eric Kinnear, and David Schinazi. *An Unreliable Datagram Extension to QUIC*. Request for Comments RFC 9221. Num Pages: 9. Internet Engineering Task Force, Mar. 2022. DOI: `10.17487/RFC9221`. URL: `https://datatracker.ietf.org/doc/rfc9221` (visited on 04/22/2024).

[27] Boris Pismenny et al. "Autonomous NIC offloads". en. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Virtual USA: ACM, Apr. 2021, pp. 18–35. ISBN: 978-1-4503-8317-2. DOI: `10.1145/3445814.3446732`. URL: `https://dl.acm.org/doi/10.1145/3445814.3446732` (visited on 03/28/2024).

[28] Boris Pismenny et al. "TLS offload to network devices". In: *The Technical Conference on Linux Networking (Netdev)*. 2016.

[29] stevewhims. *Getting started with Winsock - Win32 apps*. en-us. Feb. 2023. URL: `https://learn.microsoft.com/en-us/windows/win32/winsock/getting-started-with-winsock` (visited on 04/23/2024).

[30] Martin Thomson and Sean Turner. *Using TLS to Secure QUIC*. RFC 9001. RFC Editor, May 2021. URL: `https://datatracker.ietf.org/doc/html/rfc9001`.

[31] Maroun Tork, Lina Maudlej, and Mark Silberstein. "Lynx: A SmartNIC-driven Accelerator-centric Architecture for Network Servers". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, Mar. 2020, pp. 117–131. ISBN: 978-1-4503-7102-5. DOI: `10.1145/3373376.3378528`. URL: `https://dl.acm.org/doi/10.1145/3373376.3378528` (visited on 04/22/2024).

[32] Amin Vahdat and Soheil Hassas Yeganeh. *Announcing PSP's crytpographic hardware offload at scale is now open source*. 2022. URL: `https://cloud.google.com/blog/products/identity-security/announcing-psp-security-protocol-is-now-open-source` (visited on 03/26/2024).

[33] Venafi. *What Is TLS/SSL Offloading? — Venafi*. en. Sept. 2022. URL: `https://venafi.com/blog/what-tlsssl-offloading/` (visited on 04/22/2024).

[34] David Wagner and Bruce Schneier. "Analysis of the SSL 3.0 Protocol". en. In: (1996).

[35] Ziyang Xing et al. "Research on the transmission performance of QUIC in data center network". In: *2021 International Conference on Electronic Information Engineering and Computer Science (EIECS)*. 2021, pp. 183–187. DOI: `10.1109/EIECS53707.2021.9587937`.