Implementation and Performance Evaluation of PSP Protocol

Yanang Li

Master of Science
School of Informatics
University of Edinburgh
2024

Abstract

This study focuses on the implementation and performance evaluation of Google's Packet Security Protocol (PSP) in the Linux kernel. PSP, introduced in 2022, aims to address the efficiency needs of large-scale data center communications. Despite its potential, PSP has lacked comprehensive analysis in real-world applications. Our research tries to bridge this gap by implementing PSP in the Linux kernel and conducting extensive performance tests.

In this work, I implemented the PSP protocol in the Linux kernel with key derivation, encryption sending and decryption receiving functionalities. Based on my implemented PSP, I evaluated its performance across key metrics including throughput, latency, and CPU utilization under various network conditions and compared it with TLS protocol. The experiment results show that PSP demonstrates strong stability. Even at 25% packet loss, it is able to achieve a throughput of 227Mbps. Regarding latency, PSP's latency is half that of TLS-1.3 at the same network delay, but slightly slower than TLS-1.2. My findings also reveal that PSP is particularly advantageous for receiver-side processing, which is beneficial for data centers primarily functioning as receivers. The receiving end of PSP demonstrates a processing time of 0.01253 ms, compared to the sending side's 0.06475 ms, indicating a substantially lower processing overhead for receivers.

However, my experiments also uncover areas for improvement, particularly in throughput performance under ideal network conditions. In lossless scenarios, PSP's baseline throughput is significantly lower than TLS protocols, achieving only 54% of TLS's throughput. Although PSP demonstrates lower CPU utilization compared to TLS, this advantage is partially offset by the fact that it processes less data per unit time. For instance, in perfect network conditions, TLS completes file transfers more quickly due to its superior throughput, despite higher CPU usage.

This research provides insights into PSP's performance in real network environments. However, due to hardware limitations, the NIC offloading feature of PSP protocol was not implemented. Future work should focus on implementing PSP with NIC offloading support to fully assess its capabilities in optimized hardware environments.

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.

(*Yanang Li*)

Acknowledgements

First, I would like to thank Professor Michio Honda for providing direction for the implementation of this work. His insights were crucial in pointing me in the right direction for the implementation of this work.

Second, I am also grateful to my personal tutor, Chrissy Harris, for her support and understanding during a challenging period of my academic journey. Her kindness and understanding were instrumental in helping me overcome personal difficulties and return to my studies.

Their support has been invaluable, and this work would not have been possible without their assistance and encouragement.

Table of Contents

Chapter 1

Introduction

Transport layer protocols are the cornerstone of Internet communication. They are responsible for transmitting data across the network and ensuring data integrity and reliability. The TCP (Transmission Control Protocol) protocol, proposed by Vint Cerf and Bob Kahn in the 1970s, is one of the core transport protocols in the Internet Protocol Suite (TCP/IP). TCP provides reliable and connection-oriented communication. Connections must be established through a three-time handshake mechanism. And through the sequence number and acknowledgement mechanism, TCP can ensure the reliable transmission of data packets[12]. However, with the the popularity of e-commerce, online banking and other applications, there is an increasing demand for not just reliability but security. Reliable transmission alone can no longer meet the demand for secure data transmission. However, TCP can not provide data encryption and authentication. It cannot guarantee that the data will not be stolen or tampered by hackers in the process of transmission.

To solve this problem, Netscape introduced the SSL (Secure Sockets Layer) protocol in 1995. SSL protocol works with TCP to encrypt the data stream in transit to ensure security[20]. Later the IETF (Internet Engineering Task Force) released the TLS (Transport Layer Security) protocol as a replacement for SSL[22]. TLS is used for end-to-end encrypted communications. It has handshake mechanism and strong cryptographic algorithms to ensure data on both sides of the communication can not be stolen or tampered with by third parties. However, the price for this strong security is a very high overhead. The TLS handshake process requires multiple round trips[23], and uses public key cryptographic algorithms, such as RSA or ECC for key exchange and digital signature authentication[19]. These features involve complex computational operations, which substantially increase both CPU and memory usage. Consequently,

the latency of connection establishment is significantly increased. Akshay Narayan et al. experimented with TLS for 50000 network connections. They compared this to scenarios without TLS and found that TLS caused 40% more time overhead and 39% more energy overhead, all attributed to the handshake process[33]. Moreover, the HOL(Head-of-Line) problem of TCP also reduces the performance of TLS. Because TCP must wait for the lost packet to be re-transmitted and acknowledged before it continue to transmit subsequent packets. So, HOL problem will happen many times if network environment has a high packet loss rate. The frequent retransmission and acknowledgement process will consume many network bandwidth and computational resources, thus reducing the throughput of the entire connection[2]. Obviously, these drawbacks make TLS protocol perform poorly in scenarios involving high-volume data transmission or real-time demanding applications.

With the rapid development of the Internet industry and continuous innovations in Internet technologies over the past few decades, the use of the Internet has penetrated into people's daily lives in various fields such as finance, healthcare, dining, government, and more[1]. As users engage with the Internet, they will inevitably leave behind digital traces, leading to the generation of massive amounts of data. To store and process these data, more and more companies are starting to build their own big data centers[4]. The number of large data centers managed by hyperscale providers reached 992 by the end of 2023 and exceeded 1,000 by early 2024. At the same time, the total capacity of hyperscale data centers has doubled in just four years. The total capacity of hyperscale data centers is expected to double again over the next four years[10]. These hyperscale data centres of the same company may be located in different regions[9]. Problems arise. How to carry out secure and high-performance transmission between data centres has become a thorny issue.

One single connection may need to transmit a lot of data between data centers. From above, it is clear that the TLS protocol cannot be used for data transfer between big data centres due to its high overhead problem and HOL problem. Based on the security and efficiency considerations, Google developed a new encrypted transmission protocol named PSP (PSP Security Protocol). PSP uses mechanisms such as AES-GCM encryption algorithm and UDP-based connection to achieve efficient transmission with maximum privacy protection. UDP (User Datagram Protocol) is a connectionless, lightweight transport layer protocol. Its header is only 8 bytes, much smaller than the 20-byte header of TCP. In scenarios where a large amount of data is transmitted, UDP can transmit more payload compared to TCP. This will reduce the wastage of bandwidth

which can be a big cost saving for big data centres. Moreover, UDP does not suffer from Head-of-Line blocking due to its connectionless feature. From these, PSP protocol based on UDP is very suitable for data transfer in big data centres. But as Google open sourced this protocol design only in 2022. There is very few implementation and analysis about PSP protocol compared to the TLS, but this is again very important. Implementing PSP in real-world systems is crucial for verifying the protocol's design principles, identifying practical challenges, and enabling realistic performance testing. Equally important is a comprehensive analysis of PSP, which is essential for understanding its real-world efficiency, potential advantages, and limitations in various network conditions. Both the implementation and analysis are critical for organizations to make informed decisions about using this new protocol in their data center infrastructures. Thus, in this work, I will implement the PSP protocol in the Linux kernel and conduct a comprehensive performance analysis. My goal is to provide a practical reference implementation and performance benchmarks that can be valuable for companies and organizations considering the adoption of the PSP protocol for large-scale data center communications.

The paper is structured in the following order. In Chapter 2, I will introduce the basic components of the PSP protocol, including its architecture and key features. Chapter 3 will describe the implementation details of PSP in the Linux kernel, focusing on how its key functionalities are realized and how it integrates with the existing network stack. In Chapter 4, I will present my performance analysis of the implemented PSP protocol, comparing it with traditional encrypted transport protocol. Finally, in Chapter 5, I will draw conclusions based on my experimental findings, discuss the limitations of my work, and suggest directions for future improvements and research.

Chapter 2

Background

Since there is no academic paper about the PSP protocol, in order to better understand its advantages in improving the encrypted data transmission in terms of speed and security. This chapter will concentrate on the design of the PSP protocol combined with the related basic principles. All information regarding the PSP protocol design presented in this chapter is derived from the official PSP documentation [8]. Through an in-depth analysis of its design patterns and key technologies used, it will provide a comprehensive understanding of the innovations of the PSP protocol and its potential for application in modern big data centre transmission.

2.1 Why PSP

To understand the background and purpose of the PSP protocol, let me start with an introduction to its designer - Google. As the third largest cloud provider in the world (after Amazon AWS and Microsoft Azure)[24], it is very important for Google to keep its users' data secure.

About 12 years ago, Google began encrypting data transmission between large data centres. In order to meet the increasing demand for security, Google gradually encrypted all data transmissions. While this greatly protects users' privacy and data security, it comes at a significant cost. Encryption and decryption using software consume approximately 0.7% of Google's processing capacity, and also requirs a corresponding amount of memory. In addition, Google's servers are shared by multiple users and have high isolation requirements between users. This means that different users cannot share the same connection. According to the data published by Google, the number of real-time TCP connections in Google reaches millions. At its peak, it

maintains 100000 new connections per second[26]. From these views, it is clear that the TLS can not meet the needs of Google due to its high overhead and HOL problem. Although TLS1.3 improved to achieve 1-RTT and 0-RTT handshaking processes[6]. TLS-1.3 outperforms TLS-1.2 by more than 57.9% in terms of average performance as measured by Hyunwoo Lee et al [14]. But, Danylo Goncharskyi et al. set up experiments to recorded the time overhead of TLS handshake and came up with 107 milliseconds per connection at 2-RTT, 85 milliseconds per connection at 1-RTT, and 83 milliseconds per connection at 0-RTT. [7]. This means that 10.7% (2-rtt), 8.5% (1-rtt) and 8.3% (0-rtt) of every 1 second is spent on handshaking respectively, which is clearly a lot of wasted time for Google.

Therefore, in order to reduce the time overhead of establishing a connection and the computational and storage overhead of encryption and decryption, Google developed the UDP-based PSP protocol. PSP is also combined with the AES-GCM encryption algorithm and key derivation mechanism to enhance the performance of encrypted data transmission while ensuring the highest level of security. Below I will further introduce these basic components of PSP in detail.

2.2 UDP-based Transmission

UDP (User Datagram Protocol) is a connectionless transport layer protocol. Due to its connectionless nature, there is no need to establish a connection before sending data. So, data transmission can start immediately without waiting for handshaking and connecting. Besides, each packet (called datagram) is transmitted independently and does not depend on previous datagrams. Therefore there is no need to acknowledge and retransmit packets[3]. Thus the overhead during transmission is greatly reduced.

Compared to TCP, UDP is particularly well suited for fast and large-scale data transmission. Qiuyu Lu et al. implemented an improved version of UDP replacing point-to-point fibre-optic communication to maintain the stability of a large interconnected power system. They implemented UDP-RT by adding mechanisms such as error correction, retransmission and timeout retransmission at the application layer. Through experiments, in the case of network congestion, the communication delay of UDP-RT was 99.7 milliseconds, and the reliability was 99.606%; while TCP was 7 to 306 seconds, and the reliability was 98.263%[15]. It is clear that UDP performs much better than TCP in areas where real-time communication is required. Lisha Ye et al, considering TCP can't meet the high demands for low latency and high throughput in modern

data centre applications, implemented DCUDP for data transmission between data centers. DCUDP is based on UDP, which could provide excellent congestion control by Explicit Congestion Notification (ECN) mechanism they implemented. They set up a high congestion scenario to test the performance. Results showed the throughput of DCUDP has been able to maintain 1000 Mbps. In contrast, DCTCP's throughput can be maintained only at 850Mbps under short flows. TCP is even worse with 675Mbps under normal flows. These results demonstrate that UDP outperforms many TCP-like data centre protocols in terms of throughput[32].

Since businesses operate globally now. Large data centers of the same company may be located in different parts of the world. The distance between them may be very far which will cause high network delay. E. He and colleagues explored the performance of TCP over long distance data transmission. They found when transmitting data over long distances, such as between the United States and Europe or Asia, using TCP resulted in significant bandwidth underutilization. This was TCP's window mechanism that limited how much data can be transmitted before an acknowledgment is received. Due to the high delay on international networks, TCP protocol spent excessive time waiting for acknowledgments, which prevent client's data transmission rate from reaching network's peak capacity. Based on this issue, they developed RBUDP and conducted large-load transmission experiments. The results demonstrated that being based on UDP, even with a 7% packet loss rate, there was no significant impact on throughput for large-scale payloads[11].

In addition to this, the flexibility of UDP is also the main reason why Google chose it. Although resizing the TCP window can partially solve above problems. However, on some operating systems, this requires a kernel refactoring. Such kernel-level changes are complex, time-consuming, and can introduce system-wide instabilities. By contrast, UDP allows developers to easily build application-specific reliability and flow control on top of it, without the need for kernel modifications[28]. From all this above, we can understand why Google chose UDP for real-time and reliable transport between big data centres.

In summary, being based on UDP allows us to take full advantage of UDP's connectionless and low-latency features. And the flexibility of UDP make it easier for the PSP protocol to be deployed and used in various operating systems and network environments. Therefore, choosing UDP as the foundation for the PSP protocol is an optimal choice, fully exploiting its unique advantages in large-scale data transmission.

2.3 AES-GCM Encryption

In terms of encryption algorithm, PSP only supports the AES-GCM algorithm to provide both encryption and authentication of data. AES-GCM (Advanced Encryption Standard in Galois/Counter Mode) is an AEAD (Authenticated Encryption with Associated Data) algorithm. It combines the AES block cipher and the GCM mode of operation[29].

Choosing AES-GCM was because of its strong security. There are many studies on AES and AES-GCM algorithms. KryptAll analysed the AES algorithm theoretically. Researchers determined that even with the power of a supercomputer, breaking a 128-bit AES key via brute force would require an astonishing billion billion years—far exceeding the universe's age of 13.75 billion years[13]. Hans Dobbertin et al. analysed the AES algorithm came up with the result that, apart from the speculative threat of an algebraic attack, there have been almost no effective attacks that can be carried out on the AES algorithm itself[5]. DA McGrew et al. analysed the security of the AES-GCM algorithm in terms of system security. They found AES-GCM was shown to be very secure when used correctly, unless reusing the key IV in CTR which may lead to the loss of confidentiality[17]. DA McGrew et al. analysed the security of GCM. The results demonstrated that when using correct typical parameters, GCM is better able to resist duplicate forgery attacks. Although, some commentators argued that GCM is vulnerable to multiple forgery attacks. In fact, such attacks ware rare. This discrepancy suggested that multiple forgery attacks ware more of a theoretical problem than a practical one[16].

These facts demonstrate the robust security features of AES-GCM. This strong foundation makes AES-GCM an attractive choice for implementation in various encrypted protocols. In IPsec protocol, AES-GCM is used in ESP to provide confidentiality and data origin authentication. The AES-GCM implementation in IPsec uses a unique nonce structure. It consists of a 4-byte Salt and an 8-byte Initialisation Vector (IV). IPsec strictly requires that the same nonce and key combination not be reused. Combined with an automated key management mechanism, such as Internet Key Exchange (IKE), the security of AES-GCM can be well ensured [27]. Prabhu Thiruvasagam et al. conducted performance tests on different cryptographic algorithms in IPsec. They found that AES128-GCM16 has the best throughput among all AEAD algorithms. Beside, they also conducted experiments to compare UDP and TCP. They found that UDP's throughput was better than TCP's for all encryption algorithms[25]. This reinforces the previous point that UDP is better at performance. The AES-GCM algorithm is also used in TLS.

TLS uses a unique ”partially explicit” structure for nonce management. A 12-byte nonce consists of a 4-byte implicit `salt` and an 8-byte explicit `nonce_explicit`. The `salt` is generated during the handshake, while the `nonce_explicit` is transmitted in each TLS record. TLS strictly requires that a unique `nonce_explicit` value must be used for each GCM encryption operation. TLS uses this design to ensure security while providing implementation flexibility[21].

In the design of the PSP protocol, Google has innovated mechanisms as well such as Initialisation Vector (IV), Security Parameter Index (SPI), key derivation to ensure cryptographic security. PSP has taken a special approach to generating the IV for AES-GCM. It combines a 64-bit timestamp counter and a 32-bit SPI (Security Parameters Index) to form a 96-bit unique IV. The timestamps are generated by hardware, such as a NIC (Network Interface Card). Each timestamp is 64-bit long and increments it in picoseconds. SPI generation is also automatically generated by the NIC and does not allow user space to be provided. Each SPI must be unique, to ensure that each security association (SA) is unique. A unique timestamp and a unique SPI are combined to generate a IV. By this way, within a single security association (SA), IV is unique as well and strictly incremental, effectively preventing replay attacks. This design not only ensures IV uniqueness, but also allows the hardware to efficiently generate IVs without additional queries or storage, greatly increasing the generation speed. In terms of key management, PSP adopts an innovative master key rotation mechanism and key derivation mechanism. For master key rotation mechanism, NIC will maintain two 256-bit AES master keys (`master_key_0` and `master_key_1`). The master keys will rotate at regular intervals or when the SPI space runs out. By this way, it can ensure the uniqueness of encryption parameters. For key derivation mechanism, receiver uses each unique SPI to dynamically generate the session key through the KDF (Key Derivation Function) in Counter Mode defined in NIST SP 800-108. By this way, it can not only reduces the risk of key leakage, but also eliminates the need for receiver to store large numbers of shared keys. The detail of key derivation mechanism will be decribed below. Through these innovative IV, SPI, master key rotation and key derivation mechanisms, the integrity and confidentiality of data transmission is greatly ensured[8].

2.4 PSP Packet Structure

In order to better understand how the dynamic derivation of keys is done in PSP. In this section I will describe the design of the PSP header and tail (also known as trailer)

respectively. Then, I will calculate the payload size of the PSP packet with an MTU of 1500 bytes to compare with TLS.

2.4.1 PSP Header

Figure 2.1: PSP header

The basic length of PSP header is 16 bytes (128 bits). If the optional Virtualisation Cookie field is included, the total length will be 24 bytes (192 bits). Here I will only describe the more important of these fields in the header which will be used in the next sections. Descriptions of the full range of fields can be seen in Appendix A.

- **Crypt Offset (6 bits):** Cryptographic offset, specifies the starting position of the encrypted data, in units of 4 bytes.
- **Security Parameters Index (SPI) (32 bits):** Used to identify the Security Association (SA) and derive key.
- **Initialisation Vector (IV) (64 bits):** Timestamp information used as input to the encryption algorithm.

2.4.2 PSP Tail

The length of the PSP's tail is 16 bytes. It holds the Integrity Checksum Value field (128 bits), which is used as an authentication tag for integrity verification by the AES-GCM algorithm.

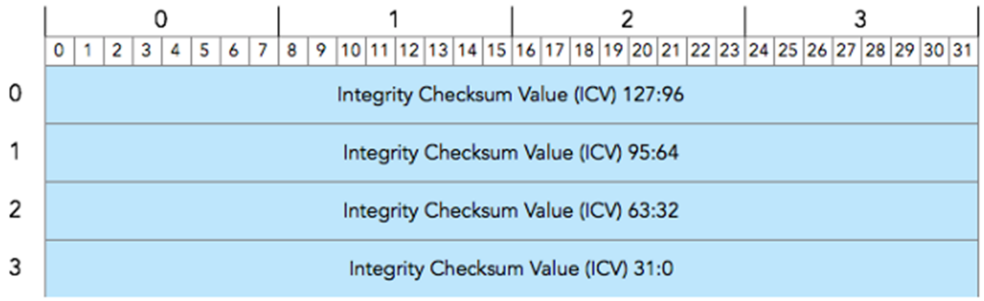


Figure 2.2: PSP tail

2.4.3 Size of Payload

Table 2.1 shows the packet payload size of the PSP and TLS protocols under different conditions. It can be seen that the payload of PSP protocol is between 1412 to 1440 bytes, while the payload of TLS protocol is between 1370 to 1439 bytes. Compared to the TLS protocol, PSP provides slightly more space for payload in some configurations, but the difference is not significant. So only comparing the payload size of this two different protocol packets, these minor differences may not cause significant performance differences in real applications.

Table 2.1: Protocol Overhead and Payload Comparison (MTU: 1500 bytes)

Protocol	IP Version	Headers + Trailer Size	Total Overhead	Payload
PSP without VC	IPv4	20+8+16+16	60	1440
	IPv6	40+8+16+16	80	1420
PSP with VC	IPv4	20+8+24+16	68	1432
	IPv6	40+8+24+16	88	1412
TLS without TCP options	IPv4	20+20+5+16	61	1439
	IPv6	40+20+5+16	81	1419
TLS with TCP options	IPv4	20+60+5+16	101	1399
	IPv6	40+60+5+16	121	1379

2.5 Key Derivation

Key derivation in PSP means that using the master key and the SPI information to dynamically compute the key for encryption and decryption. For the receiver, he only

needs to maintain two master keys in its NIC as `master_key_0` and `master_key_1`. During the handshake phase, the sender will request the master key and SPI from the receiver. Google doesn't explain the details of the PSP handshake here, so let's assume that the master key and SPI will reach to the sender securely. When they are received by the sender, the sender will use a Key Derivation Function (KDF) based on NIST SP 800-108 to generate the encryption key. KDF uses counter mode with AES-CMAC as a pseudo-random function (PRF). The input to KDF consists of four components:

1. Counter: starting from 1, used to generate multiple key blocks.
2. Label: the protocol version of PSP.
3. Context: the value of the SPI, used to uniquely identify an SA.
4. Length: the length of the generated key.

The process of a complete key derivation can be represented like this:

1. Select the active master key based on the highest bit of the SPI. The highest bit is 0 for `master_key_0`, 1 for `master_key_0`.
2. Construct the KDF inputs.
3. Use AES-CMAC as the PRF, pass in the master key, and then generate the key.
4. If it is a 256-bit key, increase the counter to 00 00 00 02, then repeat steps 2-3 and connect the two outputs to generate the key.

Chuah et al. proposed a framework for evaluating the security of key derivation functions (KDFs) in 2012. This framework defines five security models: KPM, KPS, CCM, CCS, and CPM, representing different attack scenarios from low to high levels[31].

Under this framework, the key derivation mechanism of PSP protocol shows interesting security characteristics.

First, in the KPM and KPS models, which focus on known public input attacks, PSP employs multiple elements as input: a counter, protocol version label, SPI value, and target key length to be against this. Even if an attacker obtains multiple SPI values, the unique association of each SPI with a specific SA prevents the derivation of keys for other SAs.

The CCM and CCS models allow attackers to adaptively choose context information. PSP's key selection mechanism demonstrates its strengths in these scenarios. While

attackers can choose the context information (essentially selecting the SPI), the dual role of SPI in both KDF input and master key selection complicates the key derivation process. Besides, the master key rotation mechanism, which changes the master key at fixed intervals or upon SPI exhaustion, further limits the attacker's ability to exploit any specific SPI selection.

In the most stringent CPM model, where attackers can adaptively choose all public inputs, PSP relies on its NIST SP 800-108 based KDF implementation and AES-CMAC as PRF. This design ensures that even with attacker-chosen public inputs, including SPI, the final key derivation process remains dependent on the periodically rotated master key, which remains confidential.

However, PSP is not 100% secure and potential vulnerabilities exist. In the CPM model, the master key selection mechanism's reliance on a single bit of the SPI could allow attackers to influence master key selection by manipulating the SPI. While the master key rotation mechanism mitigates this risk to some extent by limiting the window of vulnerability, it does not entirely eliminate this potential weakness after all. Aside from this minor drawback, the PSP's KDF still has a high level of security.

In addition to security, PSP also takes into account the practical requirements and hardware limitations of large-scale data centres. Except that the sender encrypts the packet using the derived key, he will also write the SPI into the header of the packet. So when the receiver receives the packet, he could use the same key derivation process to generate a key based on his own master key and the SPI value in the packet. Then receiver could use this derived key to decrypt the data.

By this mechanism, all the receiver needs to store is its master key. This greatly reduces the storage pressure caused by storing keys for data centres where 100,000 new connections are added per second. Let's do a calculation, assuming a single server may need to manage up to 15 million secure connections. If each connection requires two SA, and each SA is 512 bytes in size, the total DRAM requirement would be 512B * 2 * 15M = 15GB. By employing the PSP key derivation mechanism, this requirement can be reduced by half, saving approximately 7.5GB of DRAM. This reduction is particularly significant given the limited memory capacity of most NICs.

2.6 Network Packet Processing in Linux Kernel

Having examined the innovative mechanisms of PSP, let's now turn our attention to its practical implementation within the Linux kernel network stack. In order to better

understand how to implement PSP in Linux kernel network stack. Here I am going to briefly describe the processing flow of network packets in Linux kernel.

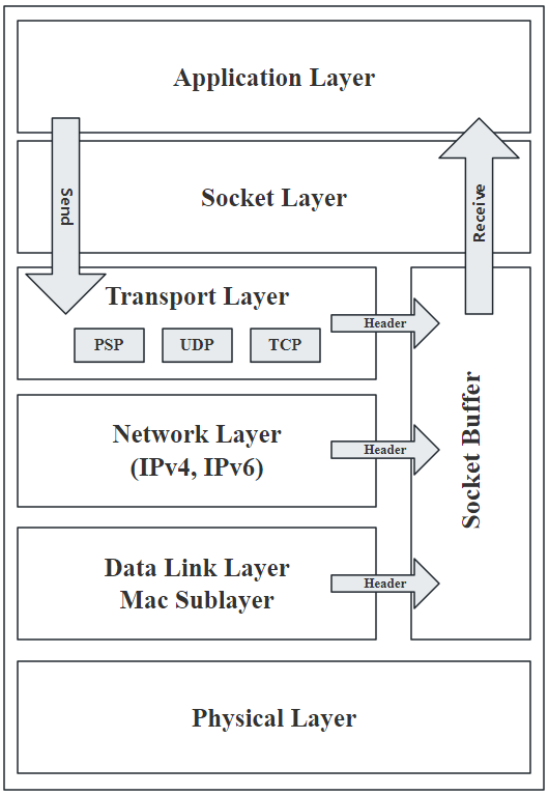


Figure 2.3: Network Packet Processing in Linux kernel

The flow of network packet processing can be described by a hierarchical structural model, as shown in Figure 2.3. Let's take the sending process as an example. In the Linux kernel, data is first generated in user space by the Application Layer. Then, applications can use the socket interface (Socket Layer) to pass the data into the kernel space. The Socket Layer acts as a bridge between the Application Layer and the Transport Layer. It provides an abstract interface for the application to call, completing the transfer of data between user space and kernel space.

Data then travels through the Socket Layer to the Transport Layer. Transport Layer has many protocols such as TCP and UDP protocol. In this layer, data starts to be encapsulated into different types of packets depending on the protocol. In UDP, for example, the kernel will add a UDP header to the packet, which contains the source port number, the destination port number, the length of the data, and the checksum information. These data will be placed in front of the payload as the UDP header of the packet.

After being encapsulated by UDP, the packet next enters the Network Layer. At this

layer, packets will continue to be encapsulated with IP header information, including source and destination IP addresses, protocol identifiers, and other control information used for routing and transmission. The Network Layer primarily provides routing functions. It uses IP addresses and routing table lookups, among other mechanisms, to ensure that packets can reach the correct destination across complex network topologies.

After passing through the Network Layer, the packet comes to the Data Link Layer. At the MAC Sublayer, the packet will be adding the mac header to be encapsulated into a frame. The frame header includes the source MAC address, destination MAC address, and other control information to ensure that the packet is transmitted correctly in the local network.

Finally, the packet reaches the Physical Layer where it interacts directly with the network hardware devices. Packets will be converted into electrical, optical or wireless signals and sent to the network over a physical medium such as copper wire, optical fibre or radio waves. By here, one data sending is done.

Conversely, receiving of data is the reverse process of sending. The data packet will be processed in reverse through the same path from the Physical Layer to the Application Layer[30].

Having understood the packet processing flow in the Linux kernel, we can know that Transport Layer is the first stop after application data enters the kernel space and the last stop before it leaves the kernel space to the user space. So, Transport Layer provides the best place to implement end-to-end secure communication. In order to ensure the confidentiality of the data, implementing the PSP protocol in the Transport Layer is the best choice. In the next chapte, I will describe how my PSP is implemented in the Transport Layer of Linux kernel.

Chapter 3

Implementing PSP in Linux Kernel

The implementation of PSP protocol mainly consists of the following parts: header file construction, key derivation function, encryption and packet sending function, packet receiving and decryption function, and PSP initialisation and configuration. In this chapter, I will present the implementation detail of PSP following the flow of how the PSP handles data. By this flow, we can understand the whole life cycle of PSP protocol in packet creation, encryption, sending, receiving and decryption.

3.1 Header File Construction

Header file construction is the very first and most important step in implementing the PSP protocol. The `psp.h` file is defined in the `include/net` directory of the Linux kernel source code.

The core part of `psp.h` is the definition of two structures: `struct psp_header` and `struct psp_trailer`.

`psp_header` struct defines the header of the PSP packet. It contains: Next header type, Header length, Ciphertext offset, Version number, Security Parameter Index (SPI), Initialisation Vector (IV) and other relevant fields. Since the Virtualisation Cookie was not needed and is optional, I did not add this field to the header.

`psp_trailer` struct defines the tail of the PSP protocol. It contains 4 ICV (Integrity Check Value) values to hold the authentication tags for verifying the authenticity and integrity of each packet.

```
6   struct psp_header {
7       __u8    next_header;
8       __u8    hdr_ext_len;
9       __u8    r:2,
10              crypt_offset:6;
11      __u8    s:1,
12              d:1,
13              version:4,
14              v:1,
15              one:1;
16      __be32 spi;
17      __be64  iv;
18  };
```

(a) psp_header

```
20  struct psp_trailer {
21      __be32 icv;
22      __be32 icv1;
23      __be32 icv2;
24      __be32 icv3;
25
26  };
```

(b) psp_trailer

Figure 3.1: Struct of psp header and tail

3.2 Key Derivation

For key derivation, I implemented the `psp_derive_key()` function for doing this. The function is defined as follows:

```
static int psp_derive_key(u32 spi, u8 *derived_key)
```

This function takes two arguments, `spi` and `derived_key`. `spi` is a 32-bit security parameter index, and `derived_key` is a pointer to the location where the derived key is stored.

Inside this function, it first calls `crypto_alloc_shash()` to initialise the CMAC-AES algorithm. Then, it will use `kmalloc()` to allocate kernel memory for the hash descriptor `desc` to ensure that there is enough space to store the descriptor and its associated data.

Next, the function uses

```
master_key = (spi & 0x80000000) ? master_key_1 : master_key_0
```

for master key selection. The principle of this is to use a bitwise AND operation with the highest bit of SPI. `0x80000000` is a binary mask whose binary representation is `1000 0000 0000 0000 0000 0000 0000 0000` which is 32 bit. Because `spi` is a 32-bit value. When `spi & 0x80000000` is executed, the function is actually checking to see if the highest bit (bit 32) of the `spi` is non-zero. If it is non-zero, then the result of `spi & 0x80000000` will be non-zero, selecting `master_key_1`. If it is 0, then the result of `spi & 0x80000000` will be 0, selecting `master_key_0`.

Next the function will construct a 16-byte array as input to the CMAC algorithm deriving key. The array `input` holds the counter value, protocol label, spi and key length. Here I am implementing the PSP protocol with the AES-GCM128 algorithm, so

the counter, protocol label and key length fields are fixed values of `1`, `0x5076300` and `0x80` respectively.

Next, the function will use `crypto_shash_setkey()` and `crypto_shash_digest()` to set key for the CMAC and perform the real key derivation operation function. After that, the message authentication code will act as a derived key and then is written to the memory location pointed by `derived_key`. Thus, the derivation of the key is completed.

3.3 Encryption and Sending Function

The encryption and sending function is done through two key functions: `psp_output()` and `psp_encrypt()`. The `psp_output()` function is responsible for integrating the PSP protocol into the UDP sending process. It serves as the entry point for PSP to enter the Linux kernel network stack, ensuring that PSP can be seamlessly inserted into the existing network stack

The `psp_output()` call occurs in the `udp_sendmsg()` function, which is the core implementation function of the UDP protocol. `udp_sendmsg()` is responsible for handling the process of sending UDP packets. Its main tasks include processing user-supplied message, setting UDP headers, and constructing the complete UDP packet and so on. I chose to place `psp_output()` function call in the non-blocking fast path (non-corking case) processing section of the `udp_sendmsg()`. At this point, all necessary parameters have been prepared, such as socket information (`sk`), message content (`msg`), length (`len`), flow information (`fl4`), IP control message (`ipc`), routing table (`rt`), and fragmentation information (`getfrag`). These parameters are critical for building and sending full PSP packets. For determining whether to enter the PSP processing flow, I used port identification to identify if the PSP protocol is used. If the destination port number of a socket is a PSP port number, it will go to the `psp_output()` function. After successful execution of `psp_output()`, it will return the `skb` with the payload encrypted and `psp_header` and `psp_trailer` added. Then this processed `skb` will be as a parameter passed into `udp_send_skb()` function to continue the next steps of normal UDP processing.

The definition of `psp_output()` is as follows:

```
struct sk_buff *psp_output(struct sock *sk, struct msghdr *msg
    , int len,
```

```
                                struct flowi4 *fl4, struct
                                   ipcm_cookie ipc,
                                struct rtable *rt,
                                int (*getfrag)(void *, char *, int,
                                    int, int, struct sk_buff *),
                                struct inet_cork cork)
```

This function takes the prepared parameters above and returns a pointer of type `sk_buff`.

Inside the `psp_output()`, it will first perform a length check of total length of the packet to ensure it will not exceed the maximum IP packet limit after adding the PSP header and trailer. Then, the function will use `ip_make_skb()` to create an initial socket buffer (`skb`). The resulting socket buffer (`skb`) encapsulates three main components: the raw IP header, an empty UDP header, and the plaintext payload. Its structure is shown below.

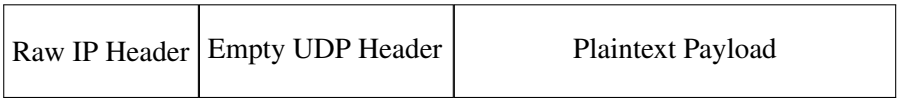


Figure 3.2: Structure of the initial socket buffer (skb)

Next, the function will call `psp_encrypt()` to perform the core encryption processing. When encryption is complete, `psp_encrypt()` will return the encrypted `skb` as below. This `skb` has ciphertext payload and added PSP header and trailer.

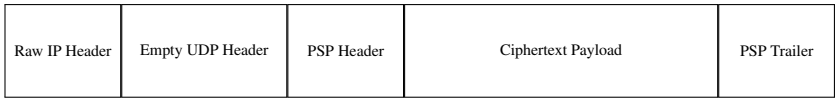


Figure 3.3: Structure of the encrypted socket buffer (skb)

At the end, `psp_output()` will update the length field in the cork according to the new encrypted `skb`.

Next let me introduce the `psp_encrypt()` function. `psp_encrypt()` function is mainly responsible for the core encryption process of the PSP protocol. It will carry out PSP header and trailer additions and data encryption for raw socket buffers.

It is defined as follows:

```
static int psp_encrypt(struct sk_buff *skb)
```

This function takes the socket buffer (`skb`) generated in `psp_output()` as function input and returns an integer value to indicate the result of the encryption operation.

First the function will calculate the raw packet length and data offset to get the start position of the payload in the `skb`. Next, using `skb_tailroom()` and `pskb_expand_head()` to expand the `skb` to ensure there is enough room to add PSP header and trailer.

When adding the header, it will use `skb_put()` to add PSP header sized space at the end of the `skb` first. Then, using the `memmove()` function to move the payload section backward by the length of the PSP header size. By doing this, the PSP header is thus moved to the location between the UDP header and the payload. Next function will fill in the PSP header and add the PSP trailer.

After adding the header and trailer, the function starts to configure various encryption parameters. For the encryption IV, the function will use `memcpy()` to merge the `header->spi` with the `header->iv` to get a 96-bit encryption IV. After this, function then will calculate the length of the plaintext as well as the ciphertext using the `crypt_offset` field in the header.

To define the encrypted and unencrypted segments of the data in `skb`, the function utilizes the scatterlist mechanism provided by the Linux kernel. Function then uses `sg_init_table()` and `sg_set_buf()` to create a three-part scatterlist structure corresponding to the PSP header and plaintext, the data to be encrypted, and the PSP trailer respectively.

Then, function will call `psp_derive_key()` to generate a encryption key based on `header->spi` and `master_key`, and then use `crypto_aead_setkey()` to set that encryption key into the AEAD encryption context. After allocating the AEAD request via `aead_request_alloc()`, function next will use `aead_request_set_ad()` and `aead_request_set_crypt()` to set the association data and relevant encryption parameters.

After executing these, all the encryption parameters are prepared. Then function will carry out the actual encryption operation by calling `crypto_aead_encrypt()`. For parameters passed into `crypto_aead_encrypt()`, here the input and output buffers use the same scatterlist `sg`, so that the encrypted ciphertext can directly overwrite the original plaintext. By encrypting in the same memory location, it can greatly reduce memory usage and allocation operations, thus reduce the risk of plaintext data leakage and improve processing speed and efficiency.

After completing encryption, the AEAD request will be released by calling `aead_request_free()`. The whole `psp_encrypt()` is done.

3.4 Packet Receiving and Decryption Process

Packet receiving and decryption is the inverse process of packet encryption and sending. They share the same implementation framework. The functionality of this part is also provided by two functions: `psp_decrypt()` and `psp_rcv()`.

Similar to `psp_output()`, the `psp_rcv()` function is responsible for integrating the PSP protocol into UDP receiving process. The call to the `psp_rcv()` function occurs in the `udp_rcv()`, which is the first function in the Linux kernel network stack that handles received UDP packets. This means that `udp_rcv()` would be called whenever an UDP packet is received. Here I chose to place the `psp_rcv()` call at the very beginning position inside the `udp_rcv()` function. So that when entering the `udp_rcv()` function, if PSP protocol is configured and the destination port of `skb` belongs to PSP port, this `skb` will be passed into the `psp_rcv()` to initiate PSP receiving processing. After execution, `psp_rcv()` will return a `skb` with ciphertext decrypted into plaintext payload and without PSP header and trailer as below. The absence of the IP header here is due to its removal by the IP layer.

Figure 3.4: Structure of the decrypted socket buffer (skb)

Then this processed `skb` will be passed into the `__udp4_lib_rc()` for the next normal UDP processing.

Here placing the PSP entry point at the very beginning of the `udp_rcv()` enables PSP packets to be recognised and processed at the very earliest stages of UDP processing. This can reduce the processing delay of PSP packets in the Linux kernel. And by identifying PSP packets early, it can avoid allocating unnecessary resources or performing irrelevant processing steps for non-PSP packets.

Next let me introduce the detail of `psp_rcv()` function. The `psp_rcv()` function is defined as follows:

```
int psp_rcv(struct sk_buff *skb)
```

This function takes a pointer of type `sk_buff`. In this case, it is the `skb` passed in from `udp_rcv()`. And this function returns an integer value indicating the result of the receive processing.

Inside the `psp_rcv()`, it will first perform a series of validity checks to ensure that

the received packet meets the basic requirements of the PSP protocol, such as checking the minimum length of the packet, ensuring the PSP header can be accessed safely, and verifying the PSP version number. After completing the initial checks, the `psp_rcv()` will call the `psp_decrypt()` function to perform the actual decryption operation. Here `psp_decrypt()` will remove the PSP header and trailer as well as decrypt the ciphertext in the `skb`.

The `psp_decrypt()` function is defined as follows:

```
static int psp_decrypt(struct sk_buff *skb)
```

This function takes the `skb` passed in from `psp_rcv()` and returns an integer value indicating the result of the decryption operation.

First the function will calculate the length of the entire PSP section in the `skb`, making sure that the length of the PSP section is greater than the minimum length requirement for a PSP packet.

Then function starts to configure various decryption parameters. The function will first extract the `header->spi`, `header->iv`, and `header->crypt_offset` values from the header. For decryption IV, similar to the encryption process, the function will use `memcpy()` along with the extracted `header->spi` and `header->iv` to construct a 96-bit decryption IV.

For setting decrypted segments, similarly, function uses `sg_init_table()` and `sg_set_buf` to create a three-part scatterlist structure corresponding to the PSP header and plaintext, the data to be decrypted, and the PSP trailer respectively.

Then function will call `psp_derive_key()` to generate the decryption key and use `crypto_aead_setkey()` to set that key into the AEAD decryption context.

Next, the function will use `aead_request_alloc()`, `aead_request_set_ad()` and `aead_request_set_crypt()` one by one to complete AEAD request allocation and setting of the associated data and decryption parameters.

Then `crypto_aead_decrypt()` is called to perform core decryption. This also uses an in-place strategy, where the decrypted plaintext directly overwrites the original ciphertext.

If decryption is successful, the function will reconstructs the packet: removing the PSP header and trailer, adjusting the length of the `skb`, and updating the length field in the UDP header. At this point decryption is completed.

3.5 Initialisation and Configuration

In addition to the core encryption and decryption functions, I also implemented some helper functions for initialisation, resource management and kernel integration. Although these functions are not directly involved in data processing, they are critical to the correct operation of the protocol. They are:

- `static int psp_init_aead(void)`

  This function is used to initialise the AEAD encryption context. It is responsible for allocating the AEAD encryption instance, selecting the AES-GCM algorithm, and setting the size of the authentication tag. Moreover, this function implements strict error handling to ensure that it exits and frees resources in case of a resource allocation failure or misconfiguration.

- `int __init psp_init(void)`

  This function is responsible for the initialisation of the entire protocol. It calls `psp_init_aead()` to set up the cryptographic context and perform necessary error checkings. Its successful execution signals that the PSP protocol is fully initialised in Linux kernel and ready to go.

All these core functions and helper functions are implemented in `net/ipv4/psp_core.c`. After implementing the PSP protocol, in order to integrate it into the Linux kernel, it needs to be properly configured. This involves modifying the kernel's build system, which mainly consists of updates to the Kconfig and Makefile files. In the Kconfig file in the `net/ipv4` directory, I added the following configuration options for PSP:

```
config PSP
    tristate "PSP (PSP Security Protocol) support"
    depends on INET
    help
      Say Y here to enable support for the PSP Security Protocol.
      This protocol provides encryption for UDP traffic.

      To compile this driver as a module, choose M here: the module
      will be called PSP.

      If unsure, say N.
```

This configuration allows PSP to be compiled as part of the kernel, as a loadable module, or disabled altogether.

In the Makefile in the same directory, I added the following line:

```
obj-$(CONFIG_PSP) += psp_core.o
```

This configuration instructs the kernel build system to compile the `psp_core.c` file based on the state of the PSP configuration.

Chapter 4

Performance Evaluation of PSP

In this chapter, I will perform functional testing and performance evaluation of the implemented PSP protocol above. Firstly, I will describe the experiment environment in detail, including the hardware setup and software environment. Secondly, I will communicate using the PSP protocol to verify that the basic functions of the PSP protocol work as designed. This will verify the correctness of the encryption and decryption processes, and the correct handling of the protocol headers and trailers. Next, I will dive into a performance evaluation, comparing the PSP protocol with the TLS protocol. I will measure the performance metrics including throughput, latency, and CPU usage of the protocols under identical conditions for PSP and TLS.

4.1 Experiment Environment

This section will describe the hardware and software used for the experiment in detail.

4.1.1 Hardware Setup

The test environment consists of two virtual machines. They are used to simulate the client and server respectively. These two virtual machines run on the same physical host to ensure consistency and controllability of the test environment. The configurations of each virtual machines are as follows:

- Processor: 8 virtual CPU cores
- Memory: 7GB RAM
- Storage: 60GB

- Network adapter: Virtual network card, configured in NAT mode

4.1.2 Software Environment

Operating system and kernel:

- Virtual software: VMware Workstation 17 Pro
- Linux release: Ubuntu 24.04 LTS
- Linux kernel version: 6.10.0

Compilation toolchain:

- GCC version: 13.2.0
- Make version: 4.3

Network Tools:

- Wireshark: For packets capture and in-depth protocol analysis
- OpenSSL: Employed for TLS protocol configuration and cryptographic operations

4.2 Functional Testing

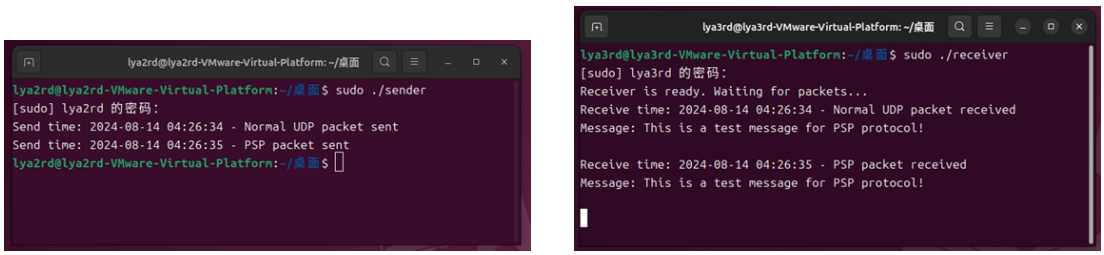
When configuring PSP, I used the built-in method to compile PSP as part of the kernel. Test was performed between the two virtual machines described above, one as the sender (IP address 192.168.75.133) and the other as the receiver (IP address 192.168.75.134).

On the sender side, I wrote and ran the program `sender.c`. This program will send a normal UDP packet and a PSP UDP packet to the receiver side, respectively, with the content ”his is a test message for PSP protocol!”.

On the receiver side, I wrote and ran the program `receiver.c`. This program will receive the UDP packets sent from sender and display the contents of received packets to the console.

It can be seen from Figure 4.1 that the receiver side can correctly receive PSP packet and display the contents.

Next, I used Wireshark to capture those packets. As shown in Figure 4.2, a normal UDP packet had a total length of 82 bytes, including a payload of 40 bytes. The data



(a) Sender side (b) Receiver side

Figure 4.1: Running results in Linux

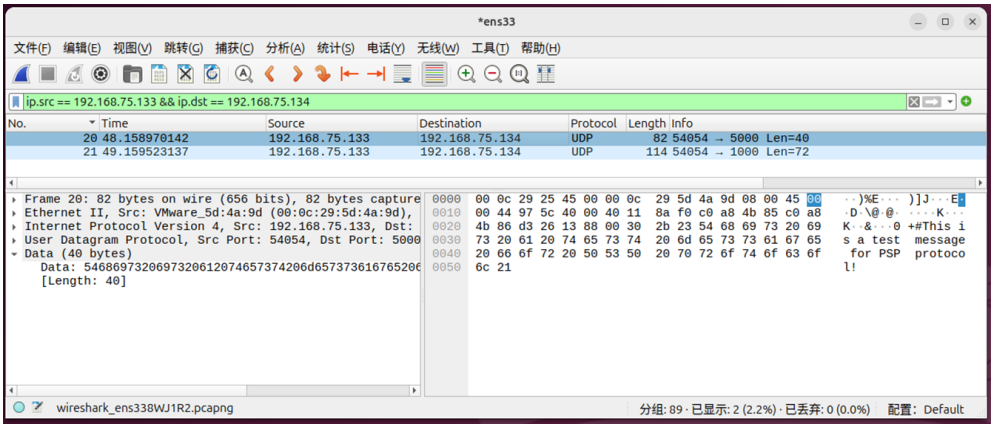


Figure 4.2: Normal UDP packet in wireshark

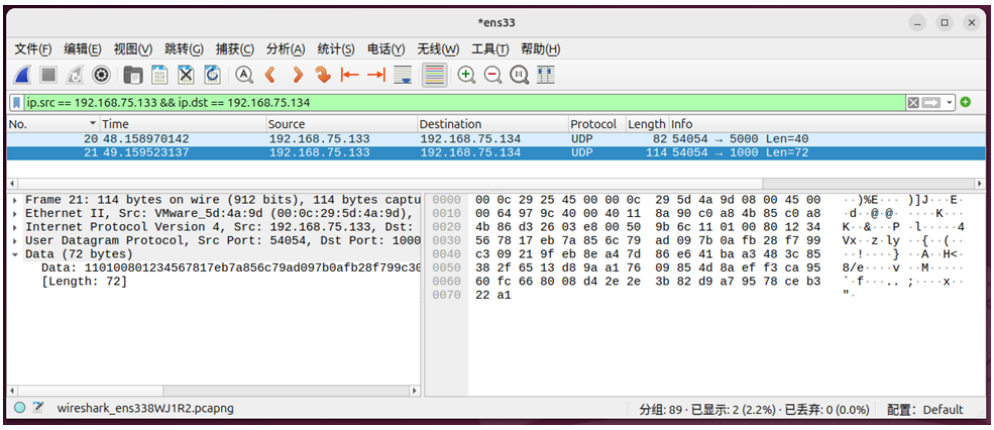


Figure 4.3: PSP packet in wireshark

content within this payload was stored in plaintext form. Figure 4.3 illustrates a PSP packet with a total length of 114 bytes, including a payload of 72 bytes. The additional $72 - 40 = 32$ bytes, compared to the normal UDP packet, correspond precisely to the combined length of the PSP header (16 bytes) and trailer (16 bytes). Observing what is saved in the packet, it can be seen that the data is saved in the packet in the form of ciphertext.

PSP can perform successfully!

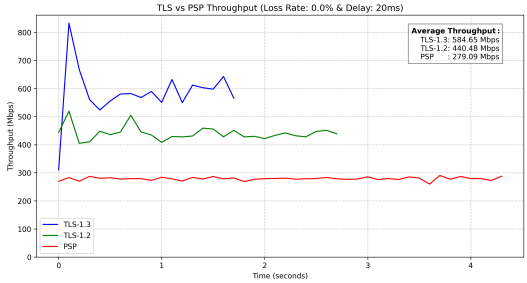
4.3 Performance Evaluation

In order to evaluate the performance of the PSP protocol, I chose to measure the throughput, latency and CPU consumption of the PSP protocol. To be more intuitive, I also tested TLS at the same time to compare their differences. For the TLS implementation, I utilized OpenSSL to do it[18].

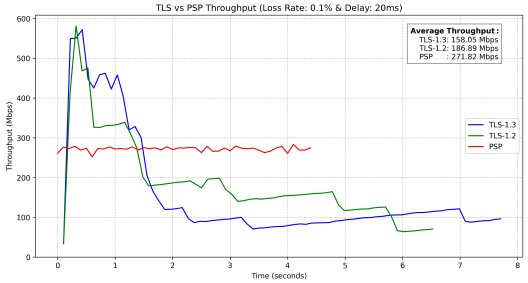
4.3.1 Throughput Evaluation

Here I used PSP protocol, TLS-1.2 and TLS-1.3 protocol respectively to establish three different connections from sender to receiver. Through these connections, I transferred a file of size 145M respectively.

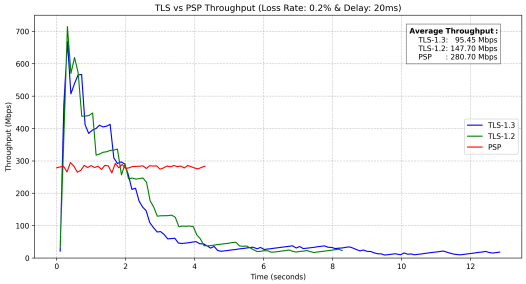
To simulate a real network environment, here I added a delay of 20ms to the sender side. At the receiver side, I recorded the transmission completion time as well as the instantaneous throughput during transmission. I tested them at 0, 0.1%, 0.2% packet loss rate respectively. The following figures show the throughput performance of three protocols at 0, 0.1% and 0.2% loss rates.



(a) 0 packet loss



(b) 0.1% packet loss



(c) 0.2% packet loss

Figure 4.4: Throughput of PSP and TLS in different packet loss

From the Figure 4.4, we can see that at zero packet loss, TLS 1.3 has the best performance with an average throughput of 584.65 Mbps, TLS 1.2 has the second best performance with an average throughput of 440.48 Mbps, while PSP has the worst performance with an average throughput of 279.09 Mbps.

When the packet loss rate reaches 0.1%, the performance of TLS starts to decrease, the average throughput of TLS-1.2 is 186.89Mbps, and TLS-1.3 is 158.05Mbps. On the contrary, the PSP is not affected by the loss of packets, and the average throughput is 271.82Mbps.

When increasing again the packet loss rate to 0.2%, the performance of TLS protocol decreases further, the average throughput of TLS-1.2 is 147.70Mbps, and TLS-1.3's is only 95.45Mbps. However, the throughput of the PSP protocol does not decrease but increase a little bit, with an average throughput of 280.70 Mbps.

In this three cases, PSP completes per transmission in just over 4s. In contrast, the performance of the TLS protocol drops drastically in the presence of packet loss. TLS-1.3 is the most affected. It is able to complete the transmission in 1.8 seconds with no packet loss, but by 0.2% delay, the time of completion reaches to about 13 seconds, which is an increase of 622%.

The results illustrate that PSP is almost unaffected by the packet loss rate. It shows the best adaptability in unstable network environments. Considering the real network environments, especially in the case of long-distance transmissions, there is usually a certain degree of packet loss. In these situations PSP will show a better performance.

To better assess the PSP protocol's resilience to packet loss, I expanded the range of loss rates and conducted additional experiments. Experiment results has been averaged and shown in Figure 4.5.

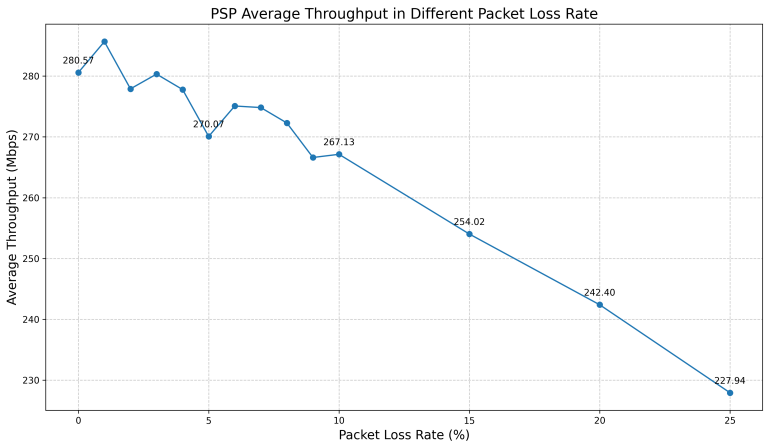


Figure 4.5: Average throughput of PSP in different packet loss

As can be seen from the Figure 4.5, the PSP protocol still demonstrates remarkable performance stability under extreme network conditions. Even in a 25% high packet loss environment, PSP was still able to maintain an average throughput of 227.94 Mbps, only 18.8% lower than in the ideal network condition.

This result is important for big data centres in terms of robustness. It suggests that even under poor network conditions, data synchronisation and backup operations between data centres can remain remarkably efficient. This is of great value in improving the overall reliability and disaster recovery of distributed systems.

4.4 Latency Evaluation

Because the clocks on the receiver side and sender side cannot be synchronised at the microsecond level. In order to accurately reflect latency, I chose to use the time of a packet to and from receiver to represent latency.

In the case of the PSP protocol, for example, an encrypted packet with a payload of 500 bytes will be sent from the sender to the receiver. After receiving the packet, receiver will reply with an acknowledgement packet. When the sender receives an acknowledgement packet it will immediately calculate the time interval between sending the packet and receiving the acknowledgement packet. This time interval represents the latency data. Since the PSP protocol does not design a handshaking mechanism, so here we do not count the handshaking time into the latency.

In order to simulate the real network environment, I set different network delay at both the sender and receiver side. The latency results are shown in the Figure 4.6.

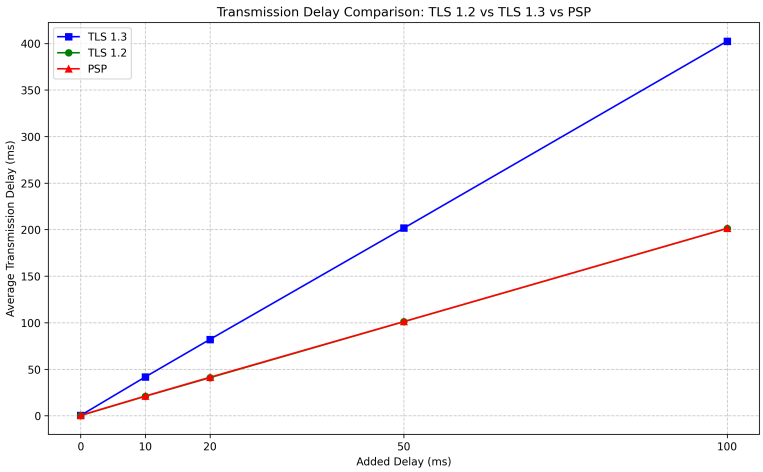


Figure 4.6: Average latency of PSP and TLS in different network delay

(Note that the x-axis shows one-side delay. To obtain the total round-trip delay, these values need to be doubled.)

As can be seen from the Figure 4.6, in the perfect condition where the network delay is zero, the latency of all three is very close to zero.

However, as the network delay increases, the transmission latency of all the protocols shows a linear growth. The latency of TLS-1.3 has the largest increase rate. Its increase rate is about twice as much as the PSP's and TLS-1.2's. Interestingly, the latency value of TLS-1.3 is about twice the sum of the network delay. This may be caused by the TLS-1.3 acknowledgement mechanism.

The PSP protocol and TLS-1.2 have almost the same latency, which is equal to the sum of the network delays at both ends. This indicates that the latency of these two protocol comes almost entirely from the delay in sending. To better compare the time spent on processing inside the protocol, I again compared the latency minus network delay for the PSP and TLS-1.2. This is shown in the Figure 4.7:

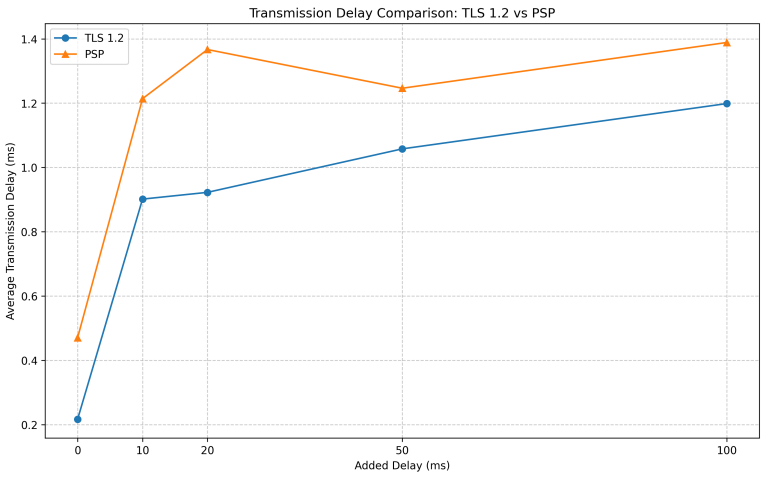


Figure 4.7: Average latency of PSP and TLS-1.2 in different network delay(minus network delay)

After removing the time of network delay, what remains is the time of processing inside the protocol. As we can see from the figure, the overall processing latency of the PSP protocol is slightly higher than TLS-1.2's. The difference is roughly in the range of 0.2 to 0.4 milliseconds. In terms of latency, the performance of TLS-1.2 is higher than PSP.

In order to look more deeply into the processing latency of the PSP protocol, I experimented again and calculated the execution time of each part of the PSP protocol when transmitting a file with 500 bytes payload.

Based on multiple executions and subsequent averaging, the results show that for this particular file, the PSP protocol exhibits an average processing time of 0.06475 ms for sending and 0.01253 ms for receiving. The time taken by each part of PSP is shown as below:

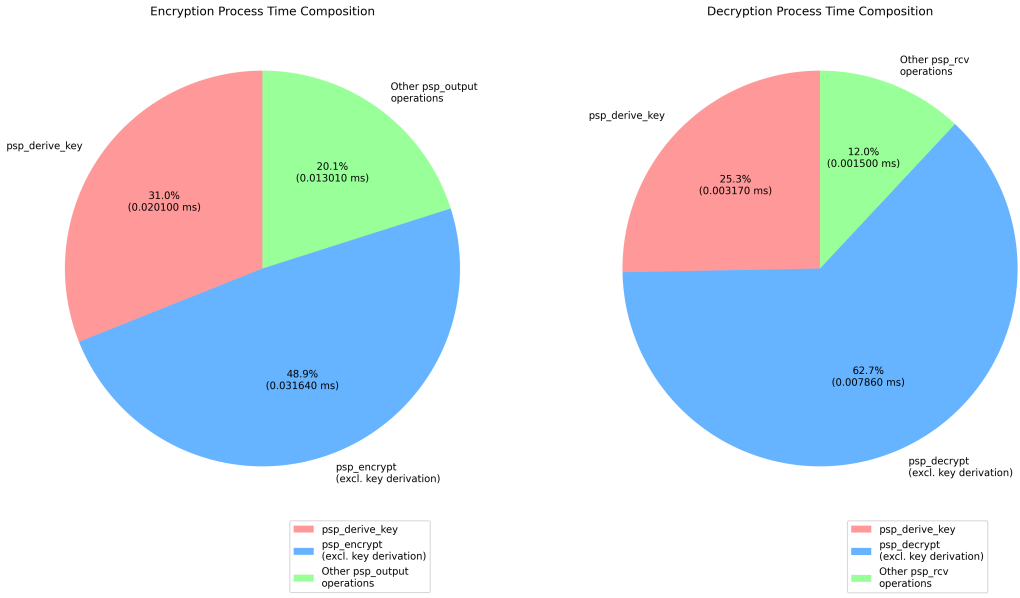


Figure 4.8: Processing time of each part in PSP

As we can see, in both sending and receiving processing, performing encryption and decryption accounts for the largest portion, 48.9% and 62.7%, respectively, corresponding to 0.03164 ms and 0.00786 ms. The second part is the key derivation part, which accounts for 31% and 25.4% respectively, corresponding to 0.0201 ms and 0.00317 ms. And the rest is some processing operations related to sending and receiving.

As discussed in Chapter 2, the PSP protocol's key derivation mechanism is designed to reduce storage pressure. However, this comes at a computational cost. Each key derivation operation consumes approximately 25% of the total PSP processing time, contributing significantly to the overall processing latency.

To illustrate this impact, consider the transmission of a 1GB file with PSP packets carrying 1000 bytes of payload each. The time spent on key derivation operations alone can be calculated as follows:

$$\frac{1,073,741,824}{1000} \times \frac{0.0201 + 0.00317}{1000} \approx 24.97 \text{ seconds} \tag{4.1}$$

This represents a considerable portion of the processing time.

Nevertheless, when comparing the overall performance, the receiving side demonstrates a notable efficiency advantage:

$$\frac{0.06475 - 0.01253}{0.06475} \times 100\% \approx 80.64\% \tag{4.2}$$

This calculation indicates that the receiving side's processing time is 80.64% less than that of the sending side. In other words, the receiving side is significantly more efficient in handling PSP packets. Given that large data centers typically function as receivers, this characteristic of the PSP protocol appears particularly favorable for such environments. The reduced processing time on the receiver can lead to improved overall performance in data center operations.

4.5 CPU Consumption

Since the big data centre often acts as the receiver side. In order to deeply measure the friendliness of the PSP protocol to the receiver side, I chose to measure the CPU consumption of the receiver side when processing PSP packets.

For a clearer comparison, I continuously sent packets to the receiver side using each of the three protocols for a period of time, and then recorded the CPU utilization changes at the receiver side. The CPU usages during the receiving process are shown below:

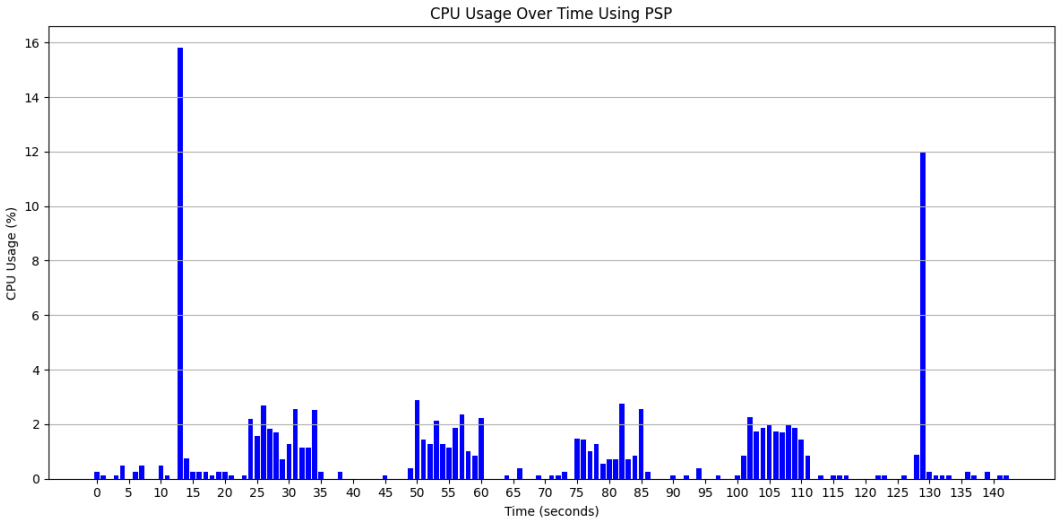


Figure 4.9: CPU usage using PSP

In each experiment, I performed 4 separate transmissions, corresponding to the 4 hump positions in the middle of the figure.

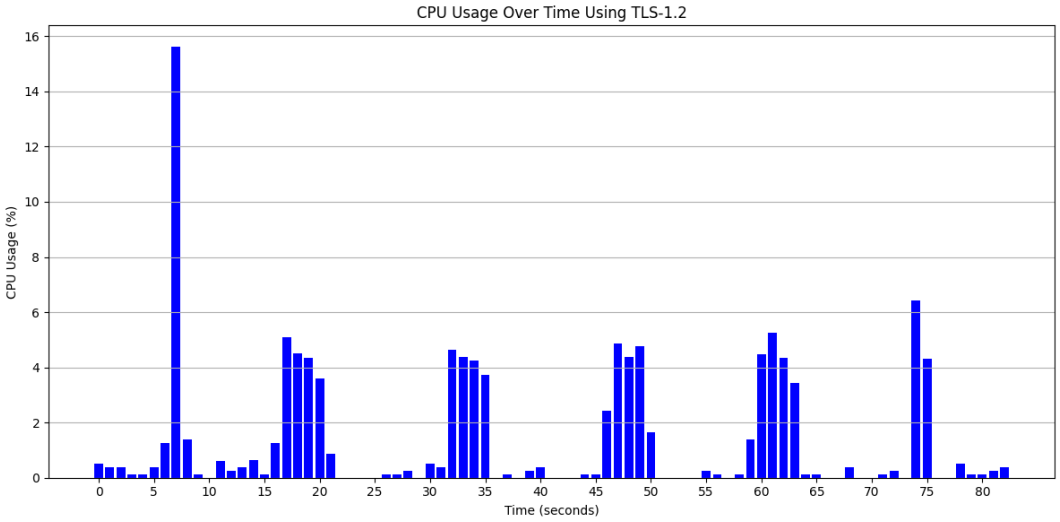


Figure 4.10: CPU usage using TLS-1.2

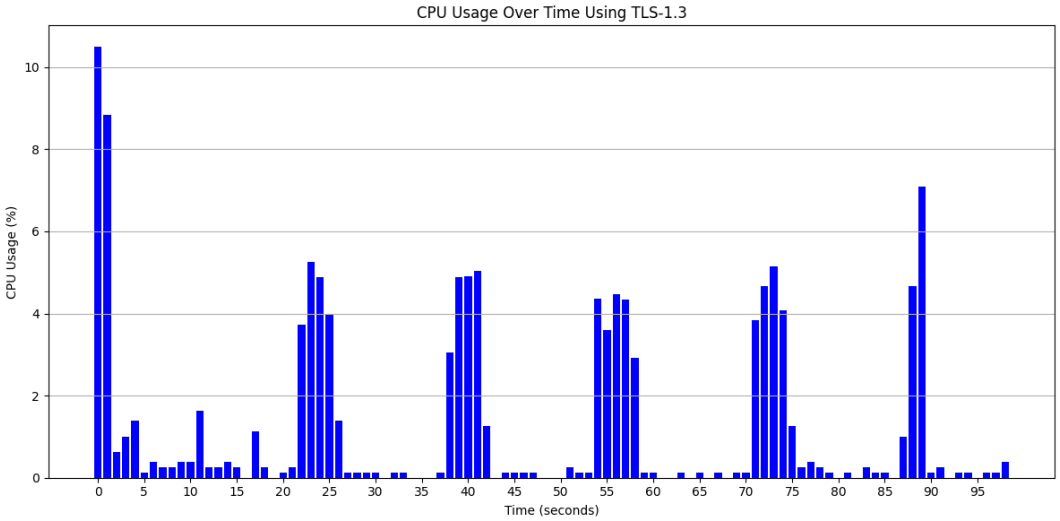


Figure 4.11: CPU usage using TLS-1.3

The results indicate that during the operation of the PSP protocol, the CPU utilization ranges between 1% and 3%. In comparison, TLS-1.2 exhibits a CPU occupancy of approximately 4%, while TLS-1.3 demonstrates the highest CPU utilization at around 5%. Evidently, among the three protocols, PSP consumes the least CPU resources, making it the most receiver-friendly protocol.

However, it's important to note that this experiment was conducted under ideal network conditions. In this scenario, the TLS protocol completed the file transfer more rapidly due to its throughput advantage. The higher CPU utilization observed in TLS might be attributed to its ability to process more packets per unit time. Conversely,

while the PSP protocol demonstrates lower CPU consumption, it requires a longer time to complete the transfer. This trade-off suggests that, in the long term, PSP must be optimized to enhance its throughput capabilities without significantly increasing its CPU usage.

Chapter 5

Conclusion

This paper explores the PSP protocol proposed by Google in 2022. The PSP protocol was proposed by Google in 2022 and was originally designed to address the need for secure and efficient transmission between big data centres. However, although the PSP protocol shows great potential in theory, there is currently a lack of in-depth analysis and evaluation of its performance in real-world environments. This situation not only limits our understanding of the actual efficacy of the PSP protocol, but also creates uncertainty for potential adopters. For any organisation considering implementing the PSP protocol in its infrastructure, a comprehensive understanding of its performance characteristics, resource consumption and possible space for optimisation is essential. Without this critical information, it is difficult to make informed decisions or fully utilise the benefits of PSP.

To fill this gap, this paper implements the PSP protocol in the Linux kernel and designs a series of experiments to fully evaluate the performance of PSP. I followed the PSP specification published by Google implementing a PSP protocol with key derivation, encryption and decryption functions. Based on this, I designed a variety of experiments to simulate the communication scenarios using PSP protocol in a real network environment. My experiments covered several key performance metrics, including:

- **Throughput:** the data transfer rate of PSP at different packet loss rates.
- **Latency:** the overall processing time of PSP.
- **CPU consumption:** the CPU utilization of PSP.

In addition, I also conducted comparative experiments with existing secure transport

protocols (e.g., TLS 1.2 and TLS 1.3) to fully evaluate the strengths and weaknesses of the PSP protocol and the potential areas for improvement.

Through experiments, it is found that the average throughput of PSP protocol without packet loss rate is at 279 Mbps. Moreover, the PSP protocol demonstrates remarkable performance stability in poor network environments. Even in a 25% high packet loss environment, the PSP protocol maintains an average throughput of 227.94 Mbps, which is only 18.8% lower than that in ideal network conditions. Besides, the PSP protocol outperforms TLS-1.3 in terms of latency performance. In a network environment with network delay, the latency of PSP is 50% as much as TLS-1.3's, but slightly inferior to TLS-1.2. In terms of CPU consumption, the PSP protocol has the lowest CPU consumption, with an average CPU consumption of about 40% of TLS-1.3's.

However, I also observed some shortcomings. In terms of throughput, when in a perfect network environment, the average throughput of the PSP is only 54% of TLS's, which is far inferior. I also found that because of the introduction of key derivation mechanism, for each packet there must be a key derivation computation. This calculation took up a quarter of the processing time of the PSP protocol, increasing the processing latency of PSP.

All these experiment results not only reveal the advantages and limitations of PSP protocol, but also highlight the importance of my research work. The work on implementing PSP in the Linux kernel, for researchers, provides a practical platform for in-depth study of PSP. For developers, my implementation can serve as an important reference for deploying PSP protocol in other systems. As for the performance evaluation work, my results provide valuable baseline data for both academia and industry. These can serve as important reference data for future research, and also help to estimate the suitability of the PSP in real-world environments.

However, there are some limitations to my research work. These limitations may have affected the complete evaluation of PSP protocol. One of the most notable limitations is that I was not able to implement the hardware offloading features mentioned in the PSP design. This is due to the hardware limitations of current NIC on my machine. The original design of PSP included the concept of using specialized NIC to accelerate certain operations, such as encryption and key derivation. However, since the existing NIC does not support these specific offloading features, my implementation had to do all the processing on the CPU. This limitation may result in experiment results do not fully reflect the potential of ideal PSP protocol. The lack of hardware offloading may

be one of the reasons why PSP does not perform as well as TLS in some aspects, such as throughput and latency. Future research could explore designing specialised NIC to support PSP protocol and re-evaluating the performance of PSP.

Bibliography

[1] Wenhong Chen, Jeffrey Boase, and Barry Wellman. The global villagers: Comparing internet users and uses around the world. *The Internet in everyday life*, pages 74–113, 2002.

[2] D.D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of tcp processing overhead. *IEEE Communications Magazine*, 27(6):23–29, 1989.

[3] DM Dakhane and RL Pardh. Udp-based multi-stream communication protocol. 2013.

[4] Thomas H Davenport and Jill Dyché. Big data in big companies. *International Institute for Analytics*, 3(1-31), 2013.

[5] Hans Dobbertin, Lars Knudsen, and Matt Robshaw. The cryptanalysis of the aes–a brief survey. In *International Conference on Advanced Encryption Standard*, pages 1–10. Springer, 2004.

[6] Danylo Goncharskyi, Sung Yong Kim, Ahmed Serhrouchni, Pengwenlong Gu, Rida Khatoun, and Joel Hachem. Delay measurement of 0-rtt transport layer security (tls) handshake protocol. In *2022 8th International Conference on Control, Decision and Information Technologies (CoDIT)*, volume 1, pages 1450–1454. IEEE, 2022.

[7] Danylo Goncharskyi, Sung Yong Kim, Ahmed Serhrouchni, Pengwenlong Gu, Rida Khatoun, and Joel Hachem. Delay measurement of 0-rtt transport layer security (tls) handshake protocol. In *2022 8th International Conference on Control, Decision and Information Technologies (CoDIT)*, volume 1, pages 1450–1454, 2022.

[8] Google. Psp architecture specification, 2022.

[9] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks, 2008.

[10] Synergy Research Group. Hyperscale data centers hit the thousand mark; total capacity is doubling every four years, 2024.

[11] Eric He, Jason Leigh, Oliver Yu, and Thomas A DeFanti. Reliable blast udp: Predictable high performance bulk data transfer. In *Proceedings. IEEE International Conference on Cluster Computing*, pages 317–324. IEEE, 2002.

[12] Charles M Kozierok. *The TCP/IP guide: a comprehensive, illustrated Internet protocols reference*. No Starch Press, 2005.

[13] KryptAll. How safe is aes encryption? — advanced encryption standard, 2019.

[14] Hyunwoo Lee, Doowon Kim, and Yonghwi Kwon. Tls 1.3 in practice: How tls 1.3 contributes to the internet. In *Proceedings of the Web Conference 2021*, pages 70–79, 2021.

[15] Qiuyu Lu, June Li, Kai Yuan, Kaipei Liu, Ming Ni, and Jianbo Luo. Udp-rt: A udp-based reliable transmission scheme for power waps. *Computer Networks*, 236:110012, 2023.

[16] David A McGrew and Scott R Fluhrer. Multiple forgery attacks against message authentication codes. *Cryptology ePrint Archive*, 2005.

[17] David A McGrew and John Viega. The security and performance of the galois/-counter mode (gcm) of operation. In *International Conference on Cryptology in India*, pages 343–355. Springer, 2004.

[18] JR Maria Navin, P Suresh, and KR Pradeep. Implementation of openssl api's for tls 1.2 operation. *International Journal of Advanced Computer Research*, 3(3):179, 2013.

[19] Ramzi A Nofal, Nam Tran, Carlos Garcia, Yuhong Liu, and Behnam Dezfouli. A comprehensive empirical analysis of tls handshake and record layer on iot platforms. In *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 61–70, 2019.

[20] Rolf Oppliger. *SSL and TLS: Theory and Practice*. Artech House, 2023.

[21] Joseph Salowey, Abhijit Choudhury, and David McGrew. Aes galois counter mode (gcm) cipher suites for tls. Technical report, 2008.

[22] Jörg Schwenk. A short history of tls. In *Guide to Internet Cryptography: Security Protocols and Real-World Attack Implications*, pages 243–265. Springer, 2022.

[23] Hovav Shacham, Dan Boneh, et al. Fast-track session establishment for tls. In *NDSS*. Citeseer, 2002.

[24] Cody Slingerland. 13 top cloud service providers globally in 2024, 2024.

[25] Prabhu Thiruvasagam, K. Jijo George, Sivabalan Arumugam, and Anand R. Prasad. Ipsec: Performance analysis in ipv4 and ipv6. *Journal of ICT Standardization*, 7(1):61–80, 2019.

[26] Amin Vahdat and Soheil Hassas Yeganeh. Announcing psp's cryptographic hardware offload at scale is now open source, 2022.

[27] John Viega and David McGrew. The use of galois/counter mode (gcm) in ipsec encapsulating security payload (esp). Technical report, 2005.

[28] Peng Wang, Carmine Bianco, Janne Riihijärvi, and Marina Petrova. Implementation and performance evaluation of the quic protocol in linux kernel. In *Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 227–234, 2018.

[29] Sheng Wang. An architecture for the aes-gcm security standard. Master's thesis, University of Waterloo, 2006.

[30] Klaus Wehrle, Frank Pahlke, Hartmut Ritter, Daniel Muller, and Marc Bechler. Linux network architecture, 2004.

[31] Chuah Chai Wen, Edward Dawson, Juan Manuel González Nieto, and Leonie Simpson. A framework for security analysis of key derivation functions. In *Information Security Practice and Experience: 8th International Conference, ISPEC 2012, Hangzhou, China, April 9-12, 2012. Proceedings 8*, pages 199–216. Springer, 2012.

[32] Lisha Ye, Lotfi Mhamdi, and Mounir Hamdi. Efficient udp-based congestion aware transport for data center traffic. In *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, pages 46–51, 2014.

[33] Wei Zhang and Hong Li. Measuring the energy impact of security protocols. CS261N, Vern Paxson, University of California, Berkeley.

Appendix A

Fields in the PSP Header

0: First 32-bit (bits 0-31):

- **Next Header (8 bits):** The protocol type of the PSP packet external header, 6 means TCP, 17 means UDP.
- **Header Extension Length (8 bits):** The length of the PSP header, in units of 8 bytes, excluding the first 8 bytes.
- **Reserved (2 bits):** Reserved bit, set to 0.
- **Crypt Offset (6 bits):** Cryptographic offset, specifies the starting position of the encrypted data, in units of 4 bytes.
- **Sample at Receiver (1 bit):** Used to trigger packet sampling at the receiver.
- **Drop after Sampling (1 bit):** Indicates whether the receiver should drop the packet after sampling.
- **Version (4 bits):** Indicates the PSP version and the encryption/authentication algorithm used.
- **Virtualisation-Cookie-Present (1 bit):** Indicates whether the Virtualization Cookie field is present.
- **Reserved Bit (1 bit):** Set to 0.

1: Security Parameters Index (SPI) (32 bits): Used to identify the Security Association (SA) and derive key.

2-3: Initialisation Vector (IV) (64 bits): Timestamp information used as input to the encryption algorithm.

4-5: Virtualisation Cookie (64-bit, optional): Used for additional identification in virtualised environments.

Appendix B

Source Code for Core PSP Protocol Functions

B.1 PSP Key Derivation Function (psp_derive_key)

```
static int psp_derive_key(u32 spi, u8 *derived_key)
{
    struct crypto_shash *tfm;
    struct shash_desc *desc;
    u8 input[16];
    u8 *master_key;
    int ret;

    tfm = crypto_alloc_shash("cmac(aes)", 0, 0);
    if (IS_ERR(tfm)) {
        return PTR_ERR(tfm);
    }

    desc = kmalloc(sizeof(*desc) + crypto_shash_descsize(tfm),
        GFP_KERNEL);
    if (!desc) {
        crypto_free_shash(tfm);
        return -ENOMEM;
    }

    desc->tfm = tfm;
```

```
    // Select master key based on SPI's MSB
    master_key = (spi & 0x80000000) ? master_key_1 :
       master_key_0;

    // Prepare input for CMAC
    *(u32 *)(&input[0]) = cpu_to_be32(1);          // counter
    *(u32 *)(&input[4]) = cpu_to_be32(0x50763000);  // "Pv0"
    *(u32 *)(&input[8]) = cpu_to_be32(spi);        // (SPI)
    *(u32 *)(&input[12]) = cpu_to_be32(0x80);  // 128 bits

    ret = crypto_shash_setkey(tfm, master_key, 32);
    if (ret) {
        goto out;
    }

    ret = crypto_shash_digest(desc, input, sizeof(input),
       derived_key);

out:
    kfree(desc);
    crypto_free_shash(tfm);
    return ret;
}
```

B.2 PSP Encryption Function (psp_encrypt)

```
static int psp_encrypt(struct sk_buff *skb)
{
    struct psp_header *psp_h;
    struct psp_trailer *psp_t;
    struct aead_request *req;
    struct scatterlist sg[3];
    u8 derived_key[PSP_KEY_SIZE_AES_GCM_128];
    u8 encryption_iv[12];
    int ret;
```

```
    u32 spi = htonl(PSP_FIXED_SPI);
    int data_len, total_len;
    int original_len, data_offset, crypt_offset, plaintext_len
        , ciphertext_len;

    // Calculate data length (excluding IP and UDP headers)
    original_len = skb->len;
    data_offset = skb_network_header_len(skb) + sizeof(struct
        udphdr);
    data_len = original_len - data_offset;
    total_len = data_len + sizeof(struct psp_header) + sizeof(
        struct psp_trailer);

    // Ensure there's enough space to add PSP header and
        trailer
    if (skb_tailroom(skb) < sizeof(struct psp_header) + sizeof
        (struct psp_trailer)) {
         if (pskb_expand_head(skb, 0, sizeof(struct psp_header)
             + sizeof(struct psp_trailer), GFP_ATOMIC))
             return -ENOMEM;
    }
    // Add space for PSP header at the end of skb
    skb_put(skb, sizeof(struct psp_header));

    // Move data
    memmove(skb->data + data_offset + sizeof(struct psp_header
        ),
         skb->data + data_offset,
         original_len - data_offset);

    // Update psp_h pointer to the correct position
    psp_h = (struct psp_header *)(skb->data + data_offset);
    psp_h->next_header = IPPROTO_UDP;
    psp_h->hdr_ext_len = 1;
    psp_h->r = 0;
    psp_h->crypt_offset = 0;
    psp_h->s = 0;
```

```
    psp_h->d = 0;
    psp_h->version = PSP_VERSION_AES_GCM_128;
    psp_h->v = 0;
    psp_h->one = 1;
    psp_h->spi = spi;
    psp_h->iv = cpu_to_be64(ktime_get_real_ns());

    // Add PSP trailer after the data
    psp_t = (struct psp_trailer *)skb_put(skb, sizeof(struct
       psp_trailer));
    memset(psp_t, 0, sizeof(struct psp_trailer));

    // Create IV for encryption
    __be32 be_spi = htonl(PSP_FIXED_SPI);
    __be64 be_timestamp = psp_h->iv;
    memcpy(encryption_iv, &be_spi, sizeof(__be32));  // First
       4 bytes are SPI
    memcpy(encryption_iv + 4, &be_timestamp, sizeof(__be64));
        // Last 8 bytes are timestamp

    // Calculate encryption offset
    crypt_offset = psp_h->crypt_offset;
    plaintext_len = crypt_offset * 4;
    ciphertext_len = original_len - data_offset -
       plaintext_len;

    // Set up scatterlist
    sg_init_table(sg, 3);
    sg_set_buf(&sg[0], psp_h, sizeof(struct psp_header) +
       plaintext_len);
    sg_set_buf(&sg[1], (u8*)psp_h + sizeof(struct psp_header)
       + plaintext_len, ciphertext_len);
    sg_set_buf(&sg[2], psp_t, sizeof(struct psp_trailer));

    // Key derivation and setup
    ret = psp_derive_key(ntohl(spi), derived_key);
    if (ret) {
```

```
        pr_err("PSP: Failed to derive key for encryption\n");
        return ret;
    }
    ret = crypto_aead_setkey(aead, derived_key,
       PSP_KEY_SIZE_AES_GCM_128);
    if (ret) {
        pr_err("PSP: Failed to set derived key for encryption\
           n");
        return ret;
    }
    req = aead_request_alloc(aead, GFP_ATOMIC);
    if (!req)
        return -ENOMEM;
    aead_request_set_ad(req, sizeof(struct psp_header) +
       plaintext_len);
    aead_request_set_crypt(req, sg, sg, ciphertext_len,
       encryption_iv);

    ret = crypto_aead_encrypt(req);
    aead_request_free(req);
    return ret;
}
```

B.3 PSP Decryption Function (psp_decrypt)

```
static int psp_decrypt(struct sk_buff *skb)
{
    struct udphdr *uh = udp_hdr(skb);
    struct psp_header *psp_h;
    struct psp_trailer *psp_t;
    struct aead_request *req;
    struct scatterlist sg[3];
    u8 derived_key[PSP_KEY_SIZE_AES_GCM_128];
    u8 decryption_iv[12];
    int ret;
    u8 crypt_offset;
```

```
int udp_payload_len, total_len, psp_payload_len,
    min_psp_len;
// Calculate total length and UDP payload length
total_len = ntohs(uh->len);
udp_payload_len = ntohs(uh->len) - sizeof(struct udphdr);
// Calculate PSP payload length
psp_payload_len = udp_payload_len - sizeof(struct
    psp_header) - sizeof(struct psp_trailer);
// Calculate minimum PSP packet length (IP header + UDP
    header + PSP header + PSP trailer)
min_psp_len = sizeof(struct udphdr) + sizeof(struct
    psp_header) + sizeof(struct psp_trailer);
// Check length
if (total_len < min_psp_len) {
     pr_err("PSP: Packet too short for PSP structure\n");
     return -EINVAL;
}
// Ensure the entire packet can be safely accessed
if (!pskb_may_pull(skb, min_psp_len)) {
     pr_err("PSP: Cannot pull required data\n");
     return -EINVAL;
}
// Get PSP header and trailer
psp_h = (struct psp_header *)(skb->data + sizeof(struct
    udphdr));
psp_t = (struct psp_trailer *)(skb->data + total_len -
    sizeof(struct psp_trailer));
// Create IV for decryption
__be32 be_spi = psp_h->spi;
__be64 be_timestamp = psp_h->iv;

memcpy(decryption_iv, &be_spi, sizeof(__be32));
memcpy(decryption_iv + 4, &be_timestamp, sizeof(__be64));
// Calculate decryption offset
crypt_offset = psp_h->crypt_offset;
int plaintext_len = crypt_offset * 4;
int ciphertext_len = psp_payload_len - plaintext_len;
```

```
    if (ciphertext_len < 0) {
        pr_err("PSP: Invalid crypt_offset\n");
        return -EINVAL;
    }
    // Set up scatterlist
    sg_init_table(sg, 3);
    sg_set_buf(&sg[0], psp_h, sizeof(struct psp_header) +
       plaintext_len);
    sg_set_buf(&sg[1], (u8*)psp_h + sizeof(struct psp_header)
       + plaintext_len, ciphertext_len);
    sg_set_buf(&sg[2], psp_t, sizeof(struct psp_trailer));
    ret = psp_derive_key(ntohl(psp_h->spi), derived_key);
    if (ret) {
        pr_err("PSP: Failed to derive key for decryption\n");
        return ret;
    }
    ret = crypto_aead_setkey(aead, derived_key,
       PSP_KEY_SIZE_AES_GCM_128);
    if (ret) {
        pr_err("PSP: Failed to set derived key for decryption\
           n");
        return ret;
    }
    req = aead_request_alloc(aead, GFP_ATOMIC);
    if (!req)
        return -ENOMEM;
    aead_request_set_ad(req, sizeof(struct psp_header) +
       plaintext_len);
    aead_request_set_crypt(req, sg, sg, ciphertext_len +
       sizeof(struct psp_header), decryption_iv);
    ret = crypto_aead_decrypt(req);
    aead_request_free(req);
    if (ret) {
        if (ret == -EBADMSG) {
            pr_warn("PSP: ICV verification failed %d\n", ret);
        } else {
```

```
            pr_err("PSP: Decryption failed with error %d\n",
                ret);
        }
    return ret;
    }
    // Move decrypted data to overwrite PSP header
    memmove((void *)psp_h, (void *)(psp_h + 1),
        psp_payload_len);
    // Adjust SKB length to remove PSP header and trailer
    pskb_trim(skb, skb->len - sizeof(struct psp_header) -
        sizeof(struct psp_trailer));
    // Update UDP header
    uh->len = htons(skb->len);
    return 0;
}
```

B.4 PSP Packet Sending Function (psp_output)

```
struct sk_buff *psp_output(struct sock *sk, struct msghdr *msg
    , int len,
                struct flowi4 *fl4, struct ipcm_cookie ipc,
                struct rtable *rt, int (*getfrag)(void *, char
                    *, int, int, int, struct sk_buff *), struct
                    inet_cork cork)
{
    struct sk_buff *skb;
    struct inet_sock *inet = inet_sk(sk);
    int err;
    int psp_extra_len = sizeof(struct psp_header) +
        PSP_AUTH_SIZE_AES_GCM_128;
    // Check if the total length would exceed the limit
    if (len + psp_extra_len > 65535 - sizeof(struct iphdr) -
        sizeof(struct udphdr)) {
        pr_info("PSP exceeds length limit\n");
        return NULL;
    }
```

```
    // Create skb
    skb = ip_make_skb(sk, fl4, getfrag, msg, len, sizeof(
       struct udphdr), &ipc, &rt, &cork, msg->msg_flags);

    if (IS_ERR(skb)) {
        long err = PTR_ERR(skb);
        pr_err("PSP: ip_make_skb failed with error %ld\n", err
           );
        return NULL;
    }



    err = psp_encrypt(skb);  // Use fixed SPI
    if (err)
        goto drop;

    // Update cork structure
    cork.length += psp_extra_len;
    return skb;
drop:
    kfree_skb(skb);
    return NULL;
}
```

B.5 PSP Packet Receiving Function (psp_rcv)

```
int psp_rcv(struct sk_buff *skb)
{
    struct udphdr *uh = udp_hdr(skb);
    struct psp_header *psp_h;
    int min_length;
    // Calculate minimum required length
    min_length = sizeof(struct udphdr) +
                 sizeof(struct psp_header) + sizeof(struct
                    psp_trailer);
    // Check packet length
```

```
    if (skb->len < min_length) {
        pr_info("PSP: Packet too short (%d < %d)\n", skb->len,
            min_length);
        return -EINVAL;
    }
    // Ensure PSP header can be safely accessed
    if (!pskb_may_pull(skb, min_length - sizeof(struct
       psp_trailer))) {
        pr_info("PSP: Cannot pull required data\n");
        return -EINVAL;
    }
    // Get PSP header
    psp_h = (struct psp_header *)(uh + 1);

    if (psp_h->version != PSP_VERSION_AES_GCM_128) {
        pr_info("PSP: Unsupported PSP version %d\n", psp_h->
           version);
        return -EINVAL;
    }
    // Decrypt packet
    int err = psp_decrypt(skb);
    if (err) {
        pr_info("PSP: Decryption failed with error %d\n", err)
           ;
        return err;
    }
    return 0;
}
```