Alternative Approach for Authenticating Subflows of Multipath Transmission Control Protocol using Application Level Key

A dissertation submitted for the Degree of Master of Science in Computer Science

T. N. B. Wijethilake
University of Colombo School of Computing
2019

Declaration

The thesis is my original work and has not been submitted previously for a degree at this or any other university/institute.

To the best of my knowledge it does not contain any material published or written by another person, except as acknowledged in the text.

Student Name: T. N. B. Wijthilake

Registration Number: 2016MCS115

Index Number: 16441157

____

Signature: Date:

This is to certify that this thesis is based on the work of

Mr. T. N. B. Wijethilake

under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by:

Supervisor Name: Dr. Kasun de Zoysa

____

Signature: Date:

Abstract

Multipath Transmission Control Protocol (MPTCP) is an extension to Transmission Control Protocol (TCP) proposed by the Internet Engineering Task Force (IETF). The main intention of MPTCP was to use multiple network interfaces in a single network connection simultaneously. MPTCP create multiple TCP connections, which are known as subflows between two hosts. With the use of multiple connections, the throughput of the connection can be improved. Due to the availability of redundant connections, MPTCP can recover from network connection failures efficiently without noticing the application.

It is clear that there is a number of advantages related to MPTCP. But researchers have identified that there are a considerable amount of security threats related to the connections initiated by MPTCP. These connections are vulnerable to a number of attacks like DoS attacks, flooding attacks, connection hijacking and so on. MPTCP shares a set of keys when establishing the first connection, also known as the first subflow and use these shared keys to authenticate the next subflows created by the hosts. These keys were in plain text format. One of the main reason for the security vulnerabilities is the exchange of keys in plain text format.

A number of solutions were proposed to mitigate these security vulnerabilities. Using an encryption mechanism to secure the keys and changing the header formats are some of them. But this research is inspired by one of the proposed solutions to use external keys to authenticate the subflows. It has proposed to use new socket APIs to obtain the keys from the application level to authenticate the connection. But still, there is no proper implementation of this solution. Therefore as a proof of concept, this research has explored some alternate mechanism to use external keys to authenticate the subflows generated by the MPTCP with minimum modifications to the currently available MPTCP version.

It has conducted a number of experiments on top of MPTCP in order to understand the behavior of the protocol, such as configuring of web server with MPTCP and connecting MPTCP enabled client so on. The final outcome of the research has been implemented on the Linux kernel and several experiments were conducted to examine the robustness of the solution, performance. Finally, the solution has evaluated whether the solution has achieved the requirement to use the external keys to authenticate the subflows.

Acknowledgement

This research would not have been possible without the support and the guidance of helpful people around me. I would like to express my gratitude towards my supervisor Dr. Kasun de Zoysa for the guidance and the support given. It would have been impossible to conduct the research and complete the thesis without the support of my co-supervisor Dr. Kasun Gunawardana.

My special thanks and appreciation goes to Dr. Chamath Keppetiyagama, who helped me to explore new avenues in my research area.

I am highly indebted to the project coordinators and the UCSC staff for the constant support and guidance given.

My heart full of gratitude goes towards my parents, who always helped me to achieve my goals without any hesitation. Finally, I would like to thank all of my friends and colleagues who helped me to complete this task successfully.

Table of Contents

List of Figures

List of Tables

List of Acronyms

DoS	Denial of Service
HMAC	Hash-based Machine Authentication Code
MAC	Machine Authentication Code
MPTCP	Multipath Transmission Control Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security

Chapter 1: Introduction

TCP, the Transmission Control Protocol is one of the major protocols in the transport layer which was introduced in 1981 [1]. The main objective of the TCP was to achieve the reliability of the communication channel between two hosts over a packet switching network. With the advancement of the technology, most of the modern devices such as laptops, mobile phones, and tablet PCs are having more than one network interface, such that Ethernet port, wifi, cellular data connection like 4G/LTE and so on. However, most of the time these devices use only one network interface at any given time and hence, researchers investigated the plausibility of employing the second network interface for different purposes. To increase the throughput and to provide redundant connectivity, it was proposed to use more than one network interface at the same time. To achieve this, an extension to classical TCP was introduced as Multipath TCP (MPTCP) in 2013 [2]. As Figure 1.1 shows, a mobile phone can use both the wifi connection and the mobile data simultaneously to connect to the internet over the Multipath TCP.

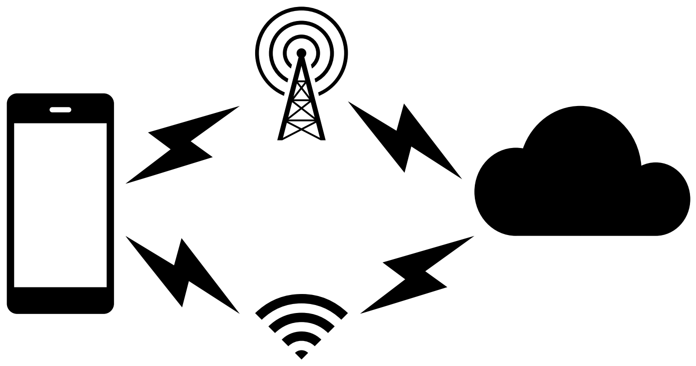


Figure 1.1 : Mobile phone using both WiFi and mobile data simultaneously

1.1 Multipath TCP

Currently, Multipath TCP kernel is available for Linux operating systems, macOS, Android and Apple iOS which can be installed separately. According to my knowledge, only Apple iOS has implemented MPTCP on its Siri voice assistant application [3]. Multipath TCP uses the normal TCP three-way handshake method to create the connections between two hosts. It does not change the currently available TCP protocol stack and the header format. All the data related to MPTCP are sent by using the TCP "option" field available in the TCP header.

Following Figure 1.2 shows the normal TCP layers and Figure 1.3 shows the Multipath TCP layers.

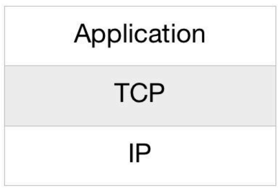


Figure 1.2 : Normal TCP layers

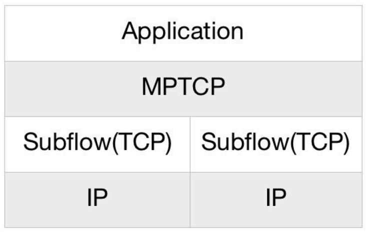


Figure 1.3 : MPTCP layers

To initiate the MPTCP connection between a client and the server, the client sends the normal TCP SYN message with the MP_CAPABLE options included in the TCP header. If the server is also configured with MPTCP, it will reply to the client using SYN_ACK with MP_CAPABLE. And finally, the connection is established with the ACK message from the client.

When sending the MP_CAPABLE SYN message at the beginning, the client sends a key to the server in plain text, as the key of the client. The server also sends a key with the MP_CAPABLE SYN_ACK message in plain text as the key of the server. Finally, with the ACK message, the client sends both the keys to the server to confirm the connection. These shared keys are used to generate the HMAC, which will be later used to authenticate the new sub-flows that would be initialized between the two nodes [4]. In any case, if one of the hosts are not configured with MPTCP, it will automatically be changed into the normal TCP connection. So MPTCP is designed to be backward compatible and independent from the applications which are being executed on the server.

If a client needs to create a new sub-flow with the server, it will send a TCP SYN message to the server with the MP_JOIN option using the client's second interface. In this case, the client sends a token to the server to authenticate itself. This token is a part of the HMAC generated by using the keys shared in the initial key exchange. After sharing the HMACs of keys between the client and the server, MPTCP will create a new sub-flow between them. Other than that there is an option called ADD_ADDR in MPTCP which can be used to advertise the available interfaces of a host to other hosts. Some of the MPTCP options are mentioned in the following Table 1.1 [1].

Creating multiple sub-flows between two hosts requires authentication of one host to another. The proposed authentication mechanism [2] employs plain text key exchange between two hosts over a public network, which opens to many security risks. If an attacker got access to these keys, he or she can create a new sub-flow with the server and even can remove the connection between with the legitimate client and the server [6].

Symbol	Name
MP_CAPABLE	Multipath capable
MP_JOIN	Join connection
DSS	Data sequence signal
ADD_ADDR	Add address
REMOVE_ADDR	Remove address
MP_PRIO	Change subflow priority
MP_FAIL	Fallback
MP_FASTCLOSE	Fast close

Table 1.1: MPTCP options

1.2 Known Exploits

Eavesdropper in the initial handshake is one of the major security threats in MPTCP. Since the initial keys are exchanged in plain text format, the attacker can get access to the keys and subsequently hijack the connection.

ADD_ADDR Attack

The ADD_ADDR option of MPTCP was used to advertise the available network interfaces to other hosts. In an MPTCP enabled client-server environment, the connection is established by sending MP_JOIN message by the client to the server. But in a situation where free interfaces available at the server, the server itself can advertise those interfaces and later client can create a connection with that interface. In the ADD_ADDR attack, the attacker takes advantage of this ADD_ADDR message and creates connections with the server as a legitimate user.

DOS Attack

When two MPTCP capable hosts A and B need to create a new sub-flow with each other, A sends a SYN+MP_JOIN message to B with the token of B generated by A using the keys shared in the initial key exchange. Token authenticates the connections and creates the new sub-flow between A and B. If an attacker can send SYN+MP_JOIN with a valid token to B, it can trigger the joining process.

But there is a limitation to store these half-opened connections per MPTCP connection depending on the implementation. Therefore by send number of MP_JOIN messages with different source addresses can exhaust the receiver. This attack is known as DoS attack on MP_JOIN.

Other

By sending SYN+MP_JOIN messages with different IP addresses and port numbers will consume the server resources which creates the SYN flooding attack. Also, an attacker can present in the path when SYN/JOIN exchange happens and can alter the source address of SYN/JOIN packets. This is the SYN/JOIN attack [7].

There were several solutions proposed for these security vulnerabilities by the Internet Engineering Task Force (IETF) in the RFC 7430 [7] which will be discussed later.

1.3 Problem Statement

The reason for most of the security threats in MPTCP is due to the exchange of keys in plain text. As mentioned earlier, there are a number of solutions proposed by IETF for this problem. Some of the solutions were developed based on the ideas proposed in the RFC7430 [7] and some are developed by combining available security protocols which will be discussed later.

The main focus of this research is to explore a method to use the application level information to authenticate the subflow generation of MPTCP and compare the performance with the original MPTCP kernel to check whether there is a performance degradation with the modifications performed.

The current implementation of MPTCP is not going to consider the application level data in establishing the connections. Therefore it is necessary to find out whether it is important to consider application level data when establishing the connections and how to take the application level information to the MPTCP layer. In order to take application level information to MPTCP, some modifications are needed for the protocol itself. Therefore the main challenge in the research is to find a proper way to take the application or user space data to the kernel space with the minimum modification to the Linux kernel and to the socket APIs. Other than that there should be an authentication mechanism to authenticate the newly created subflows using the external key obtained from the user space.

1.4 Goal and Objectives

One of the main reason for most of the security vulnerabilities of MPTCP connections is the exchange of keys in plain text format, which will later use to authenticate the connections. MPTCPsec [8] and using external keys for the initial handshake [9] are two proposed solutions for this problem which is going to be considered in this research. MPTCPsec has implemented on a Linux kernel by the developers for the research purposes which is still in development stages. So MPTCPsec is not included in the original MPTCP kernel yet. There is no implementation for the proposed solution to use external keys for the initial handshake. Therefore the proposed solution will be influenced by the proposal of using external keys for the initial handshake and the appropriateness of the security solution will be evaluated. Both of the above-mentioned solutions have used different approaches to solve the problem. MPTCPsec has implemented the security within the MPTCP layer and the proposal named as *"Securing the Multipath TCP handshake with external keys"* [9] has proposed to use the keys generated in the application layer.

1.5 Structure of Dissertation

The structure of the dissertation is aligned with the main objective of the research. It is important to get a proper idea about the MPTCP before addressing the research problems mentioned. Therefore the first part of the thesis is mainly focused on the understanding of MPTCP in different scenarios and platforms. The latter part of the document will discuss the design and implementation of the proposed solution. Finally, it will discuss the outcomes and the contribution of the research.

Chapter 2 starts with a broad description of the MPTCP and the implementation of MPTCP. Then it will discuss the available security threats of MPTCP and the proposed solutions.
Chapter 3 is about the research methodology. The research is mainly divided into four stages. Configuration of MPTCP kernel on Linux virtual machine, installing available solutions and implementing TCP sockets, using external keys to secure MPTCP and testing are the main stages. More information about these stages will be discussed in Chapter 3.

Chapter 4 contains details about the proposed solution. The solution is divided into three sections. Obtaining information from the user level to kernel level, transmitting additional information from the client node to the server node and finally, the backward compatibility of the solution are the main sections. Chapter 4 will explain the mechanisms used to perform the above-mentioned requirements.

Chapter 5 contains test results and evaluation. Testing was conducted in four main categories. First, the behavior of MPTCP was tested. Then the developed solution was tested. Third categories were to test the robustness of the proposed solution. Finally, the performance of the proposed solution was tested. Chapter 6 contains the conclusion of the research and future work.

1.6 Summary

Multipath TCP is one of the most promising new technologies in the computer networking domain. Due to the development of technology and the communication requirements, it is necessary to explore new opportunities to increase the throughput of the connection and as well as the redundancy. Though the MPTCP fulfill these requirements, still there are some security issues within the MPTCP protocol. Therefore, researchers around the globe are keen on finding solutions to mitigate the security vulnerabilities of MPTCP. The main intention of this research is to implement a proposed solution to use external keys to secure initial key exchange on the Linux kernel and compare it with the original MPTCP protocol, security wise, and performance wise.

Chapter 2: Background and Literature Review

TCP, the transmission control protocol was designed to create a highly reliable connection between nodes in a packet switching computer network. It is a connection-oriented protocol. TCP was implemented in the layered hierarchy of protocols, in between the application layer and the Internet protocol (IP) layer as shown the Figure 2.1. TCP has one interface to the application layer and another interface to the IP layer.These interfaces have a set of function calls, which are used to open and close the connections between the nodes and also to send and receive data [1].

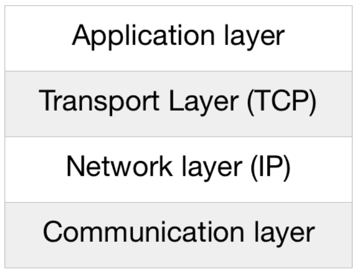


Figure 2.1 : Hierarchy of protocols

One of the limitations in the TCP is, though there are several network interfaces available in the device, it can only use a single interface to connect to the internet. With the Multipath TCP protocol, these limitations were addressed. Two of the key benefits of Multipath TCP mentioned in the RFC 6182 are, to increase the ability to recover the connectivity in a connection failure without failing the end hosts by using multiple paths and to increase the efficiency of the connections by using multiple paths [5].

2.1 Multipath TCP Implementation

Multipath TCP is an extension for the original TCP which tries to increase the redundancy of the connection as well as to increase the throughput. Because of using multiple paths in the connection, it can easily handle the situations of connection failure without affecting the end hosts very much.

The multipath TCP connection is started as a normal TCP connection. Even the application level programs do not have any indication whether it is using Multipath TCP or not. The application opens a normal TCP socket to begin the communication and the Multipath functionality is handled by the implementation itself. First, it will create a normal TCP connection between the hosts and if there are additional interfaces available, the MPTCP protocol will make use of those interfaces to connect the internet as well.

Consider a device A which has two network interfaces as eth0 and eth1 which is trying to connect to a remote server B which as an interface called eth0. All these interfaces should be separately addressable (multi-homed interfaces). First, eth0 of A make a connection with the eth0 of B as shown in Figure 2.2. After successfully establishing the first connections, device A will try to make another connection with B using the eth1 interface as well, which is called as a subflow. This is the most abstract process of the Multipath TCP.

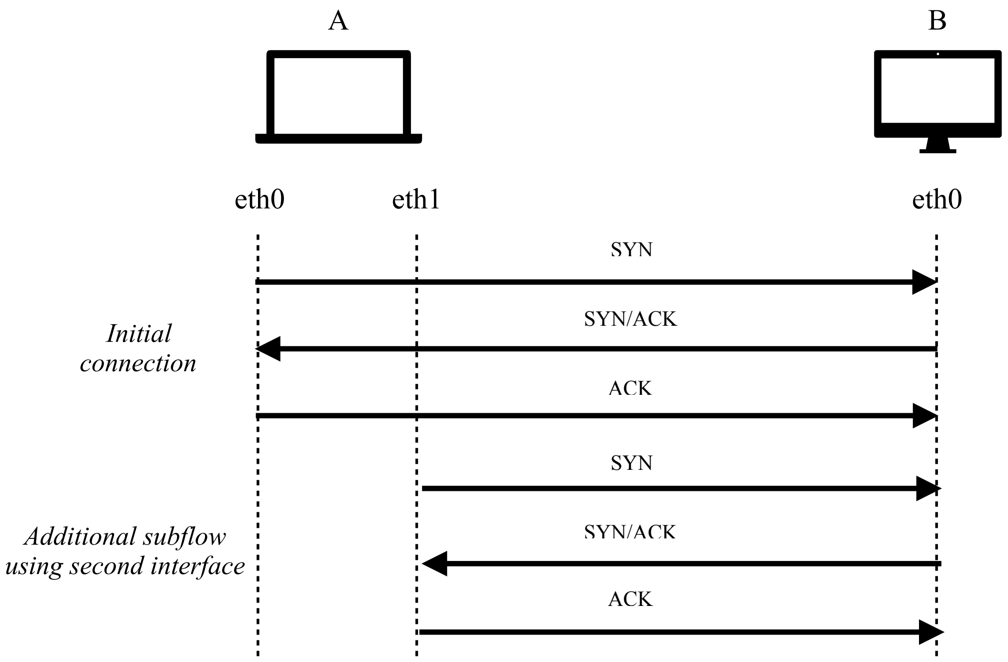
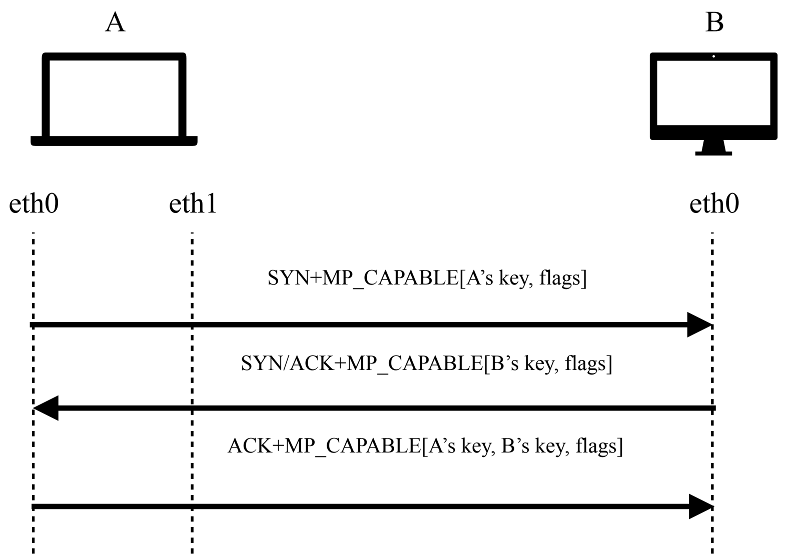


Figure 2.2 : Connection establishment

2.1.1 Initiating Multipath TCP connection

As mentioned in the introduction chapter, there is a number of MPTCP options. These options were inserted into the optional section of the TCP header to perform the MPTCP operations within the TCP connection. Because of maintaining the structure of the original TCP header, MPTCP is totally backward compatible with normal TCP.

In initiating the MPTCP connection, it has to use the MP_CAPABLE option with the normal TCP SYN, SYN/ACK, ACK packets. When the SYN packet is sent from one host to another, it declares that the sender is compatible with MPTCP. If the receiver was also compatible with MPTCP, it will send the MP_CAPABLE option with the SYN/ACK packet. Finally, the sender will confirm the MPTCP connection by sending MP_CAPABLE with the ACK packet. Other than checking the compatibility, MP_CAPABLE will perform another important task. After initiating the MPTCP connection, it has to create additional subflows which related to the initial connection. To do that there has to be a method to authenticate subflows. For that MP_CAPABLE will share key values between the hosts and some flag values. This is a 64bit key value generated by MPTCP for each and every host. In the initial phase, these keys were shared in plain text format. Assume that there are two hosts as A and B. As in the previous example A has two interfaces as eth0 and eth1. As shown in Figure 2.3, A will send the SYN packet to B's eth0 interface with MP_CAPABLE options. This packet contains the key for host A. If the host B is compatible with MPTCP, it will reply with the SYN/ACK packet including the MP_CAPABLE options with the key for host B. Then the host A will confirm that the host B is compatible with MPTCP and send the ACK packet with both the A' key and B'key. Then the connection is established and MPTCP can create other subflows to the connection created.



A
B
eth0
eth1
eth0
SYN+MP_CAPABLE[A's key, flags]
SYN/ACK+MP_CAPABLE[B's key, flags]
ACK+MP_CAPABLE[A's key, B's key, flags]

Figure 2.3 : MP_CAPABLE option

2.1.2 Joining new subflow to the existing connection

As mentioned in the previous section, the main purpose of MP_CAPABLE is to check the MPTCP compatibility and share the key values between the host. MP_JOIN option is used to connect a new subflow to an existing connection. For that, the shared key values are needed. To create a new subflow, the normal TCP three-way handshake method is used. But instead of using the MP_CAPABLE options, the MP_JOIN option is used.

MP_JOIN option has several formats. In the SYN packet, the MP_JOIN option will send a token, address ID and random number to the receiver. The token is the hash value of the key which is shared using the MP_CAPABLE option. Due to the lack of space in the TCP header, the token is truncated into 32bits. Let's assume that the host A in the previous example needs to create a new subflow between B. Then A has to create the hash value of B's key and send it to the B with the MP_JOIN option. By evaluating the hash value, B will authenticate that the request comes by the host A. other than the token, MP_JOIN sends a random value, which can be used to prevent replay attacks and the address ID is to identify the connection. Because in some cases the IP addresses might be replaced by middleboxes. In that case, it can identify the connection by the address ID.

If the token value received by B is correct, then it will send the SYN/ACK packet to A with the MP_JOIN option. In this case, MP_JOIN contains the HMAC of the B's key and random value sent by B. if the token is incorrect, then the connection will reset. After receiving the SYN/ACK by A, again A will respond with the ACK packet including the MP_JOIN option with the HMAC value of A's key. Due to the lack of space, the HMAC values were truncated and send only the first 64bit values. Finally, B will send ACK packet to establish the connection. Then A can communicate with B by using both the interface. Figure 2.4 shows the process of MP_JOIN.

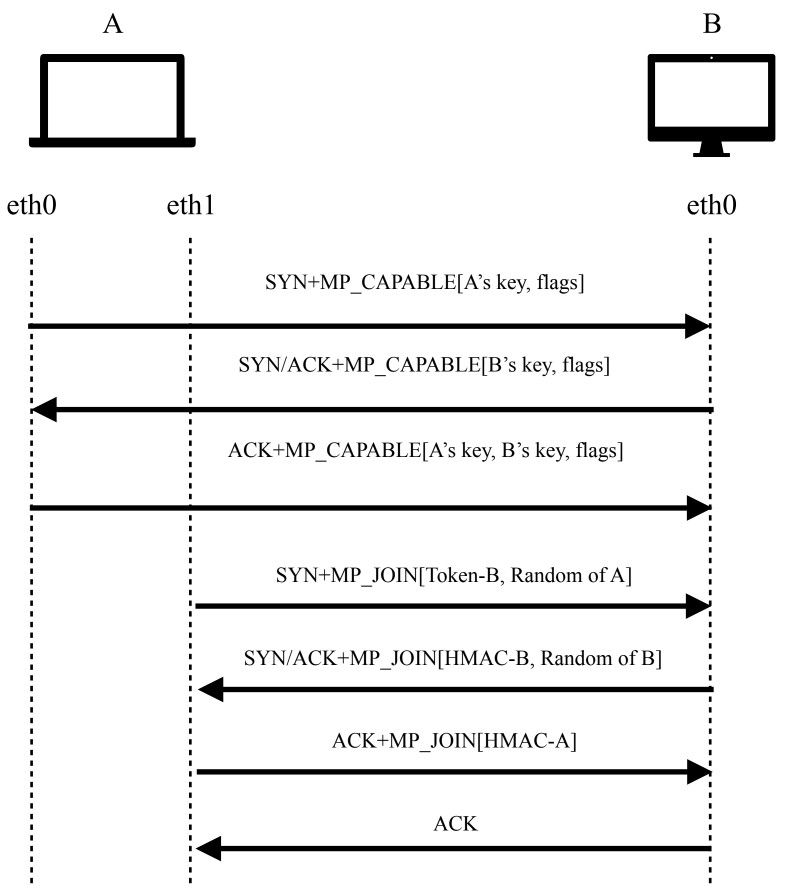

A
B
eth0
eth1
eth0
SYN+MP_CAPABLE[A's key, flags]
SYN/ACK+MP_CAPABLE[B's key, flags]
ACK+MP_CAPABLE[A's key, B's key, flags]
SYN+MP_JOIN[Token-B, Random of A]
SYN/ACK+MP_JOIN[HMAC-B, Random of B]
ACK+MP_JOIN[HMAC-A]
ACK

Figure 2.4 : MP_JOIN option

ADD_ADDR option

ADD_ADDR option can be used to advertise the additional interfaces available by a device. This can be done by a server and the clients who are compatible with MPTCP can send SYN+MP_JOIN packets and create subflows with the particular device. ADD_ADDR option can be used in any MPTCP packet if it has enough space and ADD_ADDR also contains the address ID of the interface also [4].

REMOVE_ADDR option

In case if any of the announced addresses need to be removed, REMOVE_ADDR can be used. It will announce the address ID of the particular interface and after that, all the connections with that address will be terminated.

2.2 Security Analysis and Threats

As mentioned in the Introduction chapter, there are a number of security threats related to MPTCP. These attacks can be categorize into three main groups. *Off the path attacker, partial time on path attacker* and on *path attacker* are them. The *off-path attacker* is an attacker who is not in the middle of the path of the MPTCP connection. Therefore he cannot eavesdrop the packets exchanged in the connection. The second attacker is the *partial time on path attacker*, which has access to the MPTCP connection, but not for the entire period of the connection. The final attacker is the *on-path attacker*, who is on the MPTCP connection, which means he has access to one of the subflows of the connection [6].

There are two other categories of attackers as *eavesdropper* and *active attackers*. Eavesdroppers collect data from the connection while the active attackers try to change the data on the connection [6].

2.2.1 ADD_ADDER attack

This is a man in the middle attack. The attacker uses the ADD_ADDR option of MPTCP to perform this attack [2]. Assume that A and B are two hosts connected via an MPTCP connection, which means they have already shared the security details to create a new subflow. As shown in Figure 2.5, attacker C was trying to create a man in the middle attack. First of all, C has created an ACK packet with the ADD_ADDR option by including B's address as the source address and A's address as the destination address. Which implies that the packet was created at B and forwarded to A. In the ADD_ADDR option there is a space to include the address which is going to advertise. C has used this space to advertise C's address. When A received this ACK packet with ADD_ADDR options, it assumed that this came from real host B and send an SYN+MP_JOIN packet to initiate a new subflow between A and B by using C's address as the destination address. As explained in previous sections, this SYN+MP_JOIN packet contains the hash value of the B which was previously shared in the initial key exchange. When C has received this SYN+ACK packet, it will get the security details which need to create a connection between C and B. C has used these data to create a new SYN+MP_JOIN packet and send it to B, by using C's address as the source address, B's address as the destination address and including the hash value of B which collected from the

SYN+MP_JOIN packet of A. When B received this SYN+MP_JOIN packet, it assumed that this has originated from the real host A and reply to it with SYN/ACK+MP_JOIN with HMAC of A's key and random value of B. C forwarded this details to A and A assumed these details come from B. A replied to this by ACK+MP_JOIN with A's HMAC. Then C forwarded the ACK+MP_JOIN to B and B confirmed the connection between A and B via C as the middleman. ADD_ADDR attack has been categorized as a major threat in MPTCP [4].

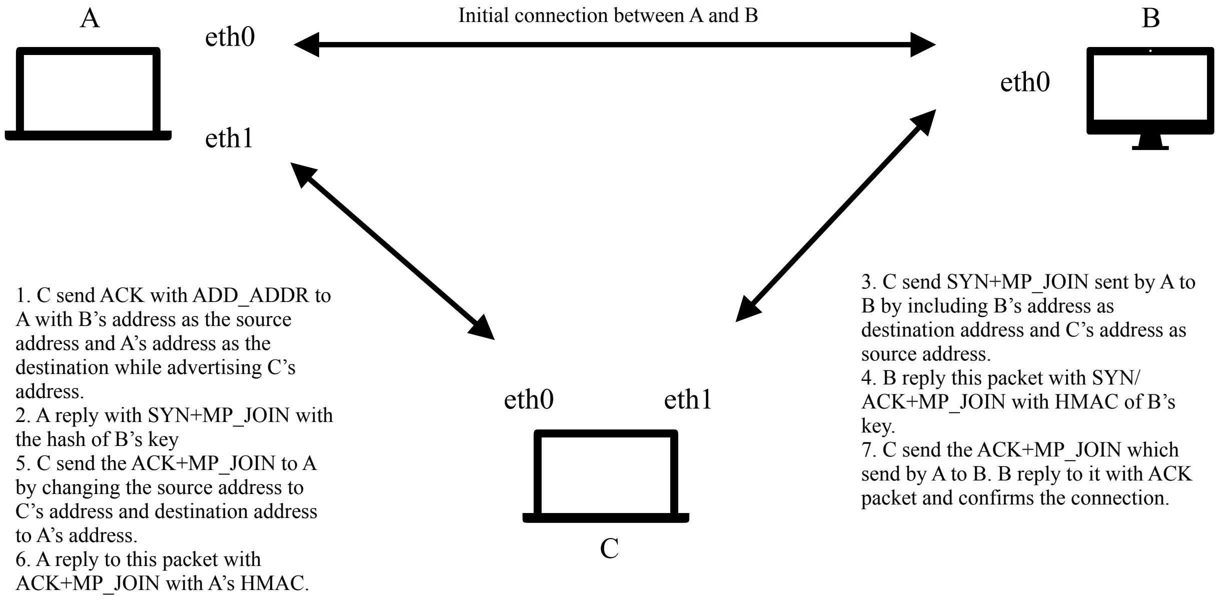

A
eth0
Initial connection between A and B
B
eth0
eth1
1. C send ACK with ADD_ADDR to A with B's address as the source address and A's address as the destination while advertising C's address.
2. A reply with SYN+MP_JOIN with the hash of B's key
5. C send the ACK+MP_JOIN to A by changing the source address to C's address and destination address to A's address.
6. A reply to this packet with ACK+MP_JOIN with A's HMAC.
eth0
eth1
C
3. C send SYN+MP_JOIN sent by A to B by including B's address as destination address and C's address as source address.
4. B reply this packet with SYN/ ACK+MP_JOIN with HMAC of B's key.
7. C send the ACK+MP_JOIN which send by A to B. B reply to it with ACK packet and confirms the connection.

Figure 2.5 : ADD_ADDR attack

2.2.2 DoS attack on MP_JOIN

As explained earlier, the MP_JOIN option is used in MPTCP to create new subflow between two hosts. Which means both the hosts had already shared their security details between them. An attacker can use the MP_JOIN option to exploit attack on MPTCP connection. By sending SYN+MP_JOIN packets to a host with a valid token, the host will open a connection. There is a maximum number of half-open connections can be maintained by a host according to the implementation. When that number is exceeded, the host becomes exhausted. An attacker can send a number of SYN+MP_JOIN packets by changing the source address by using the token. To perform this attack, the attacker should have the 32bit token which was shared in the initial key exchange. Which means that the attacker should eavesdrop the connection when it is initiating [6].

2.2.3 SYN flooding amplification

This is a denial of service attack [16]. Attackers send a number of SYN packets to a port and this made half-open connections. Due to this, there will be not enough resources to create a legitimate connection. In MPTCP, an attacker can send regular SYN packet to open an MPTCP session and perform this attack by sending a number of SYN+MP_JOIN packets with different source addresses. With this process, the server can be exhausted with less cost to the attacker [7].

2.2.4 Eavesdropper in initial key exchange

One of the main security issues in MPTCP is exchanging the keys in plain text format. In this attack, the attacker has collected the keys by listening to the initial key exchange and after that, the attacker can create new subflows using the keys [6]. By using these keys the attacker can totally hijack the connection and even remove the legitimate hosts from the connection.

2.2.5 SYN/JOIN attack

To perform this attack, the attacker should be in the connection path. The source address of the SYN+MP_JOIN packet was altered by the attacker [6]. This can be used to create man in the middle attacks also.

2.3 Proposed Solutions

As mentioned in the above sections, a number of security problems were identified [6] in the MPTCP protocol and high-level solutions were also proposed in RFC 7430 [7]. Using hash chains [10], using SSL [9] and tcpcrypt [17] are some of the proposed solutions. Some of the related solutions are given below.

2.3.1 Asymmetric key exchange

MPTCP shares a set of keys in the initial handshake to authenticate subflows lately. But the main security issue in this process is the above-mentioned keys are in plain text format. Using asymmetric key exchange is one proposed solution for this security issue[11]. Due to the space limitation in the TCP packet, it was difficult to implement asymmetric keys exchange. To overcome this problem Kim and others [11] have proposed to use Elliptic curve Diffie-Helman key exchange [12]. Because of using the Elliptic curve, the space requirement was less, therefore the TCP payload was needed to send the key information. But for the key negotiation part, it was needed four-way handshake rather than the normal TCP three-way handshake.

In the Elliptic key exchange, two values were used to generate the shared key. Consider there are two hosts A and B. A has its own two values of x and y. With the initial SYN+MP_CAPABLE packet, A has sent its x value to B. Then, B replied with ACK+MP_CAPABLE and its x values. Then again B sent its y value using SYN+MP_CAPABLE. Finally, A has replied with ACK+MP_CAPABLE and its y value. Then the initial key exchange ended and both the nodes calculated their shared value using the Diffie-Helman algorithm.

In the MP_JOIN process, A has to send a token to B to authenticate the connection. In this proposed solution, A has sent the token in plain text and HMAC of the token created using the shared key. Thought the token is on plain text, the attacker cannot authenticate it because of the HMAC value created using the shared key.

2.3.2 MPTCPsec

MPTCP secure (MPTCPsec) was proposed to satisfy two main objectives, which are detecting and recovering from packet injection attacks and to protect application level data [8]. To reach these objectives, they have divided the protocol into three phases. Encryption suite negotiation, secure handshake and securing data and control are the three phases. Key negotiation is one of the main activity in the original MPTCP. But in MPTCPsec, they have identified that sending keys in MP_CAPABLE was the main issue. Therefore they have removed the keys from MP_CAPABLE options and modified it by introducing their own option called MPTCPesn[8], which used to negotiate the encryption options between the nodes.

For the secure handshake in MPTCPsec, they have influenced their solution by several technologies such as tcpcrypt[17], TLS[19] and TCP-ENO[18]. They have used the chosen secure protocol and the MPTCPesn to derive the keys and session IDs. Protecting the data is another feature of MPTCPsec. They have used AHEAD algorithms [13] for securing the data. In MPTCP number of TCP options were used. To protect the integrity of this TCP options, MPTCPsec calculated an authentication tag and append that to the TCP payload. Therefore it can be validated whether the middleboxes have altered the content of the TCP options.

2.3.3 ADD_ADDR2

The ADD_ADDR option is used in MPTCP to advertise the available interfaces of a host. By using this option, attackers can exploit man in the middle attacks MPTCP connections. To reduce this vulnerability, a solution was proposed to change the format of the ADD_ADDR option to the ADD_ADDR2 option [4]. The proposed solution is to create a new field in ADD_ADDR option to include a HMAC value. Data of the HMAC is the address ID, advertised IP address and the port number. The key for the HMAC is the key which was shared in the initial key exchange. If the attacker eavesdrops the initial key exchange, still there is a possibility to exploit this attack.

2.3.4 Using external keys to secure MPTCP

Exchanging keys in plain text is one of the main security issues in MPTCP. One of the solutions were proposed for this problem was to use external keys such as SSL or TLS keys to authenticate the MPTCP connection. These SSL or TLS keys are already negotiated in the application layer. The proposed solution [9] has suggested a mechanism to transfer the application layer keys to MPTCP layer two types of sockets. One is MPTCP_ENABLE_APP_KEY, which is used to inform the MPTCP protocol that the application level keys are used to authenticate the connection and MPTCP_KEY is used to provide the application level key to the MPTCP layer.

Another solution was suggested in MPTLS [14], to use TLS with MPTCP to overcome some of the security issues in MPTCP. With some modifications to both the MPTCP and TLS, it has created tighter coupling with MPTCP layer and TLS. It has been evaluated that the TLS is working properly with MPTCP without any performance issues [15].

Chapter 3: Research Methodology

The main focus of the research is to authenticate the subflows generated by the Multipath TCP connection. Before modifying and exploring the opportunities to achieve the main goal of the research, it should have a proper understanding of the behavior of the Multipath TCP protocol itself. Therefore the available version of MPTCP has to be installed and configured. Then the approaches of the other researchers has to be understood by exploring their proposed solutions. With the knowledge and the experience gained by exploring MPTCP and other solutions, the method to implement the proposed solution has to be created. Finally, the implemented solution has to be tested and evaluated.

By considering the above-mentioned requirements, the flow of the research was divided into four stages. Figure 3.1 shows the stages of the research.

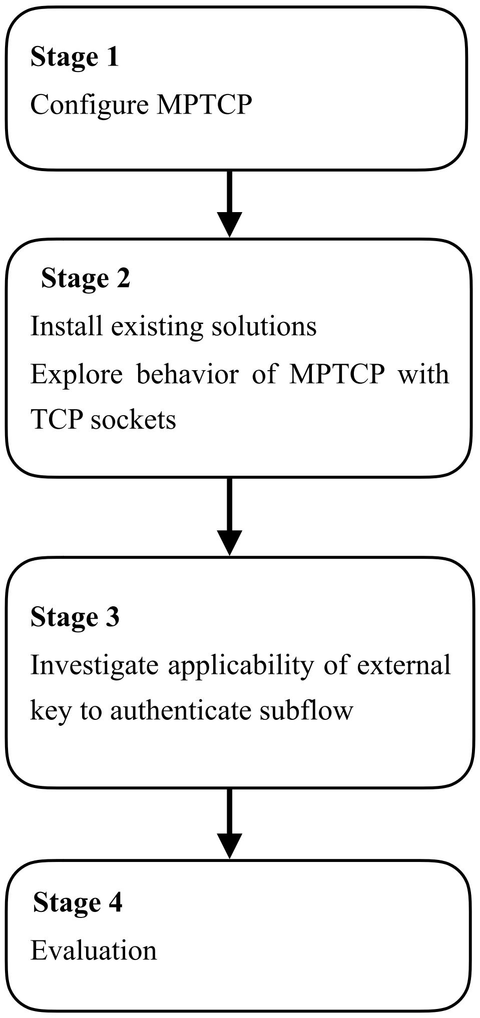

Stage 1
Configure MPTCP
Stage 2
Install existing solutions
Explore behavior of MPTCP with TCP sockets
Stage 3
Investigate applicability of external key to authenticate subflow
Stage 4
Evaluation

Figure 3.1 : Stages of the Research

3.1 Stage 1 - Configure MPTCP

Before going deep into the MPTCP, it is important to understand the behavior of the protocol in the real working environment. Therefore the first step of stage one is to install and configure MPTCP. The stable version of the MPTCP [20] can be installed in the Linux environment using the public apt-repository [21]. The version used for the research is version 0.94 and Ubuntu 16.04 LTS was used as the Linux operating systems to implement MPTCP kernel. After installing the MPTCP kernel, the virtual machine needs to be booted with the MPTCP kernel to get the MPTCP functionalities.

Likewise, two virtual machines were created and connected through the virtual network interface provided by the Virtual Box [22]. After that, one virtual machine can be configured as a server machine by installing the Apache server [23] and a simple website can be hosted. The second virtual machine can be configured as the client machine by including two network interfaces connected to the same virtual network created by the Virtual Box. The hosted website can be requested using the client virtual machine. Because both the virtual machines are configured with MPTCP, a Multipath TCP connection will be established between the two virtual machines. The packets transferred between two machines can be captured using Wireshark and the results are presented in Chapter 5.

It was important to check whether the TLS or SSL is compatible with MPTCP [15]. TLS was configured on the server as the second step of the first stage and the same client-server experiment was conducted to capture the packets using Wireshark. The results were discussed in Chapter 5.

The source file of the MPTCP implementation can be downloaded from the Git repository of the developers [24]. The third step of the first stage is to get familiar with the code of MPTCP and compiling and installing the kernel. Before compiling the code, some configurations need to be done with the kernel and it is not as straightforward as installing from the apt-repository. The same client-server experiment can be done with the compiled and installed kernel to observe the behavior of MPTCP.

One of the commercially available implementations of MPTCP is in the Apple iOS [3]. Network packets transferred on the Apple iPhone can be captured using the Apple Xcode application and some inbuilt command line tools. The final step of the first stage is to capture these packets and observe the behavior of MPTCP on iPhones. These captured packets are analyzed using Wireshark and the results are in Chapter 5.

3.2 Stage 2 - Install existing solutions and Explore behavior of MPTCP with TCP sockets

The first step of the second stage is to install the currently available solution on Linux environment and observe the behavior of them. In this research, the MPTCPsec is considered as the available solution and the most recent implantation can be downloaded from the BitBucket of the developers [25]. In this experiment, the latest version available on the BiTBucket was used which was committed on the 15th of January 2017. MPTCPsec kernel was compiled and installed on Ubuntu virtual machine and the client-server experiment explained in stage one was used to observe the behavior of the MPTCPsec implementation.

The second step of the second stage is to conduct some experiments with TCP sockets in the MPTCP environment. Different kind of TCP sockets was implemented using C language and executed on both the client and server machines to observer the behavior of MPTCP with normal TCP sockets.

3.3 Stage 3 - Investigate applicability of external key to authenticate subflow

The main intention of this research is to explore whether the external keys can be used to authenticate the subflows generated by the MPTCP protocol. For that, the behavior of the MP_JOIN option has to be understood and the MPTCP kernel needs to be modified to perform the necessary operations.

The first step of the third stage is to identify the behavior of the MP_JOIN option. For that, the MPTCP kernel has to be explored and need to identify the functions in the Linux which used to authenticate the newly created subflows.

The second step is to identify the opportunities or methods to take the user level key to the kernel level. If it is possible to take the user level key to the kernel level, then it can be used to authenticate the subflows. The third step is to explore a mechanism to authenticate the subflows by using the external keys which were taken from the user space and to send the additional information from the client side to the server side.

3.4 Stage 4 - Evaluation

The modified version of the MPTCP kernel needs to be installed in virtual machines and the client-server experiment needs to be done using TCP sockets to observe the behavior of the protocol. The performance of the proposed solution has to be compared with the original MPTCP protocol implementation to check whether there is performance degradation.

Chapter 4: Proposed Solution

As explained in the previous sections, the keys shared in plain text format are used to generate the tokens which are used to authenticate the subflows. Because of sharing the keys in plain text format it is open to a number of security threats. The main objective of this research is to explore the applicability of using external keys to authenticate the subflows created by MPTCP which eventually reduces the opportunity for the attackers to hijack the connection.

Before discussing the technical details of the proposed solution, it is more appropriate to understand the stages of the solution in a descriptive manner. As discussed in Chapter 2, when initiating the first connection between two hosts, MPTCP has used the MP_CAPABLE option. With this option, it had confirmed whether both the hosts are compatible with MPTCP and shared the keys which need to authenticate the next subflows. MP_JOIN options are used to connect the second subflow to the main connection by using the shared keys to authenticate. The solution proposed in this research is to use external keys to authenticate the second subflows generated by MPTCP. In this case, there are two assumptions made as given below.

- Both the hosts has to be agreed on the external keys before initiating the second subflow.
- The external keys have to be secure.

As an example, TLS keys can be considered as the external keys, because both the parties have agreed on the keys and the keys are already secured. But within this research, it is not going to consider the external key agreement mechanisms. Figure 4.1 shows the abstract picture of the solution proposed by this research.

The functionalists of the MPTCP are included in the Kernel of the operating system. Kernel of the operating system contains most of the basic operations related to the control and the communication of the hardware components connected. All the other services of the operating system are built on top of the kernel. When making modifications to the components of MPTCP means that eventually making modifications to the kernel of the operating system.

Obviously, this is not an easy task to perform and it should have a considerable amount of technical knowledge to work with the Kernel of the operating system.

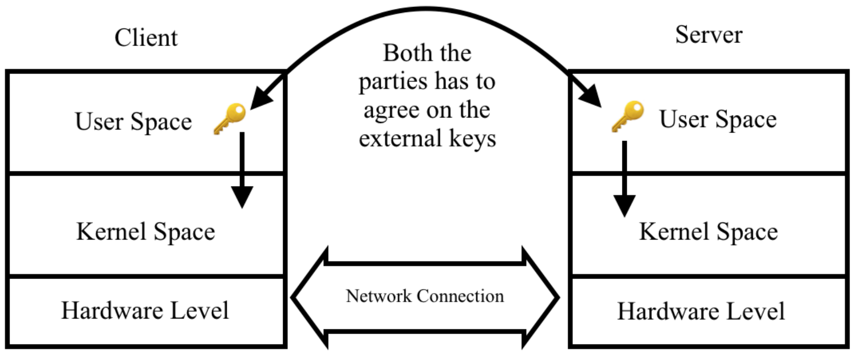


Figure 4.1 : Kernel level, User level and External keys

Even though the operating system has used the functions in the Kernel, generally the kernel level of the operating system is not directly accessible by the end users of the computer system. The normal users of the computers are interacting with the programs on the application level. These programs and applications eventually use the functions in the Kernel level via the operating system to execute the operations on the hardware level of the computer system. Therefore it is clear that the application level or the user space which the end user is interacting is external when considering from the Kernel level where the functions of MPTCP has implemented.

The external key means the secret shared between two hosts which obtained from the application level or the user space. As shown in Figure 4.1, this key has to be transferred from the user space to the kernel space. After that, the keys have to be used in the kernel level to authenticate the subflows. Therefore in this research, transferring the key values from the application level to the kernel level has identified as one of the main tasks.

When the external keys are available in the kernel level, it can be used to generate the authenticating material. The external key itself cannot be used as the authentication material. However, it is possible to achieve an extra level of security when the external key is combined with another component. Therefore generating the authentication material is one of the most important steps identified in this proposed solution.

After generating the authentication material on the client of the connection, it has to be sent to the server for the authentication purpose. The mechanism of sending additional information from one side of the connection to the other is more challenging, due to the limited space of the TCP header and the MPTCP options. It has to find a proper method to transfer the additional authentication material from client end to the server end.

Finally, this research proposes to use the authentication material transferred from the client to the server to validate the connection. If the authentication material sent from the client is identified as valid, the server can accept the newly created subflow and join it to the available connection. If not the subflow has to be dropped.

As explained in the above paragraphs, the methodology proposed in this research to authenticate the subflows using external keys can be categorized into four main tasks as mentioned below. From here onwards more technical aspects of the proposed solution are discussed based on the main four tasks mentioned below.

- Send the external key from the user space to the kernel space.
- Generate the authentication material using the external key obtained from the user space.
- Send the authentication material to the server from the client.
- Authenticate the subflow using the authentication material and if it is not a legitimate connection, then subflow has to be dropped.

As mentioned in the previous sections, MPTCP has used different MPTCP options to initiate the connection between two hosts. MP_CAPABLE option is used to check whether both the hosts are compatible with MPTCP and also the MP_CAPABLE option is used to exchange the authentication keys in the plain text format. MP_JOIN option is used to initiate the second subflow between the two hosts while using the keys shared with the MP_CAPABLE option. Though the keys were shared in the MP_CAPABLE option, MP_JOIN has used these keys to authenticate the newly created subflows. As explained earlier, the main focus of this research is to authenticate the subflows by using the external keys from the user space to provide an extra level of security. Therefore it is better to have a clear idea about the behavior of the MP_JOIN, before making any changes to the MPTCP operations.

When initiating the normal TCP connection, it has used the well known TCP three ways handshake with SYN, SYN/ACK and ACK packets. MPTCP also has used these three packets to initiate the MPTCP connection, but it has included the MPTCP options in the TCP options space. According to the proposed solution of this research, the authentication should perform when initiating the second subflow. For that, the MPTCP has used the MP_JOIN option. In order to implement the solution proposed by this research, the operation of the MP_JOIN option has to be properly understood.

Though it has described in a conceptual manner in the thesis, all these packet generations were defined in a number of functions in the Kernel of the operating system. Figure 4.2 shows the functions called in the Linux Kernel when creating the SYN packet in MPTCP connection [30]. SYN packet is created in the client and it has to be received by the server to initiate the connection.

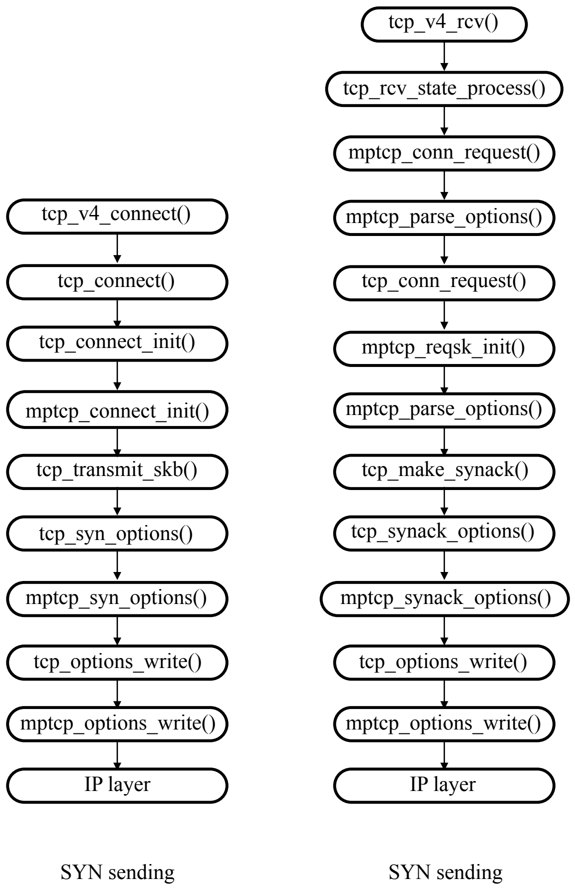


Figure 4.2 : Function calls in MPTCP

As shown in Figure 4.2 there are a number of functions defined in the Linux kernel to perform the necessary operations to initiate the MPTCP connection. The most important Kernel functions have to be identified which are relevant for the generation of SYN, SYN/ACK and ACK packets used in the TCP handshake. Not all the functions in Figure 4.2 are relevant for this research, but the identified functions are discussed later.

The *mptcp_syn_options*() function defined on the *mptcp_output.c* file of the MPTCP kernel is one of the functions identified to be modified in order to implement the proposed solution. This function is used to set the parameters which are necessary to create MPTCP subflows including the token values. The *mptcp_syn_options*() function is executed in the client side, which will need to modify in order to include the authentication materials generate using external keys in the process of authentication.

The *mptcp_parse_request*() is the next function which was identified as an important function to modify in this process. As shown in Figure 4.2, this function is executed on the server end which used to obtain the information sent from the client end. Within this *mptcp_parse_request*() all the information in the TPC SYN packet are assigned in the local data structure at the server end. Therefore this function can be modified to obtain the authentication material send for the client end and validate it to initiate the second subflow of MPTCP connection.

There was two major challenges in this research, which was to send additional information from client side to the server included in MPTCP options and to obtain the key from the user level to the kernel level which has to be used to authenticate the MPTCP subflow. For that, several avenues were identified and explored to recognize the best solution which can be implemented with the minimum modification to the kernel. Figure 4.3 shows the different approaches followed in order to send additional information in MPTCP options and Figure 4.4 show the methods explored to find a better way to transfer the key value from the user space to the kernel space.

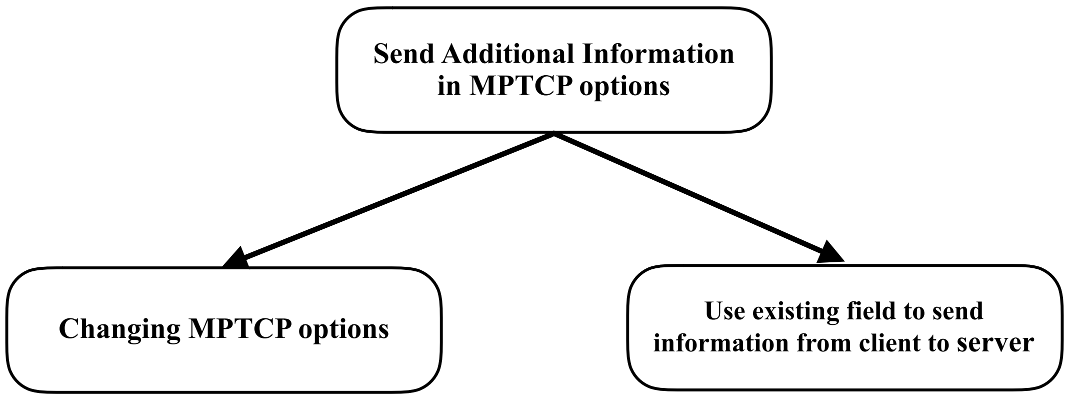

Send Additional Information in MPTCP options
Changing MPTCP options
Use existing field to send information from client to server

Figure 4.3 : Sending Additional Information

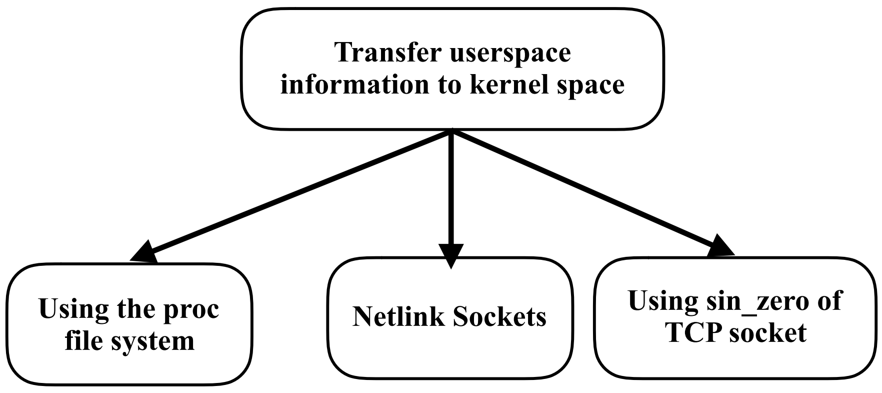

Transfer userspace information to kernel space
Using the proc file system
Netlink Sockets
Using sin_zero of TCP socket

Figure 4.4 : Transfer userspace information to kernel space

4.1 Additional information from client to server

As shown in Figure 4.3, two approaches were explored to identify the better way to send additional information from client to server with minimum modifications to the kernel.

4.1.1 Changing MPTCP options

As explained before, the interested packet in this research is the SYN packet of the second subflow. Figure 4.5 shows the MP_JOIN option for the SYN packet. It has several fields such as kind, length, subtypes, several flags, address id, receivers token and senders random number. The first approach is to add a new field in the MP_JOIN option of the SYN packet to send data from the client to the server. The *mp_join* struct was defined at the *mptcp.h* header and when inserting a new field with 32-bit size, it clashes with some other MPTCP option packet size definitions. Therefore when inserting a new field in the header options, most of the parts of the MPTCP kernel needs to be modified. Though the main idea of the research is to prove the concept of using external keys in the authentication process, an alternative solution was explored with fewer modifications to the Linux kernel.

Kind	Lenght	Subtype		Address ID
Receiver’s Token (32bit)				
Senders Random Number (32bit)				

Figure 4.5 : MP_JOIN options of SYN packet

4.1.2 Use existing fields to send information from client to server

As explained in previous sections, the proposed solution of this research is to send the authentication material from client side to the server side by combining with the key obtained from the user space. In the considered implementation of the MPTCP, it will send the token of the receiver with the MP_JOIN SYN packet. Theoretically, this token should know by the server, because it is the key of itself. So the client sends this token to the server to authenticate itself.

Therefore in the proposed solution, this token value is XORed with the key obtained from the user space and use the space defined for the token in the MP_JOIN option to send the XORed token to the server from the client to authenticate the second subflow. For the proof of concept, it has only computed the XOR value rather than calculating complex HMAC values. With this approach, no additional space was needed in the MP_JOIN option. Therefore within the limited space in the TCP options, this solution was implemented successfully.

4.2 Transfer user space information to kernel space

The biggest and the most time consumed challenge in the research was to identify the proper method to obtain data from user space to the kernel space which can be used to authenticate the subflows in MPTCP. Several avenues were identified as shown in Figure 4.4 and explored to pick out the best possible solution with fewer modifications to the kernel.

4.2.1 Using the proc file system

Proc is a pseudo file system in the Linux operating systems which can be accessed from /proc [27]. This is an interface to the kernel data structure and most of the files in the proc directory are read-only. Some of them are writable and can be used to modify kernel variable. With this approach, a new proc directory has to be created in the */proc* directory and the key value has to be written in the newly created proc directory. This can be done from the user space. After that, this value has to be accessed by the kernel file. The problems encountered were that this proc directory generation and value assigning has to be done before the invoking of TCP sockets. Other than that the value in the proc file has to be read by the kernel. For that, a separate kernel function has to be created. By using that function the key value on the proc file has to be written to a kernel variable. Which means when the kernel initiating a TCP connection, it has to read proc files and assign the values to the variables.

4.2.2 Netlink Sockets

Netlink [28] is a Linux kernel interface which can be used to communicate between kernel space and the user space, and also between different user processes also. With this method, two programs need to be executed. One program should be in the user space and the other program should be in the kernel space. This will create a connection between user space and the kernel space, and transfer data from user space to the kernel space.

4.4.3 Using sin_zero of TCP socket

The *sockaddr_in* is a data structure in the TCP sockets. This data structure contains the necessary information to create a TCP connection between two hosts. Protocol, port number and address are some of the information contains in the *sockaddr_in* data structure. Other than that there is another char array called *sin_zero* which is used as padding [29]. This space is not used by the sockets when creating the connections. Therefore, theoretically, this space can be used to transfer data from user space to the kernel space, if it is not dropped when the information is transferred from user space to kernel space. This was further explored to identify the behavior of the *sin_zero* variable and tracked the functions which transfer the data from user space to kernel space.

Compared to other solutions, using the *sin_zero* easier to send data from the user space. Char value can be easily copied to the *sin_zero* character array when creating the TCP socket. Therefore no need to customize the socket APIs. But the challenge was to retrieve the data from the kernel space. Theoretically, the *sin_zero* data should be received by the kernel space, if it was not dropped by the system calls.

Two methods were tried to obtain the data from *sin_zero* character array from the kernel space. One method was to explore the system call functions of *socket.c* to retrieve the data from the *sockaddr_in* data structure. The other method was to use the *inet* functions of *af_inet.c* to retrieve the data. Implementation was straight forward when using the *inet* functions of *af_inet.c* rather than editing the system call functions of socket.c. Therefore the *inet_bind()* function and the *__inet_stream_connect()* functions were modified to retrieve the data from *sin_zero*. More information about the implementation and the functions mentioned are given in Appendix A.

4.3 Backward compatibility

The backward compatibility is one of the important features in MPTCP. Which means if the host machines were not compatible with MPTCP, it will automatically change into the original TCP connection. Therefore the proposed solution in this research also has to be backward compatible. Which means if any of the machines was not configured with the proposed solution, it should use the normal MPTCP authentication mechanism.

To achieve this requirement, slight modifications for the code has to be done. It has to check whether the *sin_zero* value is set from the user level or not. If the value is set, it has to use the proposed solution and if not it has to use the original MPTCP authentication mechanism.

All the code segments and functions discussed in this section are further described in Appendix A.

4.4 Summary

MPTCP has used a set of keys to authenticate the subflows. The main security issue was that these keys were in plain text format. As described in this chapter the proposed solution is to use an external key to authenticate the subflows.

In order to achieve this goal, there were several challenges to face, such as obtaining information from the user level and transfer them to the kernel level, creating the authentication material, sending the authentication material from client to the server and finally authenticate the connection.

The user space key was transferred to the kernel space by using the *sin_zero* character array of the *sockaddr_in* data structure of TCP socket and the data was obtained by the kernel space using the *inet* functions of *at_inet.c* with minimum modifications to the existing kernel implementations. The authentication material was generated by XORing the token value and the external keys. This authentication material was sent to the server using the available token space in the SYN+MP_JOIN packet and the authentication material was validated at the server. Figure 4.6 shows the proposed solution.

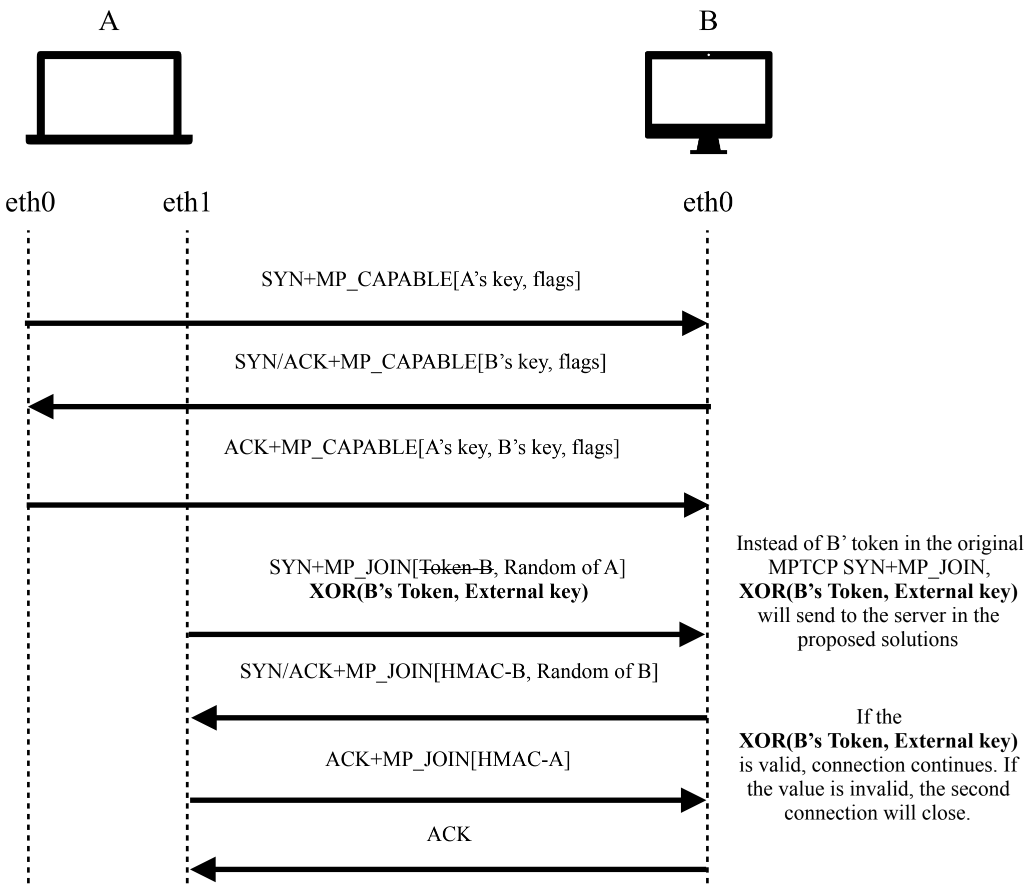

A
B
eth0
eth1
eth0
SYN+MP_CAPABLE[A's key, flags]
SYN/ACK+MP_CAPABLE[B's key, flags]
ACK+MP_CAPABLE[A's key, B's key, flags]
SYN+MP_JOIN[~~Token-B~~, Random of A]
XOR(B's Token, External key)
SYN/ACK+MP_JOIN[HMAC-B, Random of B]
ACK+MP_JOIN[HMAC-A]
ACK
Instead of B' token in the original MPTCP SYN+MP_JOIN, **XOR(B's Token, External key)** will send to the server in the proposed solutions
If the **XOR(B's Token, External key)** is valid, connection continues. If the value is invalid, the second connection will close.

Figure 4.6 : Proposed solution

Chapter 5: Evaluation and Results

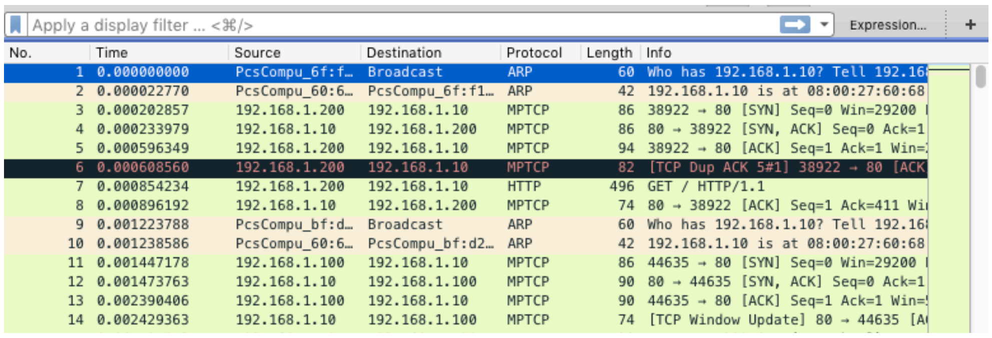
As mentioned in the research methodology, the initial stage of this research is to understand the behavior of MPTCP. For that, the MPTCP kernel was installed on Ubuntu 16.04 LTS and several experiments were done. Rather than using real client-server environment for the testing of the implemented solution, it was decided to use virtual environment which can be easily maintained and configured for different scenarios.

5.1 MPTCP Behavior Testing

As described in Chapter 3, before implementing the proposed solution in Linux Kernel the behavior of the original MPTCP has to be understood properly. Therefore some experiments were conducted on a controlled environment created by virtual machines as mentioned before.

5.1.1 MPTCP with Apache server

Apache server was installed and a simple web page was hosted on one virtual machine. Another virtual machine was configured as a client machine with two network interfaces. Both the machines were connected to the internal network of the Virtual Box [33]. Then the web page was requested by the client's web browser and the packets were captured using Wireshark. As shown in Figure 5.1, the connection was established using MPTCP. The packets captured were analyzed by the Wireshark [26]. Figure 5.2 shows the SYN packet from the client with the keys send by the client. SYN/ACK packet is shown in Figure 5.3 with the keys send by the server.



Apply a display filter ... <⌘/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	PcsCompu_6f:f…	Broadcast	ARP	60	Who has 192.168.1.10? Tell 192.16
2	0.000022770	PcsCompu_60:6…	PcsCompu_6f:f1…	ARP	42	192.168.1.10 is at 08:00:27:60:68
3	0.000202857	192.168.1.200	192.168.1.10	MPTCP	86	38922 → 80 [SYN] Seq=0 Win=29200
4	0.000233979	192.168.1.10	192.168.1.200	MPTCP	86	80 → 38922 [SYN, ACK] Seq=0 Ack=1
5	0.000596349	192.168.1.200	192.168.1.10	MPTCP	94	38922 → 80 [ACK] Seq=1 Ack=1 Win=
6	0.000608560	192.168.1.200	192.168.1.10	MPTCP	82	[TCP Dup ACK 5#1] 38922 → 80 [ACK
7	0.000854234	192.168.1.200	192.168.1.10	HTTP	496	GET / HTTP/1.1
8	0.000896192	192.168.1.10	192.168.1.200	MPTCP	74	80 → 38922 [ACK] Seq=1 Ack=411 Wi
9	0.001223788	PcsCompu_bf:d…	Broadcast	ARP	60	Who has 192.168.1.10? Tell 192.16
10	0.001238586	PcsCompu_60:6…	PcsCompu_bf:d2…	ARP	42	192.168.1.10 is at 08:00:27:60:68
11	0.001447178	192.168.1.100	192.168.1.10	MPTCP	86	44635 → 80 [SYN] Seq=0 Win=29200
12	0.001473763	192.168.1.10	192.168.1.100	MPTCP	90	80 → 44635 [SYN, ACK] Seq=0 Ack=1
13	0.002390406	192.168.1.100	192.168.1.10	MPTCP	90	44635 → 80 [ACK] Seq=1 Ack=1 Win=
14	0.002429363	192.168.1.10	192.168.1.100	MPTCP	74	[TCP Window Update] 80 → 44635 [A

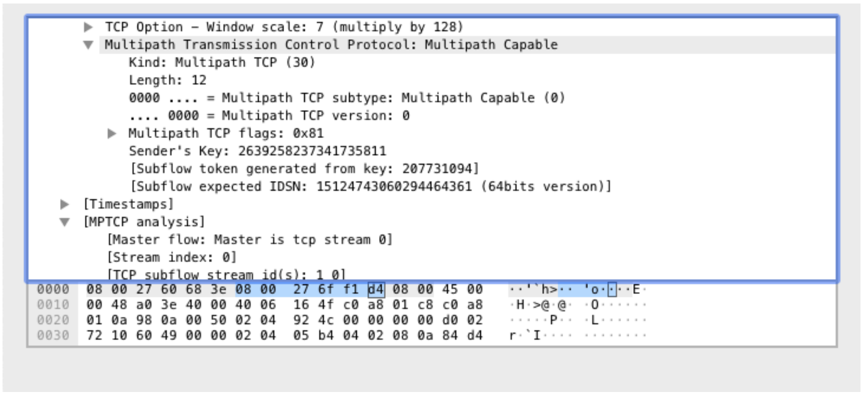
Figure 5.1 : MPTCP handshake

5.1.2 MPTCP with TLS

As mentioned in the previous sections, the proposed solution has to use an external key to authenticate the subflows. TLS was taken as an example for the key negotiation method. Therefore the compatibility of TLS with MPTCP was also tested. After installing TLS on the server, the same client-server test was conducted and the MPTCP was worked without any error. Figure 5.4 shows the initial handshake of MPTCP and TLS.

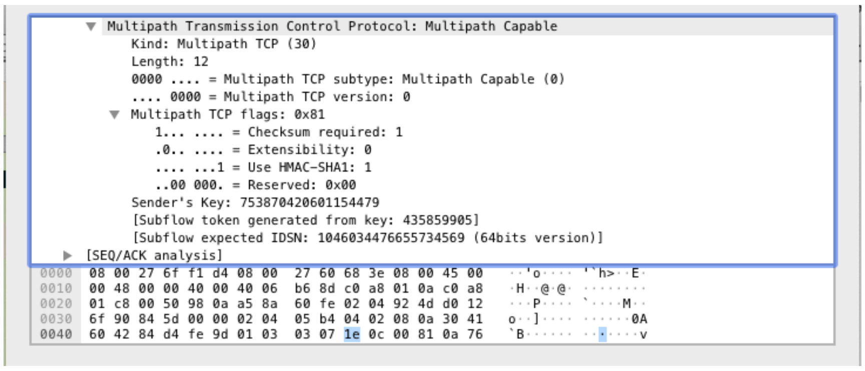
5.1.3 MPTCP with iOS

The major commercial implementation of MPTCP was in the Apple iOS. Network packets transferred in Apple iPhone was captured using Xcode app and command line tools provided by Apple. These packets were analyzed using Wireshark and the iOS has used MPTCP protocol as shown in Figure 5.5. More details about the usage of MPTCP on iOS is discussed in Appendix B.



```
    ▶ TCP Option - Window scale: 7 (multiply by 128)
    ▼ Multipath Transmission Control Protocol: Multipath Capable
        Kind: Multipath TCP (30)
        Length: 12
        0000 .... = Multipath TCP subtype: Multipath Capable (0)
        .... 0000 = Multipath TCP version: 0
      ▶ Multipath TCP flags: 0x81
        Sender's Key: 2639258237341735811
        [Subflow token generated from key: 207731094]
        [Subflow expected IDSN: 15124743060294464361 (64bits version)]
  ▶ [Timestamps]
  ▼ [MPTCP analysis]
      [Master flow: Master is tcp stream 0]
      [Stream index: 0]
      [TCP subflow stream id(s): 1 0]
0000  08 00 27 60 68 3e 08 00  27 6f f1 d4 08 00 45 00   ··'`h>·· 'o····E·
0010  00 48 a0 3e 40 00 40 06  16 4f c0 a8 01 c8 c0 a8   ·H·>@·@· ·O······
0020  01 0a 98 0a 00 50 02 04  92 4c 00 00 00 00 d0 02   ·····P·· ·L······
0030  72 10 60 49 00 00 02 04  05 b4 04 02 08 0a 84 d4   r·`I···· ········
```

Figure 5.2 : SYN packet of MPTCP

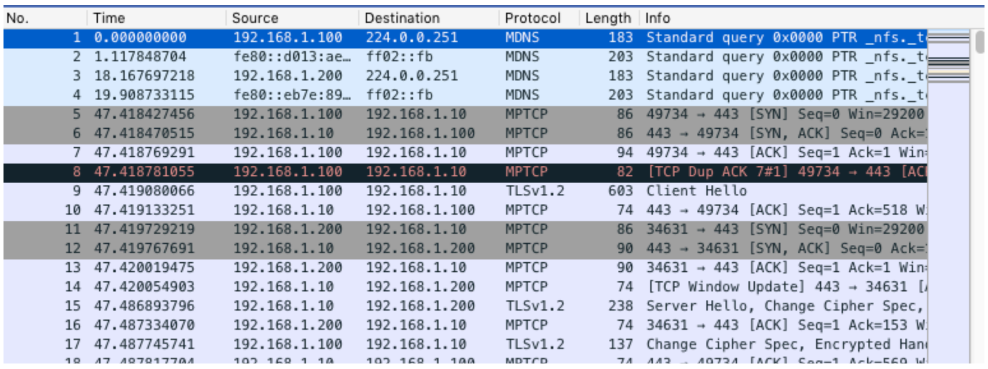


```
    ▼ Multipath Transmission Control Protocol: Multipath Capable
        Kind: Multipath TCP (30)
        Length: 12
        0000 .... = Multipath TCP subtype: Multipath Capable (0)
        .... 0000 = Multipath TCP version: 0
      ▼ Multipath TCP flags: 0x81
          1... .... = Checksum required: 1
          .0.. .... = Extensibility: 0
          .... ...1 = Use HMAC-SHA1: 1
          ..00 000. = Reserved: 0x00
        Sender's Key: 753870420601154479
        [Subflow token generated from key: 435859905]
        [Subflow expected IDSN: 1046034476655734569 (64bits version)]
  ▶ [SEQ/ACK analysis]
0000  08 00 27 6f f1 d4 08 00  27 60 68 3e 08 00 45 00   ··'o···· '`h>··E·
0010  00 48 00 00 40 00 40 06  b6 8d c0 a8 01 0a c0 a8   ·H··@·@· ········
0020  01 c8 00 50 98 0a a5 8a  60 fe 02 04 92 4d d0 12   ···P···· `····M··
0030  6f 90 84 5d 00 00 02 04  05 b4 04 02 08 0a 30 41   o··]···· ······0A
0040  60 42 84 d4 fe 9d 01 03  03 07 1e 0c 00 81 0a 76   `B······ ·······v
```

Figure 5.3 : SYN/ACK packet of MPTCP

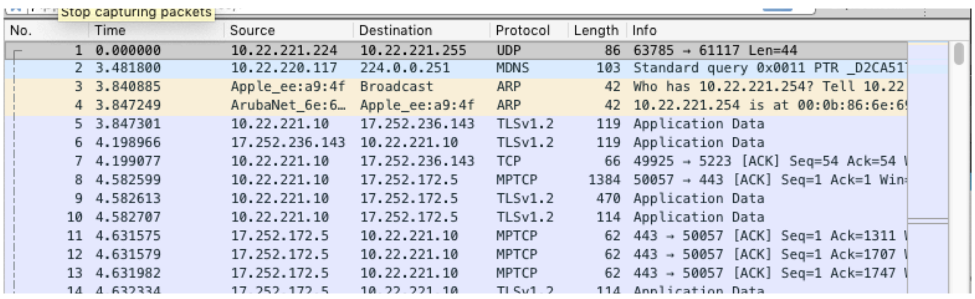
5.1.4 MPTCP with TCP sockets

If the devices are configured with MPTCP, when opening a TCP socket it should initiate a MPTCP connection. Since there are no separate MPTCP sockets, most of the experiments in this research were conducted using TCP sockets. Therefore it is better to test the behavior of MPTCP with TCP sockets. TCP client and server sockets were implemented using the C programming language and executed on two virtual machines connected via the virtual network of Virtual box. Network packets were captured in the server machine and were analyzed using Wireshark. The results show in Figure 5.6 proves that it has used MPTCP protocol to communicate and it has created two subflows using the two interfaces of the client machine.



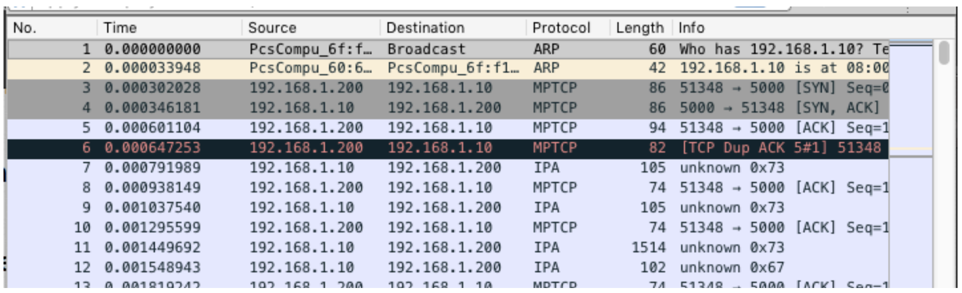
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.1.100	224.0.0.251	MDNS	183	Standard query 0x0000 PTR _nfs._t
2	1.117848704	fe80::d013:ae…	ff02::fb	MDNS	203	Standard query 0x0000 PTR _nfs._t
3	18.167697218	192.168.1.200	224.0.0.251	MDNS	183	Standard query 0x0000 PTR _nfs._t
4	19.908733115	fe80::eb7e:89…	ff02::fb	MDNS	203	Standard query 0x0000 PTR _nfs._t
5	47.418427456	192.168.1.100	192.168.1.10	MPTCP	86	49734 → 443 [SYN] Seq=0 Win=29200
6	47.418470515	192.168.1.10	192.168.1.100	MPTCP	86	443 → 49734 [SYN, ACK] Seq=0 Ack=
7	47.418769291	192.168.1.100	192.168.1.10	MPTCP	94	49734 → 443 [ACK] Seq=1 Ack=1 Win
8	47.418781055	192.168.1.100	192.168.1.10	MPTCP	82	[TCP Dup ACK 7#1] 49734 → 443 [AC
9	47.419080066	192.168.1.100	192.168.1.10	TLSv1.2	603	Client Hello
10	47.419133251	192.168.1.10	192.168.1.100	MPTCP	74	443 → 49734 [ACK] Seq=1 Ack=518 W
11	47.419729219	192.168.1.200	192.168.1.10	MPTCP	86	34631 → 443 [SYN] Seq=0 Win=29200
12	47.419767691	192.168.1.10	192.168.1.200	MPTCP	90	443 → 34631 [SYN, ACK] Seq=0 Ack=
13	47.420019475	192.168.1.200	192.168.1.10	MPTCP	90	34631 → 443 [ACK] Seq=1 Ack=1 Win
14	47.420054903	192.168.1.10	192.168.1.200	MPTCP	74	[TCP Window Update] 443 → 34631 [
15	47.486893796	192.168.1.10	192.168.1.200	TLSv1.2	238	Server Hello, Change Cipher Spec,
16	47.487334070	192.168.1.200	192.168.1.10	MPTCP	74	34631 → 443 [ACK] Seq=1 Ack=153 W
17	47.487745741	192.168.1.100	192.168.1.10	TLSv1.2	137	Change Cipher Spec, Encrypted Han
18	47.487817704	192.168.1.10	192.168.1.100	MPTCP	74	443 → 49734 [ACK] Seq=1 Ack=569 W

Figure 5.4 : TLS handshake



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.22.221.224	10.22.221.255	UDP	86	63785 → 61117 Len=44
2	3.481800	10.22.220.117	224.0.0.251	MDNS	103	Standard query 0x0011 PTR _D2CA51
3	3.840885	Apple_ee:a9:4f	Broadcast	ARP	42	Who has 10.22.221.254? Tell 10.22
4	3.847249	ArubaNet_6e:6…	Apple_ee:a9:4f	ARP	42	10.22.221.254 is at 00:0b:86:6e:6
5	3.847301	10.22.221.10	17.252.236.143	TLSv1.2	119	Application Data
6	4.198966	17.252.236.143	10.22.221.10	TLSv1.2	119	Application Data
7	4.199077	10.22.221.10	17.252.236.143	TCP	66	49925 → 5223 [ACK] Seq=54 Ack=54
8	4.582599	10.22.221.10	17.252.172.5	MPTCP	1384	50057 → 443 [ACK] Seq=1 Ack=1 Win
9	4.582613	10.22.221.10	17.252.172.5	TLSv1.2	470	Application Data
10	4.582707	10.22.221.10	17.252.172.5	TLSv1.2	114	Application Data
11	4.631575	17.252.172.5	10.22.221.10	MPTCP	62	443 → 50057 [ACK] Seq=1 Ack=1311
12	4.631579	17.252.172.5	10.22.221.10	MPTCP	62	443 → 50057 [ACK] Seq=1 Ack=1707
13	4.631982	17.252.172.5	10.22.221.10	MPTCP	62	443 → 50057 [ACK] Seq=1 Ack=1747
14	4.632334	17.252.172.5	10.22.221.10	TLSv1.2	114	Application Data

Figure 5.5 : MPTCP with iOS



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	PcsCompu_6f:f…	Broadcast	ARP	60	Who has 192.168.1.10? Te
2	0.000033948	PcsCompu_60:6…	PcsCompu_6f:f1…	ARP	42	192.168.1.10 is at 08:00
3	0.000302028	192.168.1.200	192.168.1.10	MPTCP	86	51348 → 5000 [SYN] Seq=0
4	0.000346181	192.168.1.10	192.168.1.200	MPTCP	86	5000 → 51348 [SYN, ACK]
5	0.000601104	192.168.1.200	192.168.1.10	MPTCP	94	51348 → 5000 [ACK] Seq=1
6	0.000647253	192.168.1.200	192.168.1.10	MPTCP	82	[TCP Dup ACK 5#1] 51348
7	0.000791989	192.168.1.10	192.168.1.200	IPA	105	unknown 0x73
8	0.000938149	192.168.1.200	192.168.1.10	MPTCP	74	51348 → 5000 [ACK] Seq=1
9	0.001037540	192.168.1.10	192.168.1.200	IPA	105	unknown 0x73
10	0.001295599	192.168.1.200	192.168.1.10	MPTCP	74	51348 → 5000 [ACK] Seq=1
11	0.001449692	192.168.1.10	192.168.1.200	IPA	1514	unknown 0x73
12	0.001548943	192.168.1.10	192.168.1.200	IPA	102	unknown 0x67
13	0.001819242	192.168.1.200	192.168.1.10	MPTCP	74	51348 → 5000 [ACK] Seq=1

Figure 5.6 : MPTCP in TCP socket

5.2 Testing the developed solutions

As discussed in Chapter 3, after having a better idea about the MPTCP and the behavior of subflow, the next step is to develop the proposed solutions. All the details related to the implementation of the proposed solutions are mentioned in Chapter 4.

Testing the developed solution can be broken down into three sections as shown in Figure 5.7. The user level key is assigned to the *sin_zero* variable as mentioned in the previous sections. In the first test case, it has to check whether the connection was established when both the ends use the same external key. In the second test case, it has to use different keys on both ends and check whether the connection has established or not. Finally, it has to check whether the solution is backward compatible with not assigning any value to *sin_zero*. In that case, it should use the original MPTCP authentication mechanism and establish the connections.

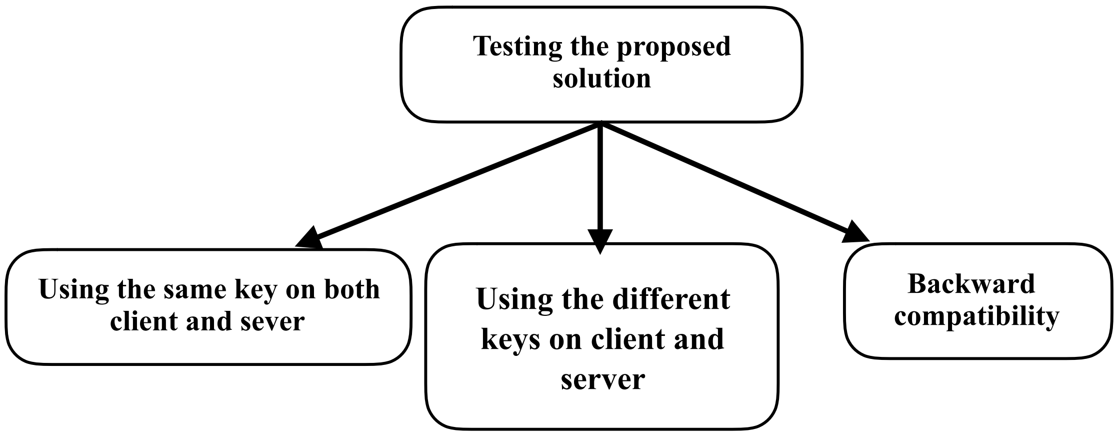
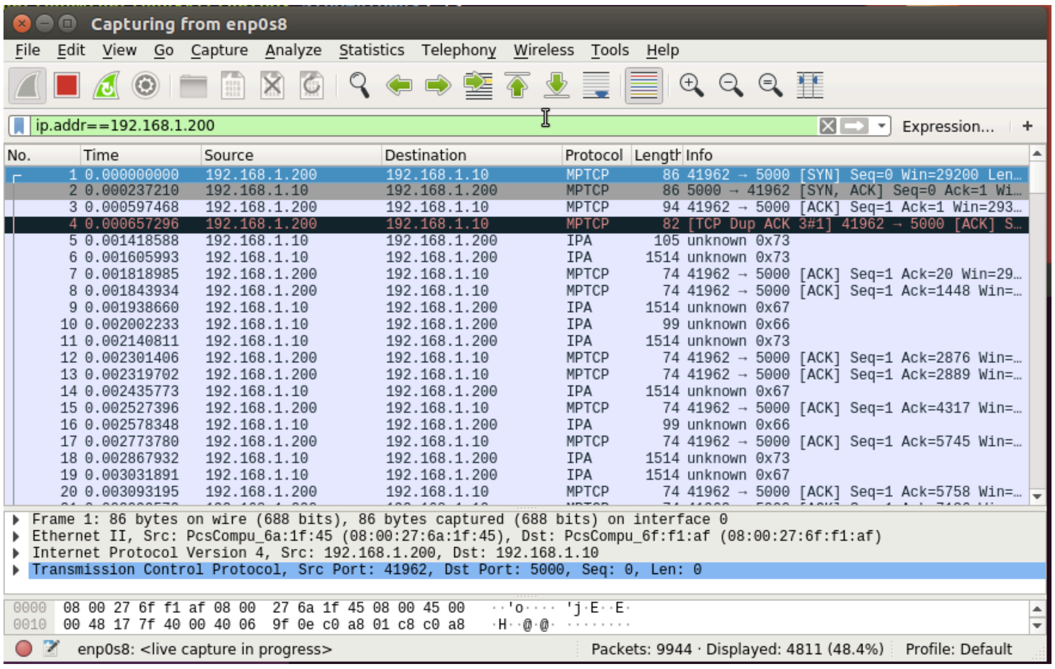

Testing the proposed solution
Using the same key on both client and sever
Using the different keys on client and server
Backward compatibility

Figure 5.7 : Testing the Proposed solution

Simple TCP socket program was used to test the implementation of the proposed solution. These programs were written in C programing language. The socket application of the server has sent a string of data to the socket application at the client. The client application has displayed the information sent by the server on the terminal. While this simple socket programming executing on the client-server environment, the network packets were captured on both ends by using Wireshark application. These network packets were analyzed to test the proposed solution by the research.

5.2.1 Using the same key on both client and server

According to the proposed solution by this research, when using the same key on both the server and the client, the MPTCP connection should start properly. It has to authenticate the second subflow using the authentication material generated by the key obtained from the user space. Therefore it should start the second subflow by using the available network interface of the client machine. The connection was established using TCP sockets from both the server and client machines and the packets were captured using the Wireshark applications. Figure 5.8 shows the packets send from interface *eth0* of the client and Figure 5.9 shows the packets send from interface *eth1* of the client. By analyzing the packets, it can come to a conclusion that both the interfaces has successfully completed the three-way handshake and established the MPTCP connection on both the interfaces successfully.


Capturing from enp0s8
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Expression…
+

```
ip.addr==192.168.1.200
```

```
No.  Time         Source          Destination     Protocol Length Info
   1 0.000000000  192.168.1.200   192.168.1.10    MPTCP      86 41962 → 5000 [SYN] Seq=0 Win=29200 Len…
   2 0.000237210  192.168.1.10    192.168.1.200   MPTCP      86 5000 → 41962 [SYN, ACK] Seq=0 Ack=1 Wi…
   3 0.000597468  192.168.1.200   192.168.1.10    MPTCP      94 41962 → 5000 [ACK] Seq=1 Ack=1 Win=293…
   4 0.000657296  192.168.1.200   192.168.1.10    MPTCP      82 [TCP Dup ACK 3#1] 41962 → 5000 [ACK] S…
   5 0.001418588  192.168.1.10    192.168.1.200   IPA       105 unknown 0x73
   6 0.001605993  192.168.1.10    192.168.1.200   IPA      1514 unknown 0x73
   7 0.001818985  192.168.1.200   192.168.1.10    MPTCP      74 41962 → 5000 [ACK] Seq=1 Ack=20 Win=29…
   8 0.001843934  192.168.1.200   192.168.1.10    MPTCP      74 41962 → 5000 [ACK] Seq=1 Ack=1448 Win=…
   9 0.001938660  192.168.1.10    192.168.1.200   IPA      1514 unknown 0x67
  10 0.002002233  192.168.1.10    192.168.1.200   IPA        99 unknown 0x66
  11 0.002140811  192.168.1.10    192.168.1.200   IPA      1514 unknown 0x73
  12 0.002301406  192.168.1.200   192.168.1.10    MPTCP      74 41962 → 5000 [ACK] Seq=1 Ack=2876 Win=…
  13 0.002319702  192.168.1.200   192.168.1.10    MPTCP      74 41962 → 5000 [ACK] Seq=1 Ack=2889 Win=…
  14 0.002435773  192.168.1.10    192.168.1.200   IPA      1514 unknown 0x67
  15 0.002527396  192.168.1.200   192.168.1.10    MPTCP      74 41962 → 5000 [ACK] Seq=1 Ack=4317 Win=…
  16 0.002578348  192.168.1.10    192.168.1.200   IPA        99 unknown 0x66
  17 0.002773780  192.168.1.200   192.168.1.10    MPTCP      74 41962 → 5000 [ACK] Seq=1 Ack=5745 Win=…
  18 0.002867932  192.168.1.10    192.168.1.200   IPA      1514 unknown 0x73
  19 0.003031891  192.168.1.10    192.168.1.200   IPA      1514 unknown 0x67
  20 0.003093195  192.168.1.200   192.168.1.10    MPTCP      74 41962 → 5000 [ACK] Seq=1 Ack=5758 Win=…
```

```
▸ Frame 1: 86 bytes on wire (688 bits), 86 bytes captured (688 bits) on interface 0
▸ Ethernet II, Src: PcsCompu_6a:1f:45 (08:00:27:6a:1f:45), Dst: PcsCompu_6f:f1:af (08:00:27:6f:f1:af)
▸ Internet Protocol Version 4, Src: 192.168.1.200, Dst: 192.168.1.10
▸ Transmission Control Protocol, Src Port: 41962, Dst Port: 5000, Seq: 0, Len: 0
```

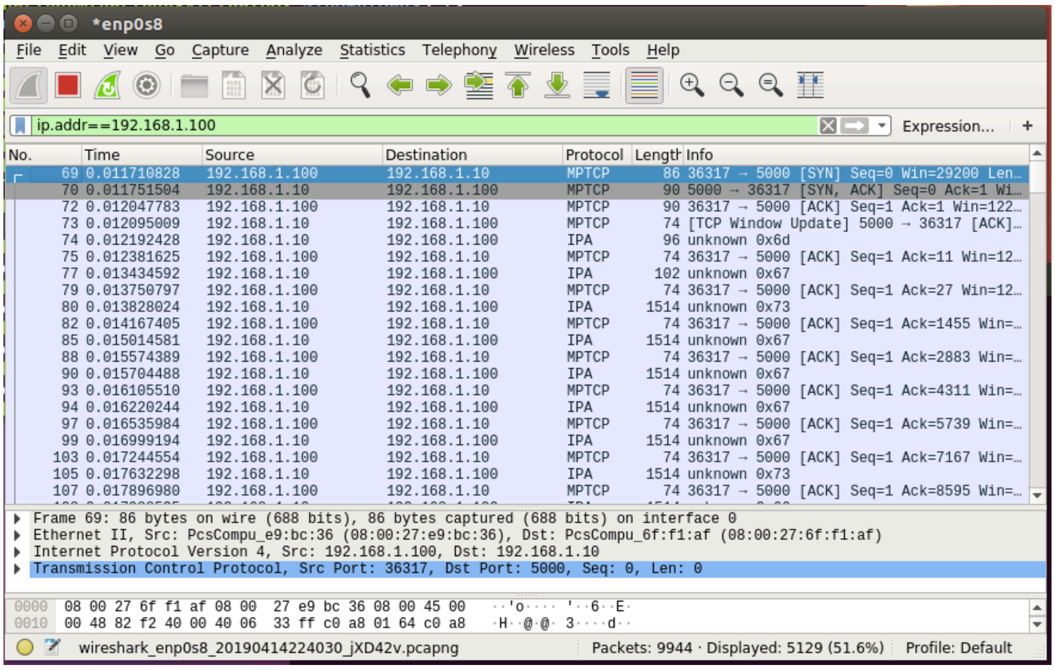
```
0000  08 00 27 6f f1 af 08 00  27 6a 1f 45 08 00 45 00   ··'o···· 'j·E··E·
0010  00 48 17 7f 40 00 40 06  9f 0e c0 a8 01 c8 c0 a8   ·H··@·@· ········
```

```
enp0s8: <live capture in progress>     Packets: 9944 · Displayed: 4811 (48.4%)     Profile: Default
```

Figure 5.8 : Packets captured from eth0 interface with same key

5.2.2 Using the different keys on client and server

According to the proposed solution, the external key is only known by the client and the server. If the client needs to create another subflow, it should provide a valid authentication material. According to the assumptions made in this research, the external keys are secure and known only by the client and the server. If an attacker tries to create a subflow with an exciting connection, the attacker has to provide an authentication material which is generated using a different external key and it is not the valid one. Theoretically, when using two different keys on the server and the client, the server should not start the second subflow. When the key used by the client is different from the key used by the server, the server authentication process fails. Therefore the protocol should refrain from authenticating the second subflow. According to the proposed solution of this research, this is the expected behavior for this kind of a scenario. If the client fails to provide the correct authentication material, the server should refuse the connection and assume that the connection is initiated by a malicious party.


*enp0s8
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Expression...
+

```
ip.addr==192.168.1.100

No.  Time         Source         Destination    Protocol Length Info
  69 0.011710828  192.168.1.100  192.168.1.10   MPTCP      86 36317 → 5000 [SYN] Seq=0 Win=29200 Len…
  70 0.011751504  192.168.1.10   192.168.1.100  MPTCP      90 5000 → 36317 [SYN, ACK] Seq=0 Ack=1 Wi…
  72 0.012047783  192.168.1.100  192.168.1.10   MPTCP      90 36317 → 5000 [ACK] Seq=1 Ack=1 Win=122…
  73 0.012095009  192.168.1.10   192.168.1.100  MPTCP      74 [TCP Window Update] 5000 → 36317 [ACK]…
  74 0.012192428  192.168.1.10   192.168.1.100  IPA        96 unknown 0x6d
  75 0.012381625  192.168.1.100  192.168.1.10   MPTCP      74 36317 → 5000 [ACK] Seq=1 Ack=11 Win=12…
  77 0.013434592  192.168.1.10   192.168.1.100  IPA       102 unknown 0x67
  79 0.013750797  192.168.1.100  192.168.1.10   MPTCP      74 36317 → 5000 [ACK] Seq=1 Ack=27 Win=12…
  80 0.013828024  192.168.1.10   192.168.1.100  IPA      1514 unknown 0x73
  82 0.014167405  192.168.1.100  192.168.1.10   MPTCP      74 36317 → 5000 [ACK] Seq=1 Ack=1455 Win=…
  85 0.015014581  192.168.1.10   192.168.1.100  IPA      1514 unknown 0x67
  88 0.015574389  192.168.1.100  192.168.1.10   MPTCP      74 36317 → 5000 [ACK] Seq=1 Ack=2883 Win=…
  90 0.015704488  192.168.1.10   192.168.1.100  IPA      1514 unknown 0x67
  93 0.016105510  192.168.1.100  192.168.1.10   MPTCP      74 36317 → 5000 [ACK] Seq=1 Ack=4311 Win=…
  94 0.016220244  192.168.1.10   192.168.1.100  IPA      1514 unknown 0x67
  97 0.016535984  192.168.1.100  192.168.1.10   MPTCP      74 36317 → 5000 [ACK] Seq=1 Ack=5739 Win=…
  99 0.016999194  192.168.1.10   192.168.1.100  IPA      1514 unknown 0x67
 103 0.017244554  192.168.1.100  192.168.1.10   MPTCP      74 36317 → 5000 [ACK] Seq=1 Ack=7167 Win=…
 105 0.017632298  192.168.1.10   192.168.1.100  IPA      1514 unknown 0x73
 107 0.017896980  192.168.1.100  192.168.1.10   MPTCP      74 36317 → 5000 [ACK] Seq=1 Ack=8595 Win=…

▸ Frame 69: 86 bytes on wire (688 bits), 86 bytes captured (688 bits) on interface 0
▸ Ethernet II, Src: PcsCompu_e9:bc:36 (08:00:27:e9:bc:36), Dst: PcsCompu_6f:f1:af (08:00:27:6f:f1:af)
▸ Internet Protocol Version 4, Src: 192.168.1.100, Dst: 192.168.1.10
▸ Transmission Control Protocol, Src Port: 36317, Dst Port: 5000, Seq: 0, Len: 0

0000  08 00 27 6f f1 af 08 00  27 e9 bc 36 08 00 45 00   ··'o····'··6··E·
0010  00 48 82 f2 40 00 40 06  33 ff c0 a8 01 64 c0 a8   ·H··@·@·3····d··

wireshark_enp0s8_20190414224030_jXD42v.pcapng   Packets: 9944 · Displayed: 5129 (51.6%)   Profile: Default
```

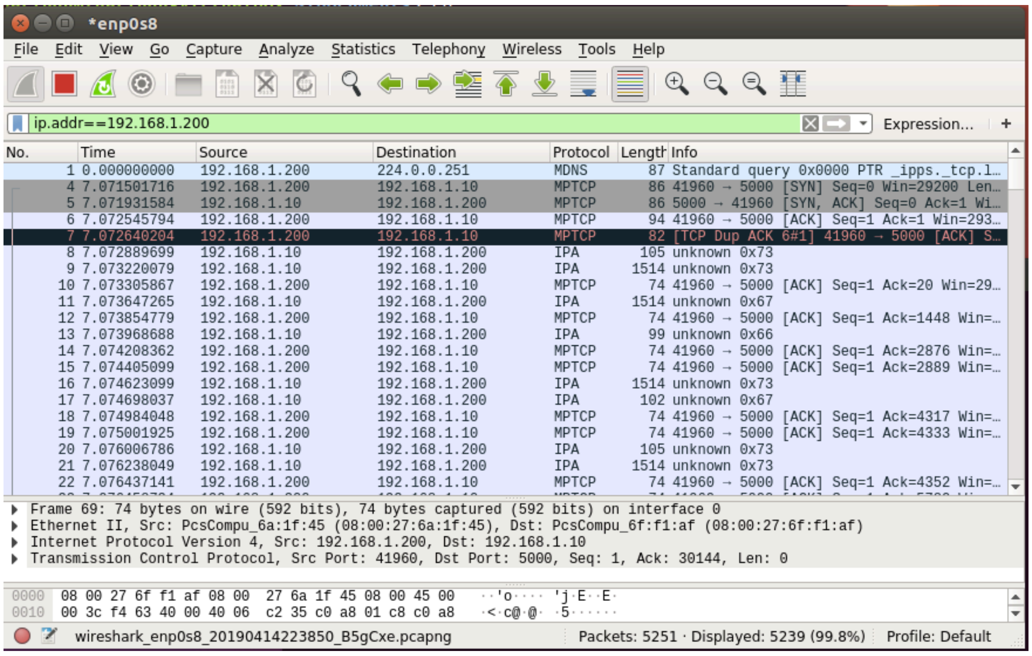
Figure 5.9 : Packets captured from eth1 interface with same key

To test this scenario, two different keys were used in server and client when starting the sockets, and the data sent was captured using the Wireshark application. Figure 5.10 shows the captured and filtered packets send from the client's *eth0* interface and Figure 5.11 shows the packets sent from the *eth1* interface of the client.

According to the captured packets, it is clear that the first connection was successfully established with the server because the three ways handshake was successfully completed. But when observing Figure 5.11, it is clear that the three ways handshake has stopped at the

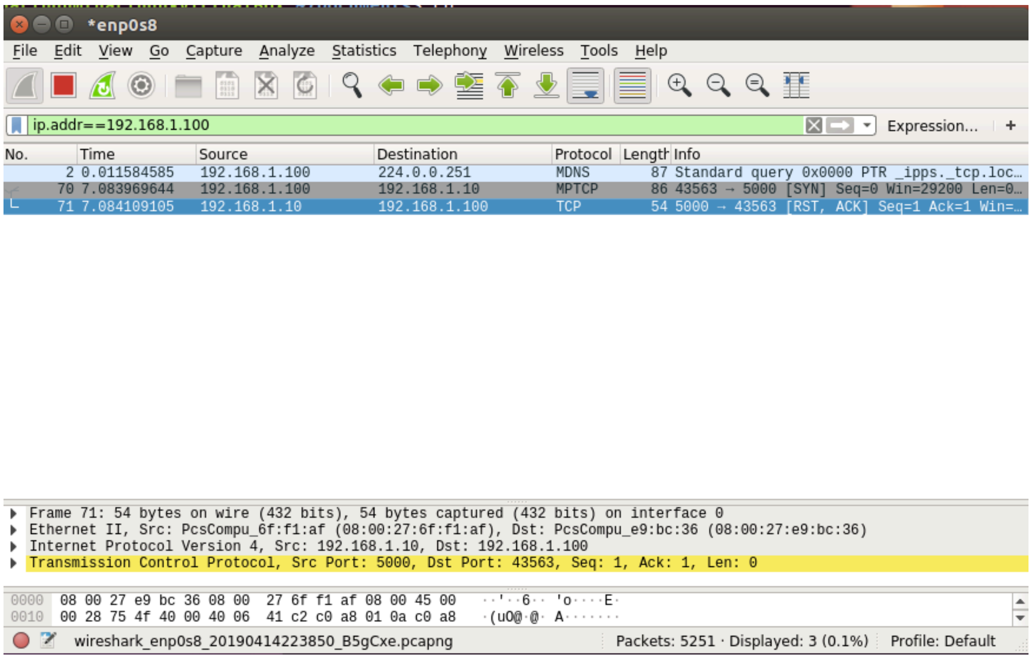
SYN_ACK stage. Because the protocol fails to complete the authentication and due to that reason it has dropped the connection.

Therefore the main intention of the research was successfully achieved. Which means without having the correct user level information, the second subflow cannot be initiated.


*enp0s8
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Expression...

```
ip.addr==192.168.1.200

No.   Time          Source          Destination     Protocol Length Info
    1 0.000000000   192.168.1.200   224.0.0.251     MDNS       87 Standard query 0x0000 PTR _ipps._tcp.l…
    4 7.071501716   192.168.1.200   192.168.1.10    MPTCP      86 41960 → 5000 [SYN] Seq=0 Win=29200 Len…
    5 7.071931584   192.168.1.10    192.168.1.200   MPTCP      86 5000 → 41960 [SYN, ACK] Seq=0 Ack=1 Wi…
    6 7.072545794   192.168.1.200   192.168.1.10    MPTCP      94 41960 → 5000 [ACK] Seq=1 Ack=1 Win=293…
    7 7.072640204   192.168.1.200   192.168.1.10    MPTCP      82 [TCP Dup ACK 6#1] 41960 → 5000 [ACK] S…
    8 7.072889699   192.168.1.10    192.168.1.200   IPA       105 unknown 0x73
    9 7.073220079   192.168.1.10    192.168.1.200   IPA      1514 unknown 0x73
   10 7.073305867   192.168.1.200   192.168.1.10    MPTCP      74 41960 → 5000 [ACK] Seq=1 Ack=20 Win=29…
   11 7.073647265   192.168.1.10    192.168.1.200   IPA      1514 unknown 0x67
   12 7.073854779   192.168.1.200   192.168.1.10    MPTCP      74 41960 → 5000 [ACK] Seq=1 Ack=1448 Win=…
   13 7.073968688   192.168.1.10    192.168.1.200   IPA        99 unknown 0x66
   14 7.074208362   192.168.1.200   192.168.1.10    MPTCP      74 41960 → 5000 [ACK] Seq=1 Ack=2876 Win=…
   15 7.074405099   192.168.1.200   192.168.1.10    MPTCP      74 41960 → 5000 [ACK] Seq=1 Ack=2889 Win=…
   16 7.074623099   192.168.1.10    192.168.1.200   IPA      1514 unknown 0x73
   17 7.074698037   192.168.1.10    192.168.1.200   IPA       102 unknown 0x67
   18 7.074984048   192.168.1.200   192.168.1.10    MPTCP      74 41960 → 5000 [ACK] Seq=1 Ack=4317 Win=…
   19 7.075001925   192.168.1.200   192.168.1.10    MPTCP      74 41960 → 5000 [ACK] Seq=1 Ack=4333 Win=…
   20 7.076006786   192.168.1.10    192.168.1.200   IPA       105 unknown 0x73
   21 7.076238049   192.168.1.10    192.168.1.200   IPA      1514 unknown 0x73
   22 7.076437141   192.168.1.200   192.168.1.10    MPTCP      74 41960 → 5000 [ACK] Seq=1 Ack=4352 Win=…

▸ Frame 69: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
▸ Ethernet II, Src: PcsCompu_6a:1f:45 (08:00:27:6a:1f:45), Dst: PcsCompu_6f:f1:af (08:00:27:6f:f1:af)
▸ Internet Protocol Version 4, Src: 192.168.1.200, Dst: 192.168.1.10
▸ Transmission Control Protocol, Src Port: 41960, Dst Port: 5000, Seq: 1, Ack: 30144, Len: 0

0000  08 00 27 6f f1 af 08 00  27 6a 1f 45 08 00 45 00   ··'o···· 'j·E··E·
0010  00 3c f4 63 40 00 40 06  c2 35 c0 a8 01 c8 c0 a8   ·<·c@·@· ·5······

wireshark_enp0s8_20190414223850_B5gCxe.pcapng    Packets: 5251 · Displayed: 5239 (99.8%)    Profile: Default
```

Figure 5.10 : Packets captured from eth0 interface with different key


*enp0s8
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Expression...

```
ip.addr==192.168.1.100

No.   Time          Source          Destination     Protocol Length Info
    2 0.011584585   192.168.1.100   224.0.0.251     MDNS       87 Standard query 0x0000 PTR _ipps._tcp.loc…
   70 7.083969644   192.168.1.100   192.168.1.10    MPTCP      86 43563 → 5000 [SYN] Seq=0 Win=29200 Len=0…
   71 7.084109105   192.168.1.10    192.168.1.100   TCP        54 5000 → 43563 [RST, ACK] Seq=1 Ack=1 Win=…

▸ Frame 71: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
▸ Ethernet II, Src: PcsCompu_6f:f1:af (08:00:27:6f:f1:af), Dst: PcsCompu_e9:bc:36 (08:00:27:e9:bc:36)
▸ Internet Protocol Version 4, Src: 192.168.1.10, Dst: 192.168.1.100
▸ Transmission Control Protocol, Src Port: 5000, Dst Port: 43563, Seq: 1, Ack: 1, Len: 0

0000  08 00 27 e9 bc 36 08 00  27 6f f1 af 08 00 45 00   ··'··6·· 'o····E·
0010  00 28 75 4f 40 00 40 06  41 c2 c0 a8 01 0a c0 a8   ·(uO@·@· A·······

wireshark_enp0s8_20190414223850_B5gCxe.pcapng    Packets: 5251 · Displayed: 3 (0.1%)    Profile: Default
```

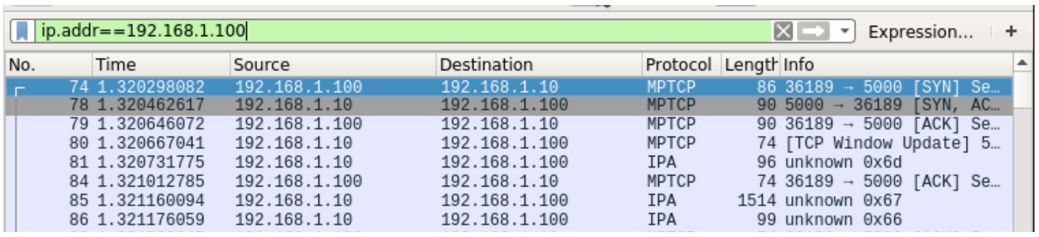
Figure 5.11 : Packets captured from eth1 interface with different key

5.2.3 Backward compatibility

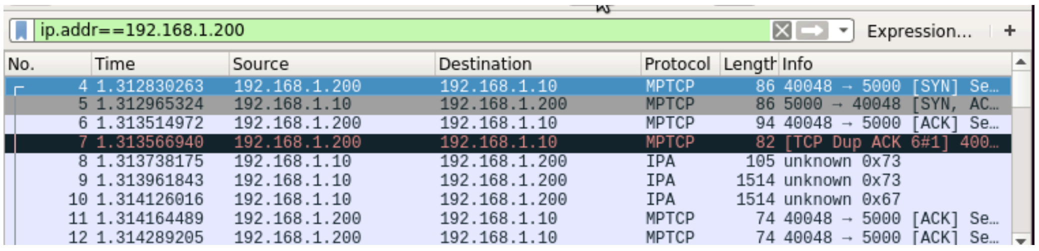
If any of the nodes are not configured with the proposed solution to use user space information to authenticate the MPTCP subflow, it should automatically use the original MPTCP. To test whether it worked, the sockets have to be set without assigning any value to *sin_zero* variable.

In order to check the backward compatibility of the proposed solution, the connection was established by using the client and the server applications without providing any external key. Figure 5.12 shows the packets captured on both the *eth0* and *eth1* interfaces of the client and it has successfully established the MPTCP connection. Which shows that it has used normal MPTCP instead of the proposed solution. Therefore the solution is backward compatible if both the client and server does not use the proposed solution.

But according to the current implementation, if one of the server or client has used the proposed solution and other does not, the backward compatibility won't work properly. To verify that scenario, an external key was used at the client and no key was used at the server. Figure 5.13 shows the packets captured while creating a connection between the hosts. So it is clear that the MPTCP connection was not successfully created. Theoretically, it should create a MPTCP connection properly. But the second subflow has not joined to the connection. The solution needs to be improved to handle this kind of situation and this is further discusses in the future work section.

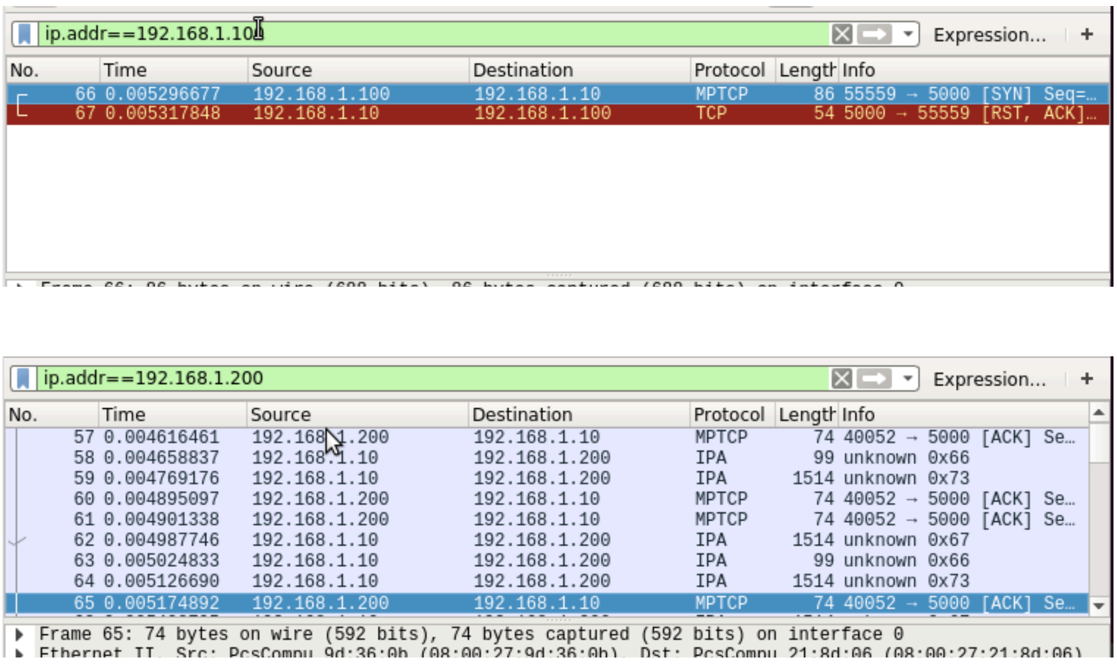


```
ip.addr==192.168.1.100                                              Expression...  +
No.   Time          Source          Destination     Protocol  Length Info
   74 1.320298082   192.168.1.100   192.168.1.10    MPTCP       86 36189 → 5000 [SYN] Se…
   78 1.320462617   192.168.1.10    192.168.1.100   MPTCP       90 5000 → 36189 [SYN, AC…
   79 1.320646072   192.168.1.100   192.168.1.10    MPTCP       90 36189 → 5000 [ACK] Se…
   80 1.320667041   192.168.1.10    192.168.1.100   MPTCP       74 [TCP Window Update] 5…
   81 1.320731775   192.168.1.10    192.168.1.100   IPA         96 unknown 0x6d
   84 1.321012785   192.168.1.100   192.168.1.10    MPTCP       74 36189 → 5000 [ACK] Se…
   85 1.321160094   192.168.1.10    192.168.1.100   IPA       1514 unknown 0x67
   86 1.321176059   192.168.1.10    192.168.1.100   IPA         99 unknown 0x66
```



```
ip.addr==192.168.1.200                                              Expression...  +
No.   Time          Source          Destination     Protocol  Length Info
    4 1.312830263   192.168.1.200   192.168.1.10    MPTCP       86 40048 → 5000 [SYN] Se…
    5 1.312965324   192.168.1.10    192.168.1.200   MPTCP       86 5000 → 40048 [SYN, AC…
    6 1.313514972   192.168.1.200   192.168.1.10    MPTCP       94 40048 → 5000 [ACK] Se…
    7 1.313566940   192.168.1.200   192.168.1.10    MPTCP       82 [TCP Dup ACK 6#1] 400…
    8 1.313738175   192.168.1.10    192.168.1.200   IPA        105 unknown 0x73
    9 1.313961843   192.168.1.10    192.168.1.200   IPA       1514 unknown 0x73
   10 1.314126016   192.168.1.10    192.168.1.200   IPA       1514 unknown 0x67
   11 1.314164489   192.168.1.200   192.168.1.10    MPTCP       74 40048 → 5000 [ACK] Se…
   12 1.314289205   192.168.1.200   192.168.1.10    MPTCP       74 40048 → 5000 [ACK] Se…
```

Figure 5.12 : Packets captured from eth0 and eth1 interfaces with no key


```
ip.addr==192.168.1.100
No.   Time          Source          Destination     Protocol  Length Info
   66 0.005296677   192.168.1.100   192.168.1.10    MPTCP       86 55559 → 5000 [SYN] Seq=…
   67 0.005317848   192.168.1.10    192.168.1.100   TCP         54 5000 → 55559 [RST, ACK]…

Frame 66: 86 bytes on wire (688 bits), 86 bytes captured (688 bits) on interface 0
```

```
ip.addr==192.168.1.200
No.   Time          Source          Destination     Protocol  Length Info
   57 0.004616461   192.168.1.200   192.168.1.10    MPTCP       74 40052 → 5000 [ACK] Se…
   58 0.004658837   192.168.1.10    192.168.1.200   IPA         99 unknown 0x66
   59 0.004769176   192.168.1.10    192.168.1.200   IPA       1514 unknown 0x73
   60 0.004895097   192.168.1.200   192.168.1.10    MPTCP       74 40052 → 5000 [ACK] Se…
   61 0.004901338   192.168.1.200   192.168.1.10    MPTCP       74 40052 → 5000 [ACK] Se…
   62 0.004987746   192.168.1.10    192.168.1.200   IPA       1514 unknown 0x67
   63 0.005024833   192.168.1.10    192.168.1.200   IPA         99 unknown 0x66
   64 0.005126690   192.168.1.10    192.168.1.200   IPA       1514 unknown 0x73
   65 0.005174892   192.168.1.200   192.168.1.10    MPTCP       74 40052 → 5000 [ACK] Se…

▸ Frame 65: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
▸ Ethernet II, Src: PcsCompu_9d:36:0b (08:00:27:9d:36:0b), Dst: PcsCompu_21:8d:06 (08:00:27:21:8d:06)
```

Figure 5.13 : Packets captured from eth0 and eth1 interfaces with only the client key

5.3 Robustness of the proposed solution

As mentioned in the problem statement, one of the main reasons for the security threats in MPTCP is the exchange of keys in plain text format at the initial handshake. There is a number of methods proposed to secure the key exchange using different technologies. But the main focus of this research is to secure the creation of new subflows by authenticating the subflows using the key from user space or application level. The proposed solution in this research is to transfer user space key to the kernel space and use the key to create authentication material. Later this authentication material is sent to the server to authenticate the subflows. As described in the previous sections, the implantation was perfectly working as designed in Chapter 3. Though it has proved that the solution is working on implementation vise, it has to be robust and should provide the solution for the threat that was discussed this research.

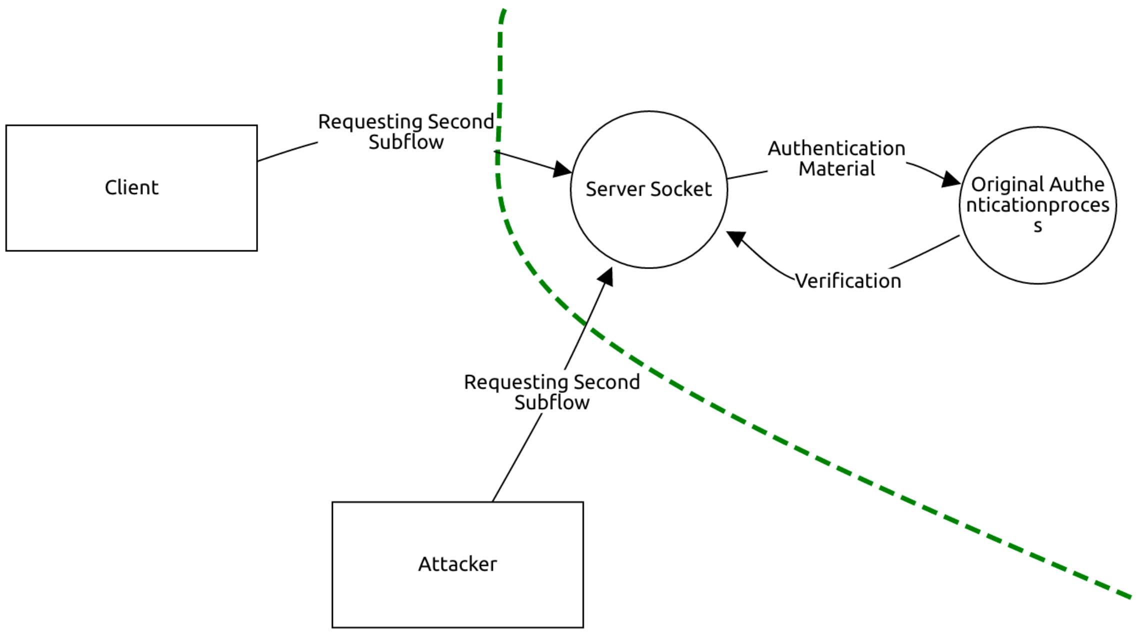

Client
Requesting Second Subflow
Server Socket
Authentication Material
Original Authe nticationproces s
Verification
Requesting Second Subflow
Attacker

Figure 5.14 : Threat model

To clearly represent the security problem, a threat model can be used. Figure 5.14 is the abstract threat model represented using data flow diagram of the security threat that has discussed in Chapter 2 about exchanging the keys in plain text format.

The client requests a second subflow to connect to the server using the MP_JOIN + SYN packet. By using the authentication process of the original MPTCP protocol, the server decides whether the request has come by the legitimate client. But as discussed in Chapter 2, the attacker can request to join to the connection using the MP_JOIN + SYN packet, by using the eavesdropped keys in the initial key exchange. Theoretically, the attacker can get access to the connection if the keys are correct.

But with the proposed solution in this research, the attacker should have another set of keys to authenticate the connection. Figure 5.15 shows the threat model drawn using data flow diagrams for the proposed solution. Both the client and the server should have the user space keys for the authentication. When the request is sent by the client to the server, the server has to use the MPTCP authentication process to authenticate the connections. But with the proposed solution, it has included another layer of an authentication mechanism. The authentication material created by XORing the token and external key has sent to the server by the client. The server compares the mentioned authentication material to evaluate the client request. If the authentication process of the proposed solution validates the client authentication material, then the MPTCP protocol allows the new subflow to initiate.

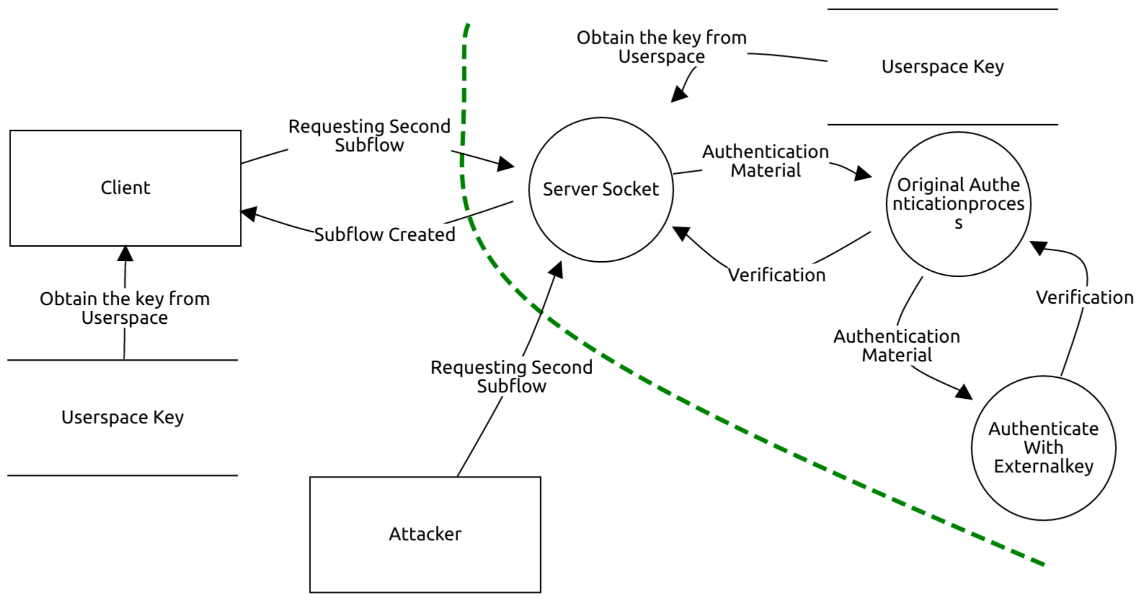

Obtain the key from Userspace
Userspace Key
Requesting Second Subflow
Client
Server Socket
Authentication Material
Original Authenticationprocess
Subflow Created
Verification
Obtain the key from Userspace
Verification
Authentication Material
Userspace Key
Requesting Second Subflow
Authenticate With Externalkey
Attacker

Figure 5.15 : Threat model of proposed solution

As the proof concept, by assuming that the key obtained from the user space is secure, it can be considered that the proposed mechanism provides a robust solution to authenticate the subflow generated by Multipath TCP.

5.4 Performance testing with the proposed kernel modifications

In the implementation of the proposed solution, the Linux kernel was modified by inserting new code segments, variables, and functions. Due to these modifications, there should be some change in the performance of the kernel. In this research, the performance of the kernel was not considered. But it is important to check whether there is any performance improvement or performance degradation due to the modifications done to the kernel.

A simple experiment was conducted to measure the performance of the modified kernel and compared it with some other kernels to evaluate the performance. The experiment was to send a file from one node to another and measure the time taken. Four files were generated using Linux terminal with 10MB, 20MB, 50MB, and 100MB capacities to transfer from one host to another. The main requirement of this experiment is to compare the performance of the original MPTCP with the modified MPTCP with the proposed solution by this research. Therefore all four files with different capacities were transferred and time was measured. Figure 5.16 shows the final result of the experiment. It has shown that the performance of both the modified MPTCP and original MPTCP are almost the same. But there is a considerable difference with the TCP and MPTCPsec. But both the TCP and MPTCPsec kernels have used only single subflow to transfer data. It can assume that the main reason for this can be the problems related to the optimization of the MPTCP kernel. Because the MPTCP kernel is still on the experiment level.

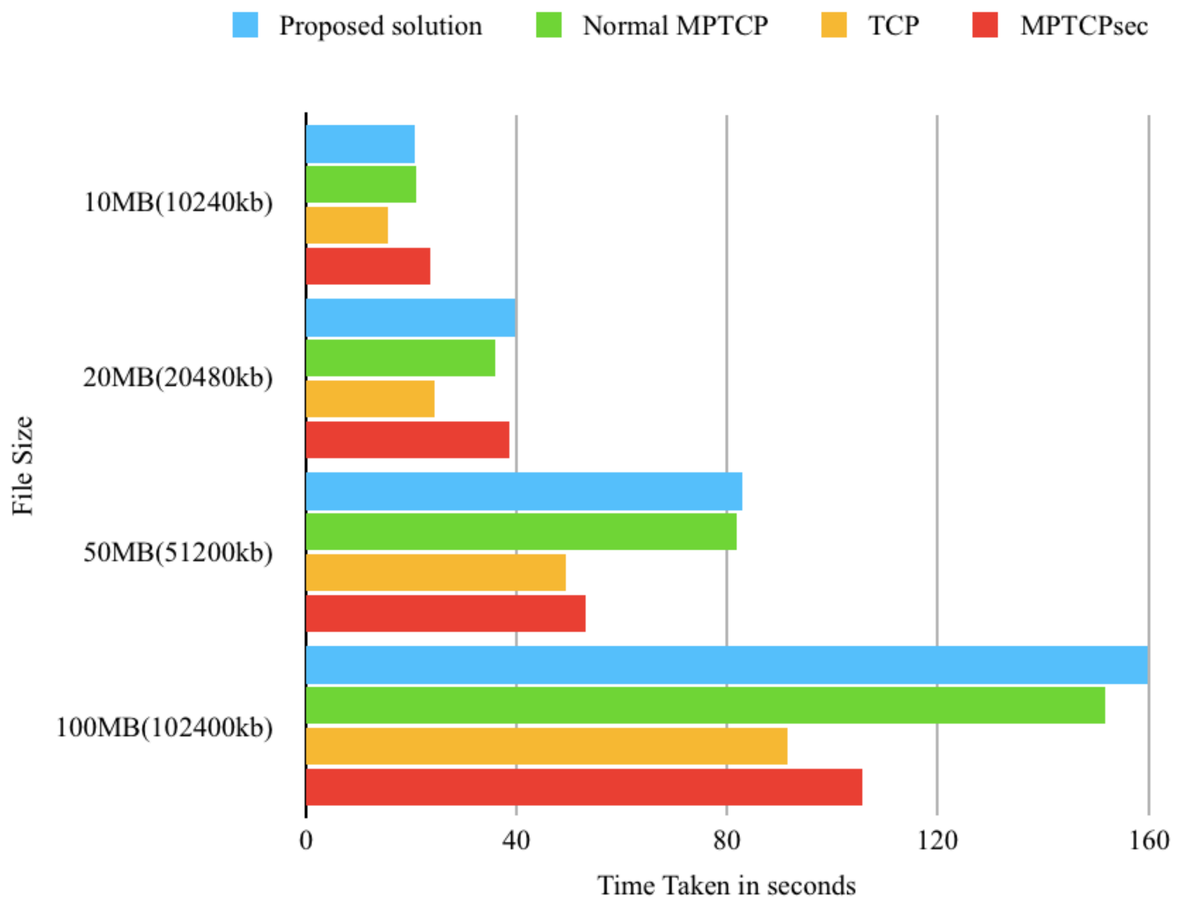


Figure 5.16 : Summarized results of performance test

Chapter 6: Conclusion and Future Work

The Transmission Control Protocol (TCP) is a widely used network protocol to make a reliable connection between two hosts. It has utilized a single network interface of a host to initiate the connection. But with the development of technology, most of the devices have more than one network interface, such as Ethernet, Wifi and so on. Multipath Transmission Control Protocol (MPTCP) has been introduced as an extension for the original Transmission Control Protocol to use more than one network interface when creating a single network connection.

Multipath TCP has not altered the headers of the original TCP protocol. Instead, it has used the TCP options space in the TCP header to send the additional information related to Multipath TCP. Multipath TCP contains a number of options. Such as MP_CAPABLE to check whether the hosts are compatible with MPTCP and MP_JOIN to join a new subflow the connection. In MPTCP, the network connections created between two hosts are known as subflows. MPTCP shares a set of keys when initiating the first subflow. These keys are used to authenticate the subflows that are created later. The important fact to consider is that the keys are in plain text format.

Though Multipath TCP has advantages such as redundancy and increase of throughput, it has created a number of security threats. When studying the security threats, it has been identified that one of the main reasons for the security threats is the exchange of authentication keys in plain text format. Attackers can create a number of attacks by using these keys.

Some of the security issues were identified and solutions were proposed for specific scenarios such as ADD_ADDR attack[4]. However, this does not provide the solution for the general issue of sharing keys in plain text format and use them to authenticate the subflows. Still, some solutions are on the development stage. If the authentication process of the subflows gets improved, the security issues of MPTCP can be minimized. Therefore some alternate methods to authenticate the subflows has to be explored and that is the main objective of this research. As a suggestion for that, this research was focused to explore the methods of using the information from user space to authenticate the subflows.

The research was conducted in three main stages as mentioned below.

- Obtain the keys from user space and transfer to the kernel space as the network connection is established in the kernel space of the operating system.
- Generate the authentication material using the user space key and send it from the client to the server.
- Authenticate the subflow using the authentication material by the server.

Before altering the MPTCP, the behavior of the protocol was identified by conducting several experiments. In order to implement the proposed solution, the important functions of the MPTCP kernel were identified and modified as required. After exploring several avenues, a method was identified to import the key from user space to the kernel space. Then the key was used to generate the authentication material to be sent to the server. At the server, the subflow was authenticated by using the authentication material sent by the client. If the authentication material is valid, the subflow was created. Otherwise, the subflow is dropped by MPTCP. As a proof of concept, this proposed solution has shown that the user space information can be used to authenticate the newly generated subflows in Multipath TCP.

In Multipath TCP, backward compatibility is an important feature. If any of the hosts are not compatible with Multipath TCP, it will automatically change to the original TCP connection. Therefore this proposed solution should be backward compatible. As mentioned in Chapter 5, the current implementation will change to original Multipath TCP if both the hosts are not using external keys to authenticate the connection. This has to be improved to switch to original Multipath TCP if one host is configured with the proposed solution and the other is not configured with the proposed solution. In this regard, the optimization of the kernel was not considered. Therefore the performance of the network connections is not at the desired level.

Other than the advantages, there are some disadvantages by using this user level information, which is the Transport layer depends on the Application layer when initiating the network connection. It violates the abstraction of the OSI layers.

6.1 Future Work

With the proposed solution it has shown that the key from user space or the application level can be used to authenticate the subflows generated by MPTCP. Which means that this solution can be improved to a state where the application has the control to use multiple paths in the network connection or not. Assume if there is a huge amount of data to transfer between two nodes and the application can decide to use multiple paths to improve the throughput. On the other hand, if there is only a very small amount of information to transfer, such as to open a web page, there is no need to incorporate multiple paths in the connection. In such cases, the application can decide not to use multiple paths. Therefore this concept can be extended to a level which the application has the control of using MPTCP or TCP for the connection.

References

[1] Postel, J. (1981). Transmission Control Protocol. Marina del Rey, Calif: Inst.

[2] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, “TCP Extensions for Multipath Operation with Multiple Addresses,” RFC Editor, RFC6824, Jan. 2013.

[3]"Observing Siri : the three-way handshake — MPTCP", *Blog.multipath-tcp.org*, 2018. [Online]. Available: http://blog.multipath-tcp.org/blog/html/2014/02/24/observing_siri.html.

[4] F. Demaria, “Security Evaluation of Multipath TCP,” p. 87

[5] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar, “Architectural Guidelines for Multipath TCP Development,” RFC Editor, RFC6182, Mar. 2011.

[6] M. Bagnulo, “Threat Analysis for TCP Extensions for Multipath Operation with Multiple Addresses,” RFC Editor, RFC6181, Mar. 2011.

[7] M. Bagnulo, C. Paasch, F. Gont, O. Bonaventure, and C. Raiciu, “Analysis of Residual Threats and Possible Fixes for Multipath TCP (MPTCP),” RFC Editor, RFC7430, Jul. 2015.

[8] M. Jadin, G. Tihon, O. Pereira, and O. Bonaventure, “Securing multipath TCP: Design & implementation,” in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, Atlanta, GA, USA, 2017, pp. 1–9.

[9] C. Paasch and O. Bonaventure, “draft-paasch-mptcp-ssl-00 - securing the multipath tcp handshake with external keys,” 2013.

[10] J. Díez, M. Bagnulo, F. Valera, and I. Vidal, “Security for multipath TCP: a constructive approach,” *International Journal of Internet Protocol Technology*, vol. 6, no. 3, p. 146, 2011.

[11] D.-Y. Kim and H.-K. Choi, “Efficient design for secure multipath TCP against eavesdropper in initial handshake,” in *2016 International Conference on Information and Communication Technology Convergence (ICTC)*, Jeju, 2016, pp. 672–677.

[12] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller, “Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS),” RFC Editor, RFC4492, May 2006.

[13] D. McGrew, “An Interface and Algorithms for Authenticated Encryption,” RFC Editor, RFC5116, Jan. 2008.

[14] O. Bonaventure, “MPTLS : Making TLS and Multipath TCP stronger together draft-bonaventure-mptcp-tls-00,” 2015.

[15] A. Hamza, M. I. Lali, and F. Javid, “Study of MPTCP with Transport Layer Security,” *Emerging Technologies*, p. 6.

[16] “Eddy - 2007 - TCP SYN Flooding Attacks and Common Mitigations.pdf.” .

[17] A. Bittau, D. Giffin, M. Handley, D. Mazieres, Q. Slack and E. Smith, “Cryptographic protection of TCP Streams (tcpcrypt) draft-ietf-tcpinc-tcpcrypt-10,” 2018.

[18] A. Bittau, D. Giffin, M. Handley, D. Mazieres and E. Smith, “TCP-ENO: Encryption Negotiation Option draft-ietf-tcpinc-tcpeno-18,” 2018.

[19] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” RFC 5246, 2018.

[20]"MultiPath TCP - Linux Kernel implementation : Main - Home Page browse", *Multipath-tcp.org*, 2018. [Online]. Available: https://www.multipath-tcp.org.

[21] "MultiPath TCP - Linux Kernel implementation : Users - Apt Repository browse", *Multipath-tcp.org*, 2018. [Online]. Available: https://multipath-tcp.org/pmwiki.php/Users/AptRepository.

[22]"Downloads – Oracle VM VirtualBox", *Virtualbox.org*, 2018. [Online]. Available: https://www.virtualbox.org/wiki/Downloads.

[23]D. Group, "Welcome! - The Apache HTTP Server Project", *Httpd.apache.org*, 2018. [Online]. Available: https://httpd.apache.org/.

[24]"MultiPath TCP - Linux Kernel implementation : Users - Do It Yourself browse", *Multipath-tcp.org*, 2018. [Online]. Available: https://multipath-tcp.org/pmwiki.php/Users/DoItYourself.

[25]"MPTCPsec-team / MPTCPsec — Bitbucket", *Bitbucket.org*, 2018. [Online]. Available: https://bitbucket.org/account/user/mptcpsecteam/projects/PROJ.

[26]"Wireshark · Go Deep.", *Wireshark.org*, 2018. [Online]. Available: https://www.wireshark.org.

[27] "proc(5) - Linux manual page", Man7.org, 2019. [Online]. Available: http://man7.org/linux/man-pages/man5/proc.5.html. [Accessed: 19- Feb- 2019].

[28]"netlink(7) - Linux manual page", Man7.org, 2019. [Online]. Available: http://man7.org/linux/man-pages/man7/netlink.7.html. [Accessed: 19- Feb- 2019].

[29]"struct sockaddr_in, struct in_addr", Gta.ufrj.br, 2019. [Online]. Available: https://www.gta.ufrj.br/ensino/eel878/sockets/sockaddr_inman.html. [Accessed: 19- Feb- 2019].

[30] M. Jadin and G. Tihon, “Secure MultiPath TCP,” p. 118.

[31]"Use Multipath TCP to create backup connections for iOS", Apple Support, 2019. [Online]. Available: https://support.apple.com/lv-lv/HT201373. [Accessed: 21- Jan- 2019].

[32]"Improving Network Reliability Using Multipath TCP | Apple Developer Documentation", Developer.apple.com, 2019. [Online]. Available: https://developer.apple.com/documentation/foundation/urlsessionconfiguration/improving_network_reliability_using_multipath_tcp. [Accessed: 21- Jan- 2019].

[33]"Chapter6.Virtual Networking", Virtualbox.org, 2019. [Online]. Available: https://www.virtualbox.org/manual/ch06.html.

Appendix A - Code Modifications

This appendix contains the information related to the code modifications done to the Linux kernel. As described in Chapter 4, the number of inbuilt functions of Linux kernel have modified in order to implement the proposed solutions. All the modifications done to the kernel files are given below.

mptcp.h

This header file is located in the */include/net* of the MPTCP Linux kernel. Two new variables were introduced to the kernel to store the token generated by the MPTCP kernel and to store the key received from the user space. Line number 5 and 6 of Listing A.1 shows the relevant code segment of the declaration of new variables in the kernel. A new function has to be defined to generate the XORed version of the token by taking the external key and token generated by MPTCP as the inputs. Line number 10 of the Listing A.1 shows the function prototype defined in the header file.

```
1.  /*
2.      varibale to store token
3.      variable to store external key
4.  */
5.  extern int token_tnb; //store token
6.  extern long external_key_tnb; //external key
7.
8.  //XOR funtion with the token and the key
9.
10. int xor_token_key_tnb(int token, int key);
```

Listing A.1 : mptcp.h header file

af_inet.c

af_inet.c is one of the most important files. Transferring user space key to the kernel space was done by modifying the functions available in the *af_inet.c* file. Listing A.2 shows the *sockaddr_in* data structure which used when defining TCP sockets. *sin_zero* is the variable shows in line number 10, which used to transfer external key from the user space to kernel space.

```
/* Structure describing an Internet (IP) socket address. */
#if  __UAPI_DEF_SOCKADDR_IN
#define __SOCK_SIZE__ 16    /* sizeof(struct sockaddr)  */
struct sockaddr_in {
  __kernel_sa_family_t  sin_family; /* Address family   */
  __be16    sin_port; /* Port number      */
  struct in_addr  sin_addr; /* Internet address   */

  /* Pad to size of `struct sockaddr'. */
  unsigned char   __pad[__SOCK_SIZE__ - sizeof(short int) -
      sizeof(unsigned short int) - sizeof(struct in_addr)];
};
#define sin_zero  __pad   /* for BSD UNIX comp. -FvK  */
#endif
```

Listing A.2 : sockaddr_in data structure

There are two functions in the *af_inet.c* file which is important in obtaining the user space key to the kernel space. First one is the *inet_bind()* function, which is used by the server to bind the TCP socket. This function has access to the *sockaddr_in* data structure which mentioned in the above section. Therefore the line segment shown in Listing A.3 was added to the function to obtain the key from the user space on the server.

```
//server uses this function
 kstrtol(addr->sin_zero, 10, &external_key_tnb);
```

Listing A.3 : inet_bind()

The second most important function in *af_inet.c* is *__inet_stream_connect()* which is used by the client when creating the TCP socket. With this function, the *sockaddr_in* data structure can be accessed and the user space key can be passed to the kernel level. Listing A.4 shows the code segment which included in the *__inet_stream_connect()* function.

```
struct sockaddr_in *addr = (struct sockaddr_in *)uaddr;
kstrtol(addr->sin_zero, 10, &external_key_tnb);//at client
```

Listing A.4 : __inet_stream_connect()

mptcp_ctrl.c

The function which is used to XOR the external key with the token generated by MPTCP protocol is defined in the *mptcp_ctrl.c* file. Listing A.5 shows the code of the function.

```
//XOR funtion with the token and the key
int xor_token_key_tnb(int token, int key){
    //this function XOR the token and the external key.
    return token^key;
}
```

Listing A.5 : __inet_stream_connect()

mptcp_output.c

The function *mptcp_syn_options()* is used by the client to create the MPTCP SYN packet. According to the proposed solution, the client needs to generate the XORed token and send it to the server to authenticate the new subflow. Therefore the relevant code segment can be included in *mptcp_syn_options()* function to perform the task. Listing A.6 shows the code segment included to execute the intendant operation. The *if* statement in line number 1 of Listing A.6 is used to check whether the external key is set in the socket or not. If the external key is not assigned, then the protocol will use the original MPTCP in order to achieve backward compatibility of the proposed solution.

```
1. if(external_key_tnb==0||external_key_tnb==“0"){
2.    opts->mp_join_syns.token = mpcb->mptcp_rem_token;
3. }else{
4.     opts->mp_join_syns.token     =     xor_token_key_tnb(mpcb-
   >mptcp_rem_token,external_key_tnb);
5. }
```

Listing A.6 : mptcp_syn_options()

mptcp_input.c

The *mptcp_input.c* file contains the function called *mptcp_parse_options()*, which is used by the server to parse the MPTCP options to a higher level. Therefore this can be used to compare the XORed token send from the client and authenticate the connection. Code segment shown in Listing A.7 is used to compare the XORed tokens and establish the connection.

```
if(external_key_tnb==0||external_key_tnb=="0"){
                mopt->mptcp_rem_token = mpjoin->u.syn.token;
            }else{
                if (mpjoin->u.syn.token == xor_token_key_tnb(token_tnb,external_key_tnb)) {
                mopt->mptcp_rem_token = token_tnb;
                break;
                } else {
                pr_info("XORed token doesnot matched. MP_JOIN dropped");
                }
            }
```

Listing A.7 : mptcp_parse_options()

Appendix B - MPTCP with iOS

As mentioned in Chapter 3, Apple iOS has included MPTCP on the operating system [31]. To understand the behavior of MPTCP on iOS some experiment was conduction and the summarized version of the results was included in this Appendix.

For MPTCP to use, there should be more than one network interface. Mobile phones have a WiFi interface and cellular interface. Therefore both WiFi connection and the cellular connection or the mobile data connection is considered when conducting the experiments. Apple iOS has a feature called WiFi-Assist [32] which uses mobile data to support WiFi connection. Therefore in this experiment, Wifi_assist also considered as a variable to check whether there is a connection between MPTCP and the Wifi-Assist. Apple has mentioned that they are using MPTCP for the Siri application [3]. Therefore a question was asked from Siri and the network packets of iPhone were captured using Command line tools of macOS. Other than that the official web site of MPTCP research group which is configured with MPTCP, was opened using Safari web browser of iPhone and network packets were captured. Table B. 1 shows the summarized results of the experiment conducted. The symbol "✓" shows the options which were activated and symbol "✗" shows that the options were inactivated. The final column of Table B.1 shows the result of each and every experiment.

According to the results shown in Table B.1, it is clear that the MPTCP protocol is used only for the Siri application, but not to load the other web site. There is no direct connection between the Wifi Assist feature with the usage of MPTCP in Siri application.

Experiment No	Wifi	Mobile data	Wifi Assists	Siri	Web Page	Result (Use of MPTCP)
1	✓	✓	✓	✓		✓
2	✓	✓		✓		✓
3	✓		✓	✓		✓
4	✓			✓		✓
5		✓	✓	✓		✓
6		✓		✓		✓
7	✓	✓	✓		✓	✗
8	✓	✓			✓	✗
9	✓		✓		✓	✗
10	✓				✓	✗
11		✓	✓		✓	✗
12		✓			✓	✗

Table B.1 : Results of MPTCP iOS experiment