UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Secure MultiPath TCP

Design & Implementation

Dissertation presented by
Mathieu JADIN , Gautier TIHON

for obtaining the Master's degree in
Computer Science and Engineering

Supervisor(s)
Olivier BONAVENTURE, Olivier PEREIRA

Reader(s)
Fabien DUCHÊNE, Florentin ROCHET

Academic year 2015-2016

Abstract

MultiPath TCP (MPTCP) is a recent transport protocol that was developed to take advantage of the multiple paths in the network that often exist between peers, and to use efficiently all the interfaces (e.g., Wifi and Ethernet). A MPTCP connection is composed of multiple TCP (Transmission Control Protocol) connections, called subflows. Compared to TCP that is restricted to a single path per connection, MPTCP improves the resource use in the network, the throughput and the resilience to network failure. It also enables to be more resilient against DoS attacks since MPTCP can easily close a subflow (i.e., a path) and re-establish another one. MPTCP is thus stronger than TCP against many Denial of Service (DoS) attacks. But despite growing adoption of MPTCP, it remains mostly unauthenticated and unencrypted. This is due to the fact that there is no encryption scheme integrated in MPTCP. There is only TLS that can be used above MPTCP in the same way than above TCP. And there is even less a mechanism to easily negotiate them. This is therefore a good opportunity to directly design a standard to negotiate these encryption schemes instead of letting people going their own way.

This document thus defines a mechanism that enables easier negotiation of encryption schemes over MPTCP: MPTCP Encryption Scheme Negotiation (MPTCPesn). Designed to be modular and easy to update, it offers a good basis for the development of cryptographic protocols for MPTCP. Indeed, adding new encryption schemes that can be negotiated with MPTCPesn can be done quite easily. But the encryption schemes themselves were still missing. Therefore, in this document, we have also designed MPTCPsec, the first cryptographic protocol specially designed for MPTCP (and even integrated in it), negotiable with MPTCPesn. It enables two hosts to exchange data by means of a secure MPTCP connection. MPTCPsec is application independent, fast and it takes advantage of the MPTCP's properties (e.g., having several subflows). MPTCPsec provides the same security properties than TLS, except the identity authentication (achieved by certificates). But it is resistant to many more DoS attacks than TLS.

MPTCPsec brings three main improvements, compared to simply running TLS above MPTCP. Firstly, the whole connection does not need to be closed when an attack is detected on a subflow (i.e., authentication failure). Only this subflow is closed, and another one can be opened to replace it. It is a great improvement since an attacker can no more close the whole connection by modifying only one byte of any segment in any subflow, as he could do against TLS. Furthermore, and this is the second improvement, an attacker is unable to interfere with a subflow without being on the path used by this subflow, even if he controls other subflows. He can neither prevent the establishment of subflows he does not control. This is due to the authentication of the MPTCP options. In fact, the most he can do is to close the subflows he is on, or to drop some or all the segments of these subflows (but they will be re-injected on other subflows). The third improvement is that segments can be decrypted as soon as they are received, even if a segment with lower DSN is not already received (i.e., segments out-of-sequence can be decrypted).

Once the design has been finished, we have implemented the main parts of our protocol, directly in the Linux kernel, in order to properly incorporate it in the MPTCP implementation. First of all to prove to ourselves that our design makes sense and then to be able to compare our solution with TLS above MPTCP with some performance tests. Both objectives are fulfilled and the results are very satisfying. Especially as our implementation is not fully optimized, contrary to TLS implementation (OpenSSL) that has been optimized and improved for several years now. Our most important achievements are that MPTCPsec is a little faster in term of data processing and that the data flow continues even if an attacker is on one subflow and tries to modify segments (while TLS connection is closed). The results also show that even if MPTCPsec has a little smaller throughput than TLS (and we explain this difference), it remains very competitive.

ACKNOWLEDGEMENT

It gives us great pleasure in acknowledging the advice of our thesis supervisors, Professor Olivier Pereira and Professor Olivier Bonaventure, their comments on our drafts of the report and their help for the most challenging problems. They were always ready to listen to us and often available for us.

It is with immense gratitude that we acknowledge the support and help of Fabien Duchêne who has been regularly disturbed in his work to answer our questions. We also thank his colleagues that we have bothered each time we have come to talk with Fabien Duchêne.

A big thanks to Pénélope Drizis for her support and to have patiently corrected the English of each part of our report, even if sometimes she did not understand the content. Thank to these corrections, our English is now better... we hope !

We also thanks Marie-Hélène for her support during long working nights and for some delicious meals that were waiting (one of) us after a hard working day.

Finally, we want to thank Anthony Gégo for the installation of the server and his help during the configuration of the server.

Contents

Acronyms

AEAD Authenticated Encryption with Associated Data.

AES Advanced Encryption Standard.

CA Certificate Authority.

DoS Denial of Service.

DSN Data Sequence Number.

DSS Data Sequence Signal.

GCM Galois Counter Mode.

HMAC Hash-based MAC.

IDSN Initial Data Sequence Number.

MAC Message Authentication Code.

MITM Man-In-The-Middle.

MPTCP MultiPath TCP.

MPTCPesn MPTCP Encryption Scheme Negotiation.

MPTCPsec MPTCP secure.

NAT Network Address Translation.

PMS Pre-Master Secret.

PRK Pseudo-Random Key.

SID Session ID.

SSN Subflow Sequence Number.

TCP Transmission Control Protocol.

TCP-ENO TCP Encryption Negotiation Option.

TLS Transport Layer Security.

Introduction

Transmission Control Protocol (TCP) [1], by default, does not include any cryptographic technique to authenticate and encrypt data. Three types of solutions have been proposed to improve the security of TCP. A first approach is to tune the TCP stack to prevent some segment injection attacks. Examples of this approach may be found in [2] or [3]. Another approach is to add authentication to the TCP protocol. The TCP MD5 option defined in [4] was the first example of such an approach. The TCP-AO option defined in [5] and [6] is a more recent example. These two solutions were designed to protect long-lived TCP connections such as BGP sessions from segment injection attacks. They assume that a secret is shared among the communicating hosts. A third approach is to extend TCP to include the cryptographic techniques directly inside the TCP stack. Tcpcrypt defined in [7] has followed this approach.

Another possibility is to develop an application layer protocol that allows to encrypt and authenticate the data exchanged between applications running on different hosts, on top of TCP. Transport Layer Security (TLS) [8] was designed for this purpose. Various documents have analysed the security of the TLS protocol from different viewpoints [9]. Over the years, various extensions to TLS have been developed and deployed. Besides attacks on the cryptographic algorithms or their implementations, TLS is also vulnerable to some forms of attacks that affect the underlying TCP protocol [1].

Something was still missing: the standardization of the negotiation of encryption schemes. In order to solve this problem, the tcpinc working group designed TCP-ENO [10]. This is a new TCP option kind providing out-of-band, fully backward-compatible negotiation of encryption. TLS and tcpcrypt have been adapted so that their use can be negotiated by TCP-ENO (respectively [11] and [12].

However TCP is restricted to a single path per connection. MultiPath TCP (MPTCP) [13] was developed to take advantage of the multiple paths that often exist between peers, and to use efficiently all the interfaces (e.g., Wifi and Ethernet). It improves the resource use in the network, the throughput and the resilience to network failure. It also enables to be more resilient against DoS attacks since MPTCP can easily close a subflow (i.e., a path) and re-established another one. MPTCP is thus stronger than TCP against many DoS attacks.

Despite growing adoption of MPTCP, it remains mostly unauthenticated and unencrypted. This is due to the fact that there is no encryption scheme integrated in MPTCP (except that TLS can be used above MPTCP in the same way than above TCP, but it is not fitted specially to MPTCP), and there is even less a mechanism to easily negotiate them. This is therefore a good opportunity to directly design a standard to negotiate these encryption schemes instead of letting people going their own way. This standard would allow incremental deployment of future encryption schemes. This document builds this negotiation mechanism, the MPTCP Encryption Scheme Negotiation (MPTCPesn). It is mainly TCP-ENO fitted to MPTCP.

As well as TLS has been adapted to be negotiable with TCP-ENO, it could be adapted to be negotiable with MPTCPesn. But TLS does not take advantage of MPTCP, it is not fitted to

MPTCP. This document builds an encryption scheme specially designed for MPTCP, integrated in it, that complies to MPTCPesn: MPTCPsec (MPTCP secure). There are the three main improvements aimed by MPTCPsec, compared to TLS above MPTCP.

- TLS has to close the whole connection when an attack (or even only an attempt of it) is detected while MPTCPsec only needs to close the subflow on which this attack has been detected.
- TLS only protects application's data (with authentication and encryption) while MPTCPsec also protects the connection itself (i.e., MPTCP options are secured). It means that an attacker cannot close the connection, cannot interfere with a subflow on a path he does not control and can barely do anything else than closing subflows established on paths he controls.
- TLS must wait for segments to be in sequence in order to decrypt them (because MPTCP delivers data to TLS in the same order than they are sent by the other host's application) while MPTCPsec enables to decrypt them as soon as they are received by the host (even out-of-order segments).

This document explores first of all the state of the art (Part I), starting by the cryptography, TCP and MPTCP (Chapter 1) that are essential bases to understand this document. Then, it details the tcpinc working group's work, TCP-ENO, and the TLS and tcpcrypt versions adapted in order to be negotiable with TCP-ENO (Chapter 2). The second part (Part II) specifies the designs of both MPTCPesn (Chapter 3) and MPTCPsec (Chapter 4). But there is a gap between theory and practice... This is why the last part (Part III) details the implementation of MPTCPesn (Chapter 5) and the implementation of MPTCPsec. The latter is divided in two chapters, the first one (Chapter 6) gives an implementation that ensures data privacy (i.e., data cannot be modified or read by an attacker, but they can be truncated) while the second one (Chapter 7) improves it to ensure data integrity and to protect the connection itself. Finally, this document presents some performance measures (Chapter 8) in order to compare our solution, MPTCPsec, with TLS above MPTCP.

Part I

State of the art

Chapter 1

Network and cryptography bases

In order to start on solid bases, let us refresh our memory. This document assumes that the reader knows TCP (Transmission Control Protocol), has some knowledge of MultiPath TCP and has already heard about cryptography. This chapter thus reminds or deepens our knowledge thereupon, focusing mainly on what we need to know to understand the main part of this document.

1.1 Cryptography

Since cryptography is a large and complex field, and as the purpose of this document is not to teach you cryptography into details, this section provides a brief overview of cryptography. More details are available in [14], whose this section is based on.

Cryptography is the science of securing information (exchanges or storing). It requires that only people for whom the information is intended know a given secret. The Kerckhoffs' principle (presented in [15] in 1883) says that *only the key should be secret* (i.e., a password), not the protocol used. It enables to develop powerful public algorithms, tested and approved by a wide community.

This security includes four aspects (among others):

- **Data confidentiality:** the information cannot be understood by anyone for whom it was unintended (e.g., data exchanged by two parties cannot be read by a third party). This includes the information itself and, ideally, any of its properties (e.g., its length).
- **Data integrity:** any undesired alteration of the data is detectable. It comes with the data authenticity, which ensures that the data are sent by the owner of the key and not altered by anyone else.
- **Identity authentication:** the parties' identities (i.e., owners of the keys) are confirmed (e.g., a sender `S` can *sign* its messages so that the receiver can be sure that `S` is the sender of these messages).
- **Non-repudiation:** someone who sends/creates some information cannot deny later that he has sent/created it.

The first aspect is the most widely known and is provided by encryption (and decryption) protocols. These protocols can be classified in two categories: symmetric ciphers (with symmetric keys) and asymmetric ciphers (with public and private keys). The following security aspect, the data integrity (and authenticity), can be achieved by the Message Authentication Code (MAC), a tag derived from messages and computed with symmetric keys. Even if the MAC does not ensure the real identity of the messages' author, it still ensures he knows the right secret key. Certificates,

which are a kind of official and digital documents, fill the gap by binding virtual identities (i.e., owners of secret asymmetric keys, known only to themselves) to real identities (e.g., companies). Finally, digital signatures provide (among others) the last aspect: the non-repudiation. They are basically the asymmetric version of the MAC (i.e., the tag is computed with asymmetric keys). All theses schemes will be explained into details in the subsequent sections.

As many computer scientists are not experienced cryptographers, their manipulations and combinations of the above algorithms have often led to security issues. The cryptographers have thus decided to develop Authenticated Encryption with Associated Data (AEAD) algorithms that combine efficiently several cryptographic algorithms in order to provide data confidentiality, data integrity and data authenticity. All in one, in a black box easier to use.

But let us begin with some considerations about the differences between theory and practice, and with some general notions and definitions (e.g., what is an oracle).

1.1.1 Keys, security level and oracles

As said above, only the keys should be secret. The security levels of cryptographic algorithms are therefore always limited at least by the keys themselves. And more precisely by the length of these keys if they are randomly chosen[1]. Indeed, an attacker is always able to try keys at random with the hope of finding the good one by luck. The longer the key, the more possibilities there are. The security level of an algorithm is thus related to the number of bits needed to represent all the possible keys. For example, a key that can be any of all the combinations of n bits gives a security level of n bits, but if this n-bits key can only take even values, the security level is of $n - 1$ bits.
But, in fact, some algorithms do not use the keys at their maximal potential and are vulnerable to more efficient attacks than trying all the possibilities. The security level must thus be estimated. For instance, the Diffie-Hellman key exchange algorithm is based on the difficulty of factorizing large numbers, not directly on the length of the keys [16].

In a perfectly secure world, an attacker should never find the right key (i.e., break the security), but in theory there are always tiny probabilities he succeeds. Therefore, we speak about *computational security*, i.e., not impossible to break, but infeasible with today's computational power. Moreover, if the computational power of the host and its attacker increases, for an equivalent effort from both of them, the success probabilities of the attacker should decrease. The attacker should, with *high probabilities*, never find the right key. Today's security level should be at least of 112 bits, but it is better to aim to 128 bits, according to NIST recommendations[2].

The attacker can be helped to find the key, for example by letting him try as many keys as he wants. This is typically what happens on insecure websites when you want to log in: you can try passwords until finding the good one. It is called an *oracle*. When high security is needed, e.g., for your mobile phone, you can only try several passwords and then you are blocked for a while (or forever). But other kinds of oracles exist, such as letting an attacker get the encryption (or signature) of a text of its choice. Most of today's cryptographic protocols are theoretically secure against attackers with oracles, but in practice, implementations of these protocols can lead to security issues, due to oracles (see examples of attacks in [17, 18]).

[1] In the case of a human password, the key "12345" is obviously less secure than the key "P$-4v" of the same length. The attacker knows that it is a human that has chosen the key and will therefore try the common keys first.

[2] `https://www.keylength.com/en/4/`

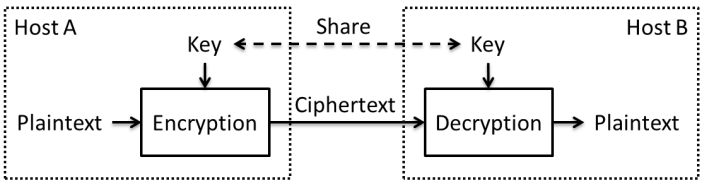


Figure 1.1: Encryption and decryption of data with a symmetric cipher.

1.1.2 Symmetric ciphers

Encryption protocols transform the data (called the plaintext) into something meaningless (called the ciphertext) so that nobody can know what are the data behind. Decryption protocols enable to recover the data encrypted by their associated encryption protocols. Since the protocols are public, a key is introduced in the computation in order to prevent anyone from being able to decrypt the ciphertexts.

Let us take the Caesar's cipher as an example (it is insecure but easier to understand). The encryption protocol shifts the letters in the alphabet (e.g., $A \to E$, $B \to F$, $C \to G$) and the decryption protocol shifts the letters in the reverse direction. The key is the number by which shifting the letters. There are 26 possible keys, and $2^4 < 26 < 2^5$: the level of security is 4. For a key that equals to 3, the encryption of "Hello" gives "Khoor" and the decryption with this key gives back the "Hello" message.

Symmetric-key algorithms use the same keys for both encryption and decryption of data, as shown in Figure 1.1. Therefore the keys must be known by those for whom the data are intended (and by the sender himself). This is the main issue of symmetric cipher. It is indeed difficult to securely exchange the keys.

There are two types of symmetric-key algorithms:

- `Stream ciphers` encrypt data byte per byte, one at a time
- `Block ciphers` encrypt data by block of fixed size (commonly 64 bits). If there are not enough data to fill the last block, some padding is added

The symmetric ciphers can be used to build other cryptographic primitives like the Message Authentication Code (Section 1.1.4).

1.1.3 Asymmetric ciphers & key-agreement scheme

Public-key algorithms (also called asymmetric algorithms) use public keys for encryption and private keys for decryption. As their names indicate, the public key can be sent in clear without any security consideration ; only the private key, for decryption, must be kept secret by its owner. Contrary to symmetric ciphers, there is no need to previously exchange keys. But the computational cost is higher (100-1000 times less efficient). Asymmetric ciphers are thus only used to exchange small data blocks, typically the transfer of a symmetric encryption key.

The encryption and decryption protocols are represented on Figure 1.2. Anybody having the public key can encrypt data, but the ciphertexts can be decrypted only with the private key.

The asymmetric ciphers can be used to agree on a shared secret, as the well-known Diffie–Hellman key exchange scheme. This is a `key-agreement scheme`, represented on Figure 1.3. The combination of the public key of the remote host with the local private key gives a secret. If the other host does the same (with the corresponding keys), it will compute the same secret. This secret can be used as a symmetric encryption key.

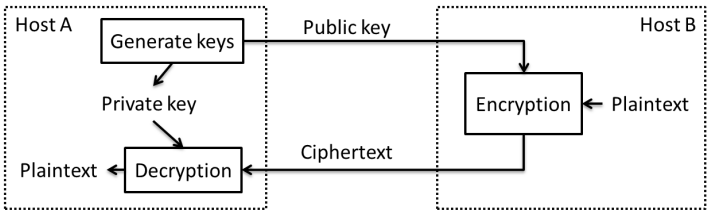


Figure 1.2: Encryption and decryption of data with an asymmetric cipher.

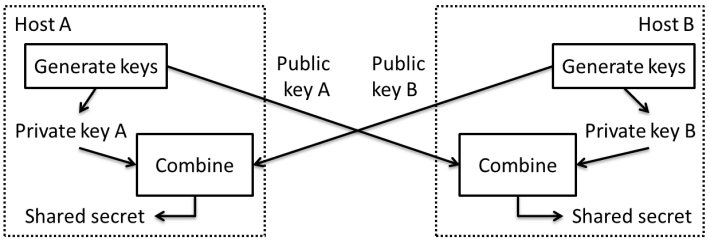


Figure 1.3: Key-agreement between two hosts in order to agree on a shared secret.

But an attack against these protocols still exists: the Man-In-The-Middle attack. This attack can be prevented with certificates and is explained in Section 1.1.6.

1.1.4 Message Authentication Code (MAC)

In order to enable the receiver to detect some undesired modifications of the data, there must be a kind of redundancy of these data. It can be done by adding to this message a tag (also called a digest) computed from the message itself.
In practice, the host A computes the tag of the data it wants to send to host B and puts the result at the end of the message. Upon reception of this message, host B will also compute the tag of the message and it will compare the result with the tag added at the end of the message. If both tags are equal, the message is authentic ; otherwise host B knows that some undesired modifications have been done.

Among protocols doing that, there are the well-known `hash` functions. These functions take data of arbitrary size as input and output data of fixed size, the tag. Hash function are efficient to detect transmission errors but it is not enough to ensure security against an active attacker. The attacker can change the data and then, he can compute the new tag to replace the old one in the message (since protocols are always public). Therefore, it is needed to include some secret values in the computation of this tag. Adding keys brings us data authenticity, since only the other host knows the key. This is why these tags are usually called `authentication tag`, or even `Message Authentication Code` (MAC). The latter, along with symmetric ciphers, uses symmetric keys.

A well known MAC, based on hash functions, and thus called Hash-based MAC (HMAC, defined in [19]), can be computed with the following formula :

$$HMAC(K, m) = H\Big((K' \oplus opad) \mid\mid H\big((K' \oplus ipad) \mid\mid m\big)\Big)$$

with

- H, a cryptographic hash function,
- K, the secret key,

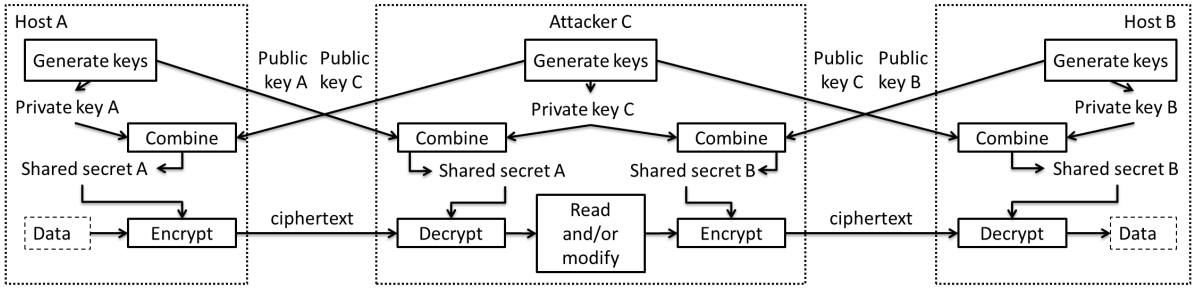


Figure 1.4: Man-In-The-Middle attack. Neither host A nor host B knows that the attacker is between them and can read and/or modify data.

- m, the message to be authenticated,
- K', another secret key, derived from the original key K (by padding K to the right with extra zeroes to the input block size of the hash function, or by hashing K if it is longer than that block size),
- *opad*, the outer padding (0x5c5c5c... 5c5c, one-block-long hexadecimal constant),
- *ipad*, the inner padding (0x363636... 3636, one-block-long hexadecimal constant),
- $||$, the notation of concatenation,
- $\oplus$, the notation of exclusive or (XOR).

1.1.5 Digital Signature

A digital signature is a MAC (Section 1.1.4) done with asymmetric keys. It is computed with a private key, and verified with the corresponding public key. It follows that everybody knowing the public key can verify that the tag is correct, but only the owner of the private key is able to compute this tag. Therefore, if the tag is correct, only the owner of the private key can have sent it (i.e., non-repudiation) and no one can have modified this message without corrupting the tag (i.e., integrity). Along with the MAC, it also brings data authenticity. But it is less used than MAC because non-repudiation is not needed in most cases, and because the computational cost is higher, like asymmetric ciphers compared to symmetric ciphers (even if optimization can be done, by signing only the hash of the messages instead of the whole messages).

1.1.6 Certificates

The asymmetric ciphers are weak against the Man-In-The-Middle (MITM) attack. In this attack, represented at Figure 1.4, the attacker intercepts the connection request of host A sent to host B and establishes the connection with host A himself (impersonating the host B). He generates a public and a private key and sends his public key to the host A. Then he establishes a connection with the host B. When the key exchange is finished, he can read (and even modify) all the data sent between host A and host B. He intercepts the packets, decrypts them with his private keys and then encrypts them again but with the public key of the other host.

Public key certificates (also known as digital certificates or identity certificates) ensure the identity of the public keys' owners. They make the attack above harmless since the attacker cannot impersonate the host B. The certificate is a digital document providing information about the key and its owner's identity (e.g., a society). The document is signed with the digital signature of the entity that vouches for the veracity of the document, called the `certificate authority` (CA). With such a certificate, a host can introduce himself to everyone who trusts

the CA and knows the CA's public key (in order to verify the signature). Indeed, if the signature is valid and if the signer is trustworthy, the public key very likely belongs to the owner specified by the certificate.

The CA can be a personal friend, college, acquaintance, etc. that is trustworthy (for example because he has met the key's owner himself). But this is not convenient for large network. There are thus some companies which propose to issue certificates to their clients[3].

1.1.7 Authenticated Encryption with Associated Data (AEAD)

Authenticated Encryption with Associated Data (AEAD, defined in [20]) protocols are black boxes designed by cryptographers who were fed up with security issues coming from the bad combinations of cryptographic protocols, made by the computer scientists. AEAD black boxes provide data confidentiality, data integrity and authentication by means of a single, easy to use programming interface. To use AEAD on data to be sent, only one function call is needed, with the following inputs:

- the symmetric key (like in symmetric ciphers),
- a nonce unique over all time, that enables, among others, to have non-determinism (i.e., an AEAD run twice on the same plaintext gives two different ciphertexts) and to avoid replay attacks,
- the plaintext that must be authenticated and encrypted,
- some additional data that only need to be authenticated (they are optional).

The output is a ciphertext, which is at least as long as the plaintext and which includes the authentication tag. The tag verification and the decryption are also done in a single step, with the key, the same nonce, the same additional data and the ciphertext as inputs.

Inside these black boxes, the cryptographic protocols can be different. Here are, for instance, two AEAD:

- The `AES-GCM` (see [21]) that works with the Galois Counter Mode (a mode of operation for symmetric key block ciphers) and with the Advanced Encryption Standard (a symmetric cipher).
- The `CHACHA20-POLY1305` (see [22]) that works with the chacha20 (a stream cipher) and the Poly1305 (a cryptographic message authentication code).

Both of these algorithms will be used in this document, while designing MPTCPsec (Chapter 4).

1.2 Transmission Control Protocol (TCP)

TCP, specified in [1], provides a reliable[4] connection between two hosts that want to exchange data. TCP can adapt itself to the network (unlike User Datagram Protocol [23]) in order, for example, to handle congestion (the congestion control is detailed in [24]). The establishment of a TCP connection is made via a three-way handshake. The data can then be sent and

[3] An example is Let's Encrypt, *a free, automated, and open certificate authority brought to you by the non-profit Internet Security Research Group (ISRG)*, as said in `https://letsencrypt.org/`.

[4] Either data sent by a host will be received (in order) by the other host or both hosts will detect the failure (in the absence of active adversary).

received reliably by both hosts, by means of TCP segments composed of a header and a data portion. These segments are reordered by TCP upon reception and a mechanism also enables detecting transmission errors. The connection can be released abruptly or gracefully. This section focuses on the TCP segment header (and gives an overview of TCP by the way), the connection establishment and the abrupt connection release.

1.2.1 TCP segment header and overview

Each TCP segment contains a header, with some fixed fields and some space for additional options (Figure 1.5). The size (in 32-bits words) of this header is equal to value of the `Data offset` field, which is 4 bits long. So, the option space is limited to 40 bytes (i.e., 15 32-bits words minus 5 for the fixed fields leaves us 10 32-bits words for the options). This point is really important to better understand implementation choices of many protocols: this limit is really restrictive ! All protocols (running above TCP) try to spare this space as much as possible.

TCP connections are identified by IP addresses and port numbers of both hosts (i.e., the source and the destination). And in a TCP connection, each byte of data is covered by a sequence number, starting from a random value (the Initial Sequence Number) at the beginning of the connection, and increased by one for each byte of data covered. The sequence number of the first data byte of the segment is written in the `Sequence Number` field. It enables hosts to communicate together on which data have been received and which have not. Indeed, hosts can advertise, with the `Acknowledgment Number` field of the header, the sequence number until which all data have been received (i.e., data mapped with a sequence number before this one are acknowledged too). This is the cumulative acknowledgment scheme. There exists an option, the selective acknowledgment (SACK) option, defined in [25] and widely used, that allows to acknowledge data that are not in sequence.

A point interesting to keep in mind is the need to know (or guess) the `Source Port`, `Destination Port` and the current `Sequence Number` of a connection to be able to forge a valid TCP segment. It restricts attacks on MultiPath TCP (Chapter 1.3), as the attacker must be on the physical path used by the connection to know these fields.

It is also interesting to remember that the `Window Size` advertises the maximal number of unacknowledged bytes that the host is currently willing to receive. Therefore, this value directly impacts the data throughput (i.e., it is vulnerable to Denial Of Service attack, as explained later). In order to enable windows larger than 2^{16} bytes, the TCP window scale option can be exchanged during the handshake in order to advertise a scaling factor. [26] defines this option. It also defines the timestamps option, attached to each data segment, that allows the receiver to know in which order the segments have been sent (it can be useful to help adapting the network use by TCP).

If the `URG` flag is set, the `Urgent Pointer` field is an offset from the sequence number indicating the last urgent data byte. These data must be processed directly upon reception.

And finally, another widely used option of the TCP handshake is the MSS option, defined in [27]. It specifies the largest amount of data (in bytes) that the sender is willing to receive in a single segment. It is usually chosen to be smaller than the minimal MTU (Maximum Transmission Unit, also defined in [27]) of the path used by the connection, in order to avoid IP fragmentation as much as possible (a protocol that discovers the smallest MTU of a path is defined in [28]).

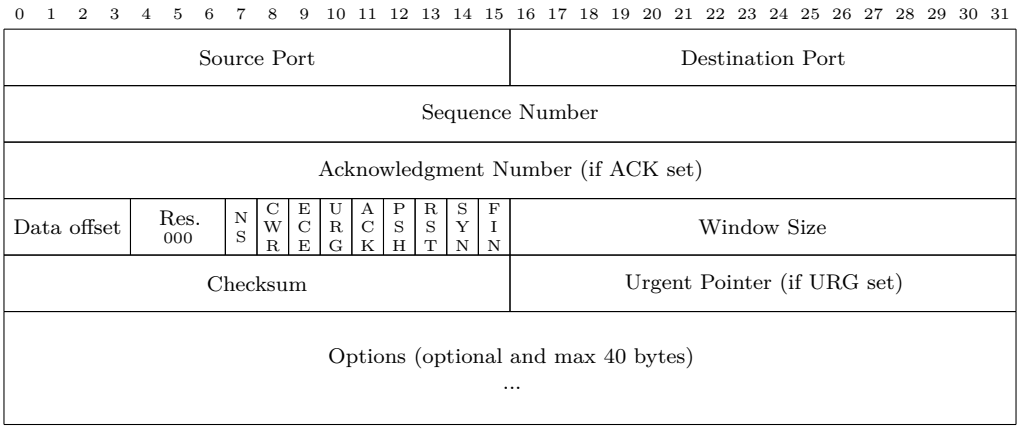
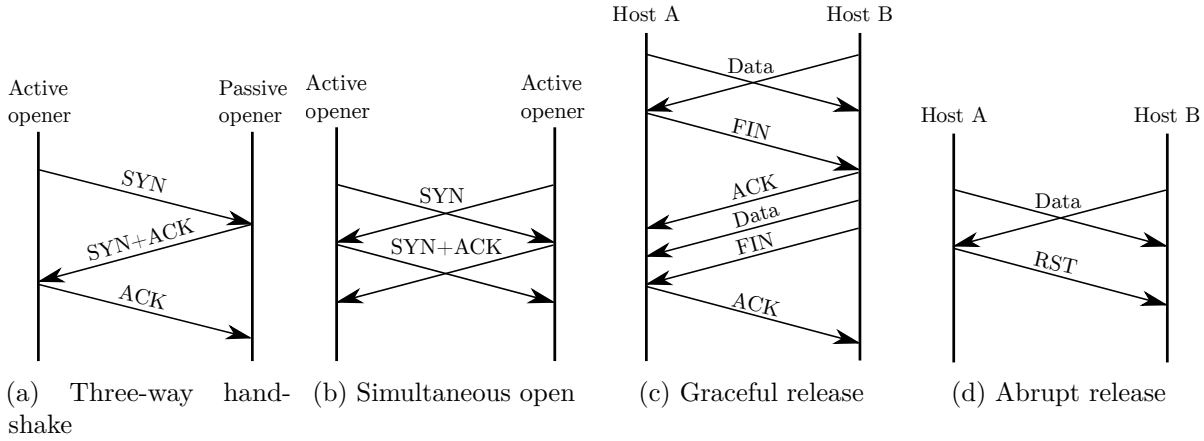


Figure 1.5: Format of the TCP segment header.



(a) Three-way handshake (b) Simultaneous open (c) Graceful release (d) Abrupt release

Figure 1.6: TCP connection establishments and releases

1.2.2 Connection establishment

The establishment of a TCP connection is made with a three-way handshake, as shown in Figure 1.6a :

- **SYN :** The host which wants to open the connection sends a TCP segment with the `SYN` flag set to 1 (Figure 1.5). This host is called the `active opener`.
- **SYN-ACK :** The host which receives the SYN segment responds with a TCP segment whose `SYN` and `ACK` flags are set to 1. This host is called the `passive opener`.
- **ACK :** The first host finishes the handshake by sending a segment with the `ACK` flag set to 1 (and `SYN` flag set to 0).

Initial Sequence Numbers (ISN) of both hosts are advertised with the `Sequence Number` fields of the SYN segments (i.e., SYN and SYN-ACK segments).

It can happen that both hosts are active openers (Figure 1.6b). This case is called `simultaneous open` and only happens under certain uncommon circumstances. This connection establishment is more like two simultaneous two-way handshakes. Each host sends a SYN segment (at the same time), and upon reception of the SYN segment of the other host, sends a SYN-ACK segment. No more segments are needed.

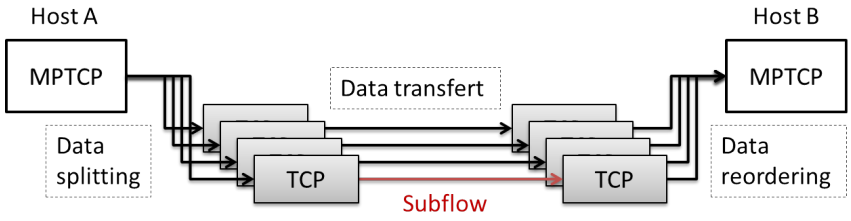


Figure 1.7: Tranfert of a bunch of data from Host A to Host B with MPTCP.

1.2.3 Connection release

The connection can be gracefully released in a four-way handshake. It is depicted in Figure 1.6c. When a host knows that it has no more data to send, it sends a segment with the `FIN` flag set. The other host can still send data, but must acknowledge this segment, while the sender of the FIN segment cannot send data anymore (but can still receive those of the other hosts). When the other host has finished too, the same is done in the reverse direction. The connection can be released only when both `FIN` segments have been acknowledged.

But it is the abrupt connection release, faster than the graceful one, that matters for this document. It is shown in Figure 1.6d. Hosts can send a TCP segment with the `RST` flag set (Figure 1.5) to immediately close the connection. There is no acknowledgment. This obviously leads to Denial of Service attacks (DoS attack, explained in details in [29]): the attacker can close a connection by sending a forged RST segment.

1.3 MultiPath TCP

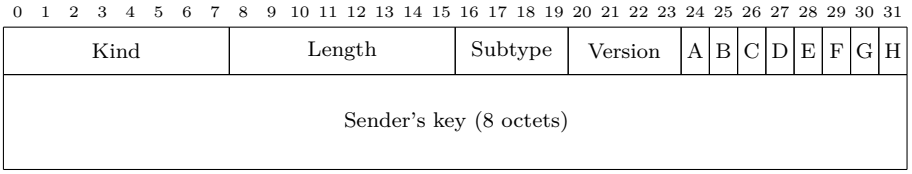
MultiPath TCP (MPTCP, specified in [13]) connections use multiple TCP connections (called `subflows` and depicted in Figure 1.7) in order to improve the throughput (and network use), the reliability and even the security, compared to a regular TCP connection. Indeed, while TCP is able to use only one physical path in the network and only one network interface of the host, MPTCP can use all available interfaces (e.g., Ethernet and WiFi) and many physical paths in the network. The throughput is often better (it is almost the sum of the throughput of each interface). The reliability is improved because if one TCP connection fails (e.g., due to a network failure), the connection can quickly establish new subflows. The security is also improved since the data flow is spread among multiple subflows. Therefore, the attacker needs to be on all these subflows to get all the data exchanged. The Denial of Service (DoS) attacks should be harder too[5], but there is no proof of that.

As depicted in Figure 1.7, MPTCP splits the data and distributes the resulting chunks among the TCP subflows. Then, all these chunks are received at the other side, and they are re-ordered. In order to make it possible and to handle the creation and removal of subflows, TCP segment headers contain some MPTCP options. These options are reviewed in this section, as well as the options used for the establishment and closure of the MPTCP connection. All along these explanations, the main parts of the MPTCP protocol are described.

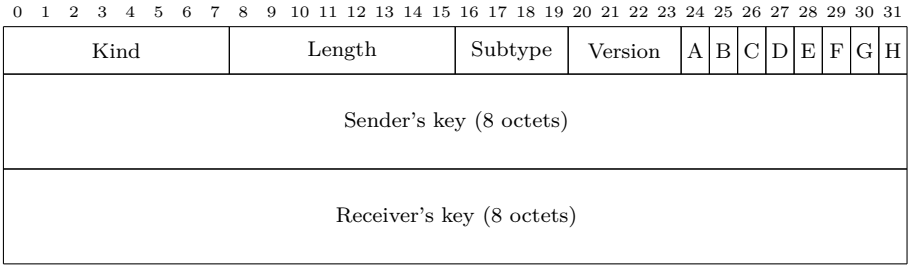
1.3.1 `MP_CAPABLE`

The `Multipath Capable` (`MP_CAPABLE`) option is used during the TCP three-way handshake, as shown in the top of the time sequence diagram in Figure 1.9, in order to negotiate the use of MPTCP and to exchange 64-bit keys. These keys are sent in clear only during the first handshake

[5]The solution proposed by this document, MPTCPsec, tries to make them even harder.



(a) `MP_CAPABLE` option in SYN and SYN+ACK segments.



(b) `MP_CAPABLE` option in ACK segment.

Figure 1.8: Format of the `MP_CAPABLE` option during the initial three-way handshake.

(and when the connection is abruptly closed with the `MP_FASTCLOSE` option, Section 1.3.5). MPTCP uses them to authenticate the addition of new subflows by means of 32-bits `tokens`, which are their cryptographic hashes. The tokens uniquely identify the MPTCP connection (for a given host). The `Initial Data Sequence Number` (IDSN) of a host, which corresponds to the Data Sequence Number (DSN, Section 1.3.4) of the first byte of data, is also derived from its key.

In the SYN and SYN-ACK segments, , each host sends its own key and a bunch of flags (as shown in Figure 1.8) :

- **`A` flag** : set to 1 if the use of the `DSS` option's `Checksum` field, that enables to detect transmission errors, is required.
- **`B` flag** : extensibility flag that is currently unused and must be set to 0.
- **`C` through `H`** : each of them identifies an HMAC algorithm[6] (Section 1.1.4). The sender sets to one the flags corresponding to the algorithms it supports and the receiver chooses one (and only one) of them. The chosen algorithm will be used to derive IDSNs and tokens from keys, and to authenticate the MP_JOIN options (Section 1.3.2).

The last ACK summarizes the exchange (e.g., both keys), confirms the negotiated values (e.g., the use or not of the checksum and the cryptographic algorithm chosen) and ensures that the MPTCP option is not removed by a middlebox. This summary enables to have a stateless server.

1.3.2 `MP_JOIN`

The `Join Connection` (`MP_JOIN`) option enables hosts to establish new subflows for their MPTCP connection. This option is added to regular TCP handshake segments and needs one extra acknowledgement (it is a four-way handshake). An example is given in Figure 1.9. The format of the `MP_JOIN` option is different for each segment (Figure 1.10).

In the first SYN segment, the sender identifies to which MPTCP connection the segment belongs with the `Receiver's token` (Section 1.3.1). It also sends the `Address ID` (unique identifier of an IP address, more details in Section 1.3.3) of the address from which it sends this segment (as the sending IP can be changed by NATs). Among the four flags, only one is currently assigned, the `Backup` (`B`) flag that indicates if this subflow should be used as a backup link or not (more details in Section 1.3.6). Finally, the sender must generate a random number.

[6]Currently only the `H` bit is assigned and identifies the HMAC-SHA1 algorithm [30].

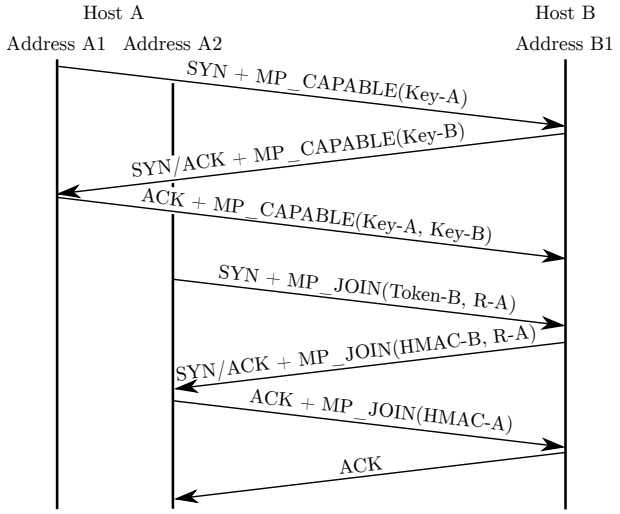


Figure 1.9: MPTCP three-way handshake (between addresses A1 and B1) and subflow establishment (between addresses A2 and B2). `R-A` (resp. `R-B`) is the random number of Host A (resp. Host B).

The receiver responds with a `MP_JOIN` option containing its preferences (the flags) and again the `Address ID` from which the segment is sent. The `Sender's truncated HMAC` (the leftmost 64 bits of a Hash-based Message Authentication Code, Section 1.1.4) is computed with the keys exchanged and the HMAC function negotiated during the initial handshake to prove the identity of the sender, and with the received random number in order to prove that this segment isn't a replay attack. Then it adds its own random number in order to enable the other host to do the same, in the third segment (the last ACK of the TCP three-way handshake), with a full HMAC (160 bits).

Since this third segment is an identity proof, it is a really important message. Therefore, it must be acknowledged before starting to send data. This leads to the four-way handshake.

1.3.3 `ADD_ADDR` & `REMOVE_ADDR`

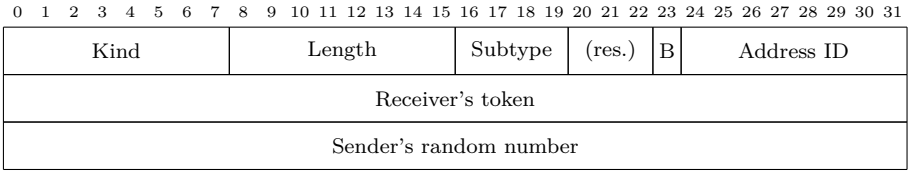
The `Add Address` (`ADD_ADDR`) and `Remove Address` (`REMOVE_ADDR`) options allow hosts to advertise and remove addresses on which subflows can be established. Advertising an address means that the sender is ready to accept a connection request for a new subflow on this address. These options are represented on Figure 1.11.

The first one can add addresses only one by one, via the `Address` field (whose length is dependent on the IP version, given by the `IPVer` field). An `Address ID` is defined in order to uniquely identify this address, within the scope of the connection. It enables to be compliant with the NATs (Network Address Translation, defined in [31]). It is also useful for the `MP_JOIN` and `REMOVE_ADDR` options as it is shorter than the IP address. A `Port` can be specified if wanted (but it is usually the same as the one of the initial subflows).

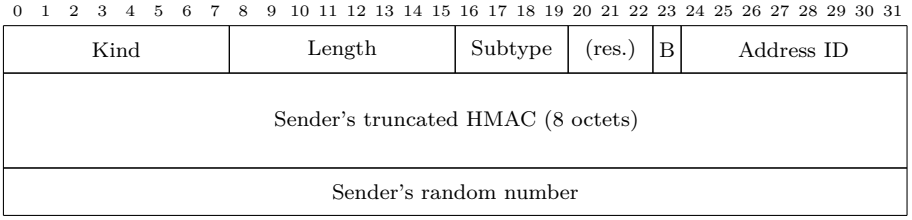
The `REMOVE_ADDR` can remove one or more addresses at once, identified by the 8-bits `Address ID`'s. When this option is received, the host must check if the address is not used. If not, it can remove the address from the available ones, otherwise it must check if this address is truly unreachable before closing the connection. In other words, this option is more like a warning in order to avoid long timeout before closing a dead connection.

1.3.4 `DSS`

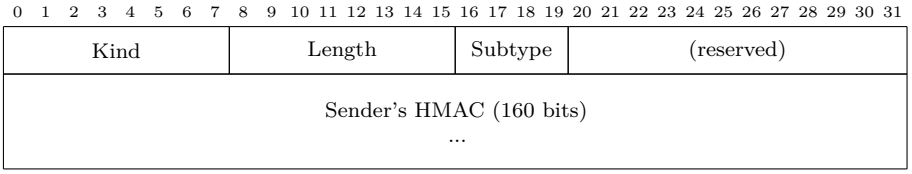
The `Data Sequence Signal` (DSS) option, whose format is given at Figure 1.12, is all about the data. As in TCP, each byte of data is covered by a sequence number in the MPTCP level,



(a) `MP_JOIN` option in SYN segment.

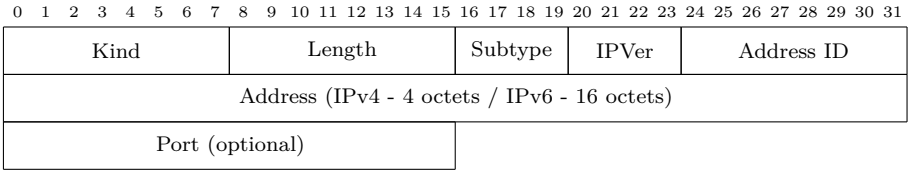


(b) `MP_JOIN` option in SYN+ACK segment.

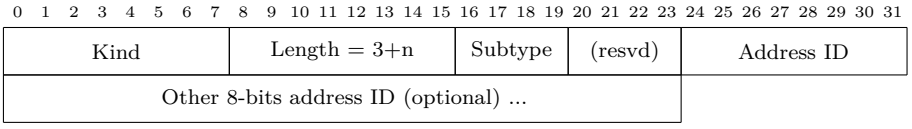


(c) `MP_JOIN` option in ACK segment.

Figure 1.10: Format of the `MP_JOIN` option during the four-way handshake establishing a new subflow.



(a) `ADD_ADDR` option.



(b) `REMOVE_ADDR` option, where n is the number of Address IDs advertised.

Figure 1.11: Format of the `ADD_ADDR` and `REMOVE_ADDR` options.

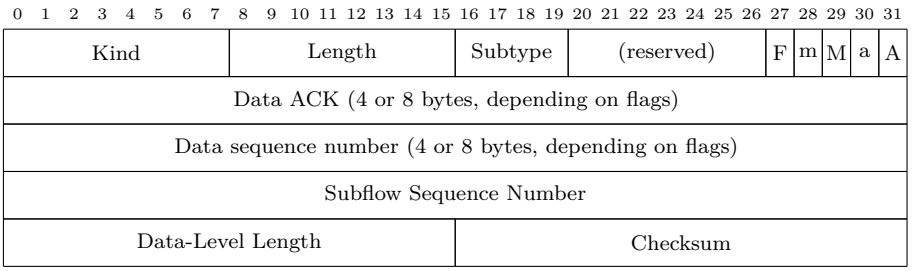


Figure 1.12: Format of the `DSS` option.

the `Data Sequence Number` (DSN). This number starts from a random value (computed from the keys, Section 1.3.1) and is incremented one by one for each byte of data sent. These data are acknowledged in the same way than in TCP (cumulative acknowledgment) with the `Data ACK` field. Once these data are mapped to DSNs, they can be sent on different subflows, which use TCP and cover them again with their own sequence numbers. The latter are specified in the DSS option, with the `Subflow Sequence Number` (SSN).

All data is covered by a DSS option, but a DSS option can cover more than one segment. The total length of data covered by this DSS must therefore be specified. This is the purpose of the `Data-Level Length`. Of course, all the data covered by the same DSS must be sent on the same subflow, in sequence w.r.t. TCP sequence numbers.

If a subflow is closed, it encounters some congestion issues or even some network failures, the data sent on this subflow can (or sometimes must) be sent on other subflows. It is called `re-injection`. The DSS option must be changed (at least the SSN is modified). The word `re-transmission` is also used for segments that are sent again, but this time on the same subflow (this is simply the TCP re-transmission, in case of packet losses for example).

The first 4 bytes of the DSS option are the followings flags and fields:

- **`M` (resp. `A`) flag** : set to one if the segment contains data (resp. acknowledges data), and thus, if the Data Sequence Number (DSN), Subflow Sequence Number (SSN), Data-Level Length and Checksum fields (resp. Data ACK field) are present.
- **`m` (resp. `a`) flag** : set to one if the DSN (resp. Data ACK) is 8-bytes long (it is 4-bytes long otherwise).
- **`DATA_FIN` (`F`) flag** : set to one if this is the end of the data stream (i.e., the sender has no more data to send). This is the connection-level equivalent of the TCP's `FIN` flag, i.e., the graceful connection release (Section 1.2.3).
- **`(reserved)` bits** : reserved for future use.

The last field is the `Checksum`, that can be used to ensure the data integrity (in the absence of active adversary, i.e., in order to detect transmission errors). Its use can be negotiated during the MPTCP handshake, with the `A` flag (Section 1.3.1)

1.3.5 `MP_FASTCLOSE`

The `Fast Close` (`MP_FASTCLOSE`) option is to MPTCP what `RST` segment is to TCP, i.e., the abrupt connection release (Section 1.2.3). The host which wants to close the MPTCP connection must send an ACK segment with this option on one subflow and RST segments on all other subflows. The last subflow is closed by the other host, with a RST segment that somehow acknowledges the reception of the `MP_FASTCLOSE` option. At that time, the whole MPTCP

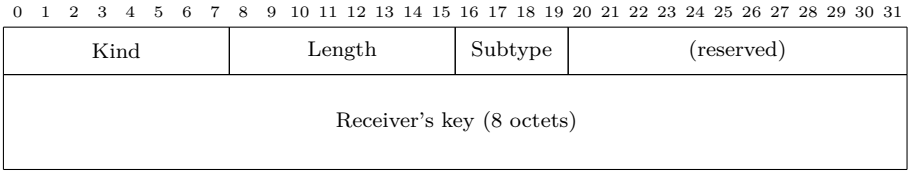


Figure 1.13: Format of the `MP_FASTCLOSE` option.

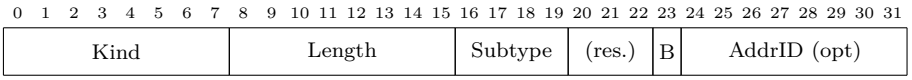


Figure 1.14: Format of the `MP_PRIO` option.

connection can be shutdown. The other host can close the whole connection as soon as it has sent the RST segment on the last subflow. If no RST segment is received after one re-transmission timeout, the `MP_FASTCLOSE` should be retransmitted.

The `MP_FASTCLOSE` option (Figure 1.13) contains the key of the remote host in order to authenticate it (i.e., to prove its validity, to prove that this isn't an attacker who has sent this segment). The key is sent in clear as we don't need it anymore anyway.

1.3.6 `MP_PRIO`

The `Change Subflow Priority` (`MP_PRIO`) option enables to change the `Backup` flag (`B` flag in Figure 1.14) associated to the subflow referenced by the `AddrID` field (or the subflow on which the packet is received if there is no `AddrID` field defined). This flag is initialized with the `B` flag of the MP_JOIN option (Section 1.3.2).

If this flag is set for a subflow, this means that this subflow should be used as a backup link by the host. It should be used only if no other subflows with the `B` bit set to 0 are usable (but some local policies or technical difficulties may override this flag).

1.3.7 `MP_FAIL`

The `Fallback` option (`MP_FAIL`) is a little more subtle and complex, used to solve corner cases after the establishment of the MPTCP connection (i.e., after a successful three-way handshake). These corner cases appear when an incorrect checksum is received or when the path used by a subflow contains middleboxes that remove MPTCP options from non-SYN segments (and let those of SYN segment, during the three-way handshake). In these cases, MPTCP tries to fallback to the previous, safe operation, either by falling back to regular TCP or by removing a problematic subflow.

If there is more than one subflow, `MP_FAIL` tries to synchronize the DSN of both hosts to the last valid one (that it advertises, as shown in Figure 1.15). The hosts close the failing subflows. But if there is only one subflow, hosts cannot close it. `MP_FAIL` therefore leads to a fallback to a regular TCP connection (in fact, this is still an MPTCP connection, but it is strictly restricted to regular TCP connection behavior).

An example of a situation requiring MP_FAIL option is show in Figure 1.16. In the first subflow (Figure 1.16a), a middlebox adds data in the flow. This change is noticed by the other host thanks to the DSS checksum of the DSS option (note that a MPTCP-aware middlebox can change the DSS checksum too and the change can't be noticed by the receiver). The MP_FAIL option will be used in order to advertise the other host what was the last valid DSN received. Finally, the transfer is made on the other subflow.

What you should keep in mind for the rest of this document is that this option is used when the payload is changed or when MPTCP options are removed after the first handshake.

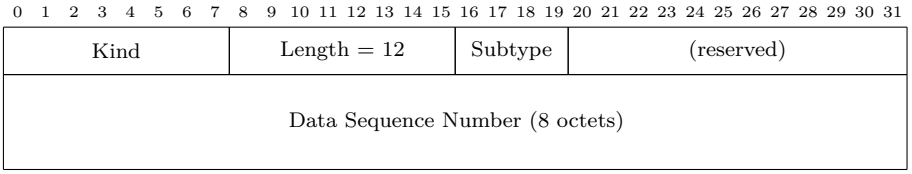
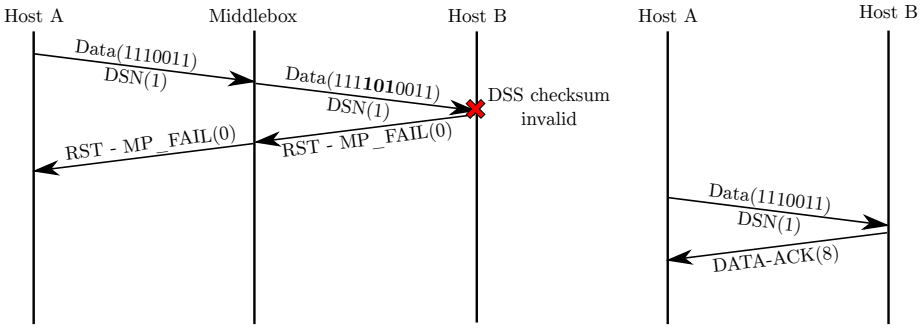


Figure 1.15: Format of the `MP_FAIL` option.



(a) Subflow with a middlebox (b) Other subflow

Figure 1.16: Example of usage of MP_FAIL option

1.4 Conclusion

Through this chapter, we have covered the bases needed to understand the main part of this document. Indeed, our protocol is based on MPTCP, and uses the cryptography to secure the connection. And MPTCP itself is based on TCP since each subflow is a TCP connection. A good understanding of TCP, MPTCP and the cryptography was thus essential to understand the work presented by this document. We are now ready to look at what have been already done to secure TCP, or in other words, on what we have based our work. The second chapter of this part details the protocols that have been developed over TCP to provide security. Our solution is mainly the adaptation of these TCP protocols to include them in MPTCP.

Chapter 2

TCP : Encryption & authentication

TCP does not include encryption and authentication. Many protocols have been proposed to provide these features, such as the popular Transport Layer Security (TLS) Protocol (Version 1.3 in [32]) and tcpcrypt [7]. But despite these efforts, most of TCP traffic on the Internet remains unencrypted and unauthenticated. This is due to the fact that integrating TLS requires applications to be modified, while tcpcrypt alone does not provide server authentication (weakness against man-in-the-middle attacks). Moreover, there is not a common way to handle any of these protocols since each one is designed in its own way, without any convention. This has motivated the IETF to provide a generalized and backward compatible mechanism for incrementally deploying encryption. TCP-ENO was born.

This chapter gives an overview of TCP-ENO (the details are the same as those of MPTCPesn, described in Chapter 3). Then, it explains in details tcpcrypt. The last section introduces the main parts of TLS. It is not explained in details because it acts like a black box: it is not truly integrated in and merged with TCP. And also simply because this document focuses more on the tcpcrypt part of MPTCPsec (Chapter 4).

2.1 TCP-ENO

TCP-ENO, defined in [10], is not an encryption method. It is an efficient way of communicating between two hosts that enables them to negotiate an encryption scheme[1]. As its full name indicates, TCP Encryption Negotiation Option is a TCP option. TCP-ENO is designed to be modular and easy to update. The supported encryption schemes can vary widely in terms of wire format, use of TCP option space, and integration with the TCP header and segmentation. Adding new encryption schemes can also be done quite easily. TCP-ENO provides therefore a really good generalization and eases the use of encryption.

This section provides an overview of TCP-ENO. The details are kept for the MPTCPesn part (one of the main subjects of this document, with MPTCPsec) which is the adaptation of TCP-ENO to MPTCP, and is therefore the same in many aspects.

2.1.1 TCP-ENO handshake : general case

TCP-ENO is a TCP option used during the first three-ways handshake of the connection (Section 1.2.2), i.e., the connection establishment (and example is shown in Figure 2.1).

In the first SYN segment of the TCP handshake, the ENO option contains a list of sub-options (Figure 2.2). They are by default one byte long, except the last one that can be of variable length (i.e., with additional data, as explained later in Section 2.1.5). One and only one sub-option of

[1]The encryption scheme defines the protocol (e.g., tcpcrypt), the cryptographic functions used (e.g., the key-agreement protocol) and can also defined, for example, if this is a first connection or a connection resumed.

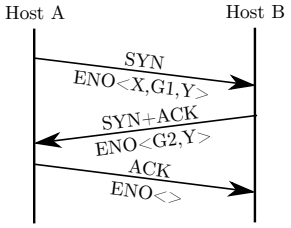


Figure 2.1: Here is an example of a regular handshake (not simultaneous open) where A proposes X and Y as encryption and authentication schemes, with general sub-option G1, and B chooses Y (and adds its general sub-option G2)

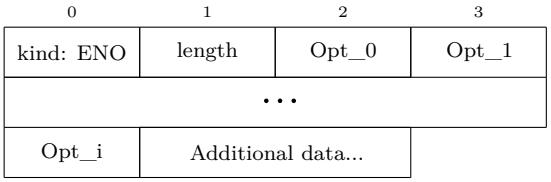


Figure 2.2: Format of a TCP-ENO option.

the ENO option defines general conditions independent from the negotiated scheme (Section 2.1.2) while others define encryption and authentication schemes proposed by the host. We want to draw your attention to the fact that each byte of the TCP header is precious (Section 1.2.1), and therefore, ENO can negotiate only 8-bits identifiers (e.g., there are 5 identifiers reserved for tcpcrypt).

The SYN-ACK segment of host B accepts one proposed scheme. If there is more than one scheme identifier, only the last valid one is kept (the one which can contain additional data). The TCP-ENO option also contains the general sub-option.

Both ACK segments (SYN-ACK and ACK segments except for simultaneous open 1.2.2) confirm that the ENO option is not removed by middle-boxes (in both directions). The last ACK packet does not contain any sub-option and is therefore only 2 bytes long.

2.1.2 General Sub-option

The general sub-option of the TCP-ENO option is used to negotiate general conditions independent from the negotiated scheme. It is depicted in Figure 2.3.

Two bits are reserved to specify if the sending host application is aware of the encryption and authentication and if the receiver host application must also be aware of it (the `Application-aware bits`, noted `aa`).

One bit is used by the role determination (the `Role-override bit`, noted `b`). It is useful only in case of simultaneous open. This is explained in Section 2.1.3.

Another bit is reserved for future use (the `Zero bit`, noted `z`) and the four remaining are always set to zero (to differentiate the general sub-option from the encryption scheme identifiers).

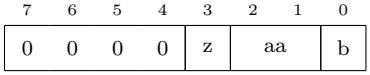


Figure 2.3: Format of the general sub-option.

2.1.3 Roles

During the TCP three-way handshake, TCP-ENO defines abstract roles for each host to distinguish them, either A or B. It can be useful in case of asymmetric algorithms. Following a normal three-way handshake with no special configuration, the host A is the active opener (the one which sends first the SYN packet) and the host B is the passive opener. We can also designate them as the client (host A) and the server (host B).

TCP-ENO has an additional mechanism to break the symmetry of simultaneous open (i.e., both hosts are active openers). It reserves a bit (the `Role-override bit`) in the general sub-option (Section 2.1.2) for this purpose. If this bit is set to 1 (resp. 0), it means that the sender of the segment wants to play the host B role (resp. host A role). If both hosts set the bit to the same value, TCP-ENO fails and reverts to unencrypted TCP.

2.1.4 Negotiation transcript

To prevent attacks on the handshake itself (e.g., an attacker that removes all schemes of the ENO option except the weakest one), TCP-ENO defines a transcript based on the ENO options. This transcript is called `eno-transcript` in the rest of this document. All encryption methods must use this transcript so that if it is not the same for both hosts, the encryption method will fail with high probability.

2.1.5 Variable-length sub-options

Variable-length sub-options are used when additional data is necessary. The main example is the connection/session resumption, whose one main goal is to avoid another handshake after the TCP handshake (e.g., for keys exchange). Additional data is thus necessary, e.g., to enable hosts to verify that the sender is still the same (and is not an attacker that tries to resume another connection).

The last sub-option of the TCP-ENO option can have additional data, whose length is deduced from the TCP-ENO option length. But if a host wants to negotiate more than one sub-option with additional data, it is necessary to specify the length of these data. A specific sub-option, the `marker byte`, was designed to this end. It is placed directly before the sub-option with additional data and advertises its length. TCP-ENO even defines a `marker word` to advertise additional data longer than 32 bytes. But it will only be useful when the TCP header size restriction will be solved (this issue is discussed in [33], and a solution is proposed).

2.1.6 Session ID

Each encryption and authentication scheme must define a session ID whose value uniquely identifies the TCP connection. This session ID must fulfill a bunch of requirements such as being at least 33-bytes long.

2.2 Tcpcrypt

Tcpcrypt is a cryptographic protocol that protects TCP payload data. It can be negotiated by means of TCP Encryption Negotiation Option (TCP-ENO). It is defined in [12] (and this section is inspired by this document). The main goal of tcpcrypt (besides security) is to be application independent. It can be implemented directly inside TCP stack. It is still possible for an application to communicate with tcpcrypt, but it is not necessary. As counter part, tcpcrypt does not provide a mechanism to exchange certificates (Section 1.1.6, but it can help the application to do so).

2.2.1 Tcpcrypt handshake

Due to TCP option size restriction (Section 1.2.1), both hosts cannot exchange a shared secret[2] during the ENO handshake. They can only agree on a specific sub-option. Tcpcrypt (currently) reserves 5 sub-options. Four of them specify key-agreement schemes[3] (Section 1.1.3) that will be used to build the shared secret. Tcpcrypt must still perform the exchange of the ephemeral public keys used by these key-agreement schemes. The corresponding ephemeral private keys are kept secret and never sent by hosts. One additional handshake is necessary to perform this key exchange. This cost can be avoided in case of connection resumption (i.e., if both hosts have recently established a previous tcpcrypt connection, Section 2.2.5), with the last sub-option reserved by tcpcrypt.

Tcpcrypt takes advantage of this extra-handshake to enable hosts to negotiate the symmetric cipher (Section 1.1.2), or more precisely the AEAD algorithm[4] (Section 1.1.7), that will be used to encrypt and authenticate the data. Host A sends a list of AEAD algorithms to host B, which chooses one of them and sends it back to host A. This handshake is composed of two messages, one from host A to host B (roles defined by TCP-ENO, Section 2.1.3), and one in the reverse direction:

$A \rightarrow B$: init1 = { `INIT1_MAGIC`, sym-cipher-list, N_A, Pk_A }
$B \rightarrow A$: init2 = { `INIT2_MAGIC`, sym-cipher, N_B, Pk_B }

with:

- sym-cipher-list: a list of symmetric ciphers (AEAD algorithms) acceptable to host A.
- sym-cipher: the symmetric cipher selected by B from the sym-cipher-list sent by A.
- N_A, N_B: nonces chosen at random by A and B, respectively.
- Pk_A, Pk_B: ephemeral public keys for A and B, respectively, required for the key-agreement scheme.
- `INIT1_MAGIC`, `INIT2_MAGIC`: constants defined by tcpcrypt.

Note that this exchange is made in the data portion of the TCP segments, once the TCP handshake is finished.

2.2.2 Keys and Session ID derivation

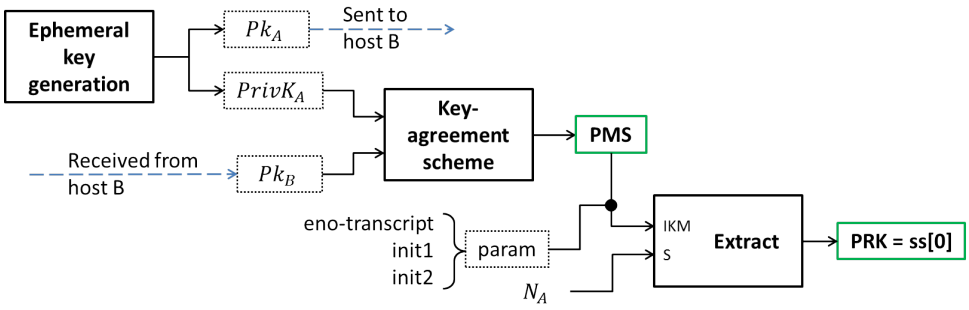
With the ephemeral public keys exchanged during the handshake, both hosts can compute the shared secret, called the pre-master secret (PMS), with the negotiated key-agreement scheme. This PMS is the result of the key-agreement algorithm whose inputs are the local host's ephemeral private key (not transmitted) and the remote host's ephemeral public key. This, as well as the entire key derivation process, is represented in Figure 2.4.

Then hosts need an extract function to generate a pseudo-random key (PRK) from the PMS. Indeed, the output of the key-agreement algorithm is not necessarily uniformly random or

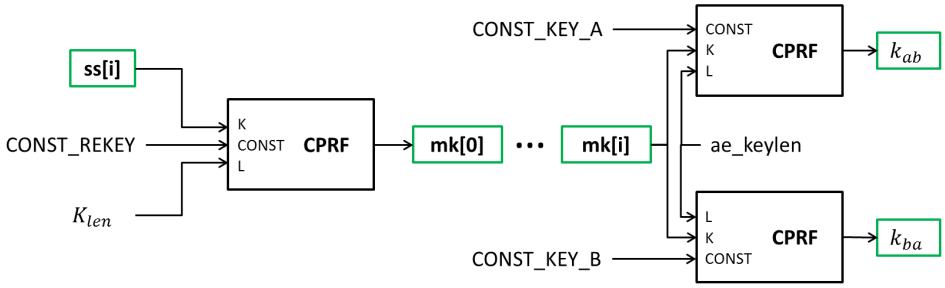
[2]Cryptographic protocols need a shared secret from which they can extract the secret keys used to encrypt and authenticate the data.

[3]The key-agreement schemes that can be currently negotiated are ECDHE-P256, ECDHE-P521, ECDHE-Curve25519 and ECDHE-Curve448 (defined in [34]).

[4]The AEAD that can be currently negotiated are AEAD_AES_128_GCM (16-bytes-long keys), AEAD_AES_256_GCM (32-bytes-long keys), specified in [21], and AEAD_CHACHA20_POLY1305 (32-bytes-long keys), specified in [22]. A one-byte identifier is defined for each of them, in order to negotiate them easily.



(a) Initial Pseudo Random Key derivation (for the host A).



(b) Encryption and authentication keys derivation.

Figure 2.4: Figure (a) depicts the Pseudo-Random Key (PRK) derivation on host A side, that needs to be done only at the initialization of the first connection (it is avoided with connection resumption). The derivation on host B side is easily obtained by replacing all "A" (resp. "B") by "B" (resp. "A") except for the N_A that is sent by host A. The dot means that PMS and param are concatenated.
Figure (b) presents the derivation of the initial master key (mk) from the session secret, and the derivation of the final keys from a given master secret. The ss[i] (resp. mk[i]) can be obtained with a CPRF on the ss[i-1] (resp. mk[i-1]).

```
HKDF-Extract(salt, IKM) -> PRK
    PRK = HMAC-Hash(salt, IKM)

HKDF-Expand(PRK, CONST, L) -> OKM
   T(0) = empty string (zero length)
   T(1) = HMAC-Hash(PRK, T(0) | CONST | 0x01)
   T(2) = HMAC-Hash(PRK, T(1) | CONST | 0x02)
   T(3) = HMAC-Hash(PRK, T(2) | CONST | 0x03)
   ...

   OKM  = first L octets of T(1) | T(2) | T(3) | ...
```

Figure 2.5: The symbol | denotes concatenation, and the counter concatenated with CONST is a single octet. The image comes from [12].

pseudorandom. It can be not uniformly distributed. The extract function outputs a fixed-length pseudorandom key that is cryptographically strong, by concentrating the entropy of the input. After that, a collision-resistant pseudo-random function (CPRF) is needed in order to generate multiple cryptographic keys (one in each direction, $A \rightarrow B$ and $B \rightarrow A$) from this pseudo-random key. Tcpcrypt uses by default the Extract and Expand functions of HKDF[5] (HMAC-based Key Derivation Function, defined in [35]). They are described in Figure 2.5, in which the pseudo random function `HMAC-Hash(key, value)` is a negotiated HMAC function.

Let us use the following notations:

- **Extract(S, IKM),** the output of the extract function with salt S (optional) and initial keying material IKM,
- **CPRF(K, CONST, L),** the output of L bytes of the pseudo-random function identified by key K on CONST.

Tcpcrypt uses the eno-transcript (Section 2.1.4) during the generation of the PRK so that the connection will be closed if the transcript is not the same at both connection's ends (e.g., if an attacker has tried to modify the ENO option). Indeed, the keys will be different on both sides and the hosts will not be able to communicate. For the same reason, tcpcrypt also includes the init messages. The PRK is computed as follows :

param := { eno-transcript, init1, init2 }
PRK := Extract (N_A, { param, PMS })

This PRK is the initial `session secret`, ss[0]. From a session secret, ss[i], tcpcrypt enables to compute the following one, ss[i+1]. The purpose is to avoid, in case of connection resumption (Section 2.2.5), the most costly part of the key exchange protocol: the public key cipher.

From this session secret, hosts can compute the Session ID, SID[0], and an initial master key, mk[0], as follows :

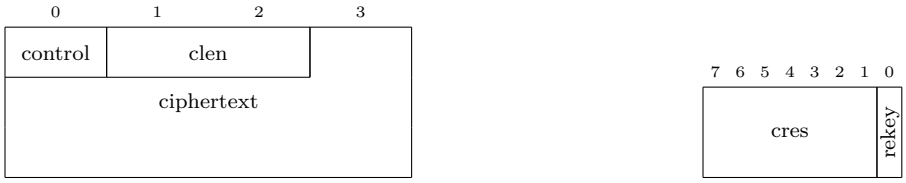
SID[i] := CPRF (ss[i], `CONST_SESSID`, K_{len})

mk[0] := CPRF (ss[i], `CONST_REKEY`, K_{len})

with:

- ss[i]: the session secret of the current session i.
- `CONST_REKEY` and `CONST_SESSID`: constants defined by tcpcrypt.

[5]Tcpcrypt uses HKDF-Expand-SHA256 as the CPRF, and nonces and session keys of 32 bytes (`N_A_LEN`, `N_B_LEN` and `K_LEN` in [12]).



(a) Application frame (b) Control byte

Figure 2.6: Format of an application frame (data portion of the TCP segment).

- K_{len}: depends on the tcpcrypt sub-option negotiated during the ENO handshake[6].

In case of rekeying (Section 2.2.4), the new master key mk[i+1] is derived from the previous one, mk[i].

Finally, the keys can be derived from the master key. We need one key for each direction, $A \rightarrow B$ and $B \rightarrow A$:

k_{ab} := CPRF(mk[i], `CONST_KEY_A`, ae_keylen)
k_{ba} := CPRF(mk[i], `CONST_KEY_B`, ae_keylen)

with:

- mk[i]: the current master key.
- `CONST_KEY_A` and `CONST_KEY_B`: constants defined by tcpcrypt.
- ae_keylen: depends on the authenticated-encryption algorithm selected during the tcpcrypt handshake.

The key k_{ab} (resp. k_{ba}) is used for data sent by host A (resp. B). In other words, host A (resp. B) encrypts data with this key and host B (resp. A) decrypts with it.

2.2.3 Data transfer

Hosts are now ready to exchange data, protected by encryption and authentication with the agreed symmetric keys. This protection is provided by the AEAD (Section 1.1.7) that is negotiated during the tcpcypt handshake. The application frame format used by tcpcrypt is depicted in Figure 2.6.

The application data stream is truncated into chunks that are encrypted and authenticated with the AEAD. The output of the AEAD is put in the `ciphertext` field of the application frame. The length of the ciphertext is given by the `clen` field. Most of the bits of the `control` byte are reserved for future uses (`cres` field on Figure 2.6). Only the `rekey` bit is used, to inform the other host that we have initiated a rekeying (Section 2.2.4). In order to protect these two fields from modifications, they are used as associated data when encrypting the data chunk. Finally, the AEAD needs a unique nonce for each chunk. Thus, each host keeps a specific structure (the `frame nonce`, which includes the offset in the data stream) used to build this nonce, and ensure its uniqueness.

2.2.4 Re-keying mechanism

When hosts want to negotiate new keys and wipe the current one from memory, they can start a re-keying mechanism. They do not need to perform a second handshake, only to inform

[6]currently tcpcrypt puts this value to 32 bytes in all cases

the other host by means of a single bit (the rekey bit of Figure 2.6). They only need to compute a new master key mk[i+1] from the current one, mk[i], as follows :

mk[i+1] := CPRF (mk[i], `CONST_REKEY`, K_{len})

with `CONST_REKEY` a constant defined by tcpcrypt and K_{len} depending on the tcpcrypt sub-option negotiated.

The keys can then be computed as usual (Section 2.2.2) with this new master key.

2.2.5 Connection Resumption

Tcpcrypt enables two hosts to resume, if it exists, their previous connection. They do not need to perform a new handshake, which is costly due to the public-key operations. To do so, both hosts must have computed the session secret ss[i+1] during the previous connection, and remember it:

ss[i+1] := CPRF (ss[i], `CONST_NEXTK`, K_{len})

with ss[i] the session secret of the previous connection, `CONST_NEXTK` a constant defined by tcpcrypt and K_{len} depending on the tcpcrypt sub-option negotiated.

From this, the host which wants to resume the connection must compute the new Session ID, SID[i+1]. Then, it can send the SYN packet with a TCP-ENO option containing the tcpcrypt sub-option reserved for the connection resumption, and with the first nine bytes of this new SID as additional data. The remote host, if it remembers the ss[i+1], computes the SID[i+1] and compares the first nine bytes. In case of success, both hosts compute the master key as usual and avoid the tcpcrypt handshake (the A and B roles are the same as in the initial connection, no matter which host has sent the SYN packet of the current connection). Otherwise the remote host responds with one another sub-option of the TCP-ENO option and behaves as if it was the first connection.

2.3 Transport Layer Security (TLS) Protocol

The TLS protocol, whose Version 1.3 is defined in [32], provides communication security over the Internet. The protocol enables client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. TLS can be negotiated by means of TCP Encryption Negotiation Option (TCP-ENO), but some adaptations are needed. They are explained in [11] and this section is based on this version.

The main goal of TLS is the security, and it provides a way to exchange certificates (Section 1.1.6) in order to ensure the identity of the other host. But this requires applications to be adapted and heavier handshakes.

This section provides a quick overview of TLS. The first point is the handshake. Then the data protection is explained, followed by the connection resumption. This section concludes with the keys management.

2.3.1 TLS handshake

The handshake of TLS (the version adapted for TCP-ENO) is a four-way handshake and is shown in Figure 2.7. The first message is the `ClientHello` message. It contains a random nonce, a list of ciphers (AEAD algorithms), a list of signatures/hash algorithm pairs and the willingness to accept a raw public key rather than an X.509 certificate (for the authentication of the server). The client also proposes a list of Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) groups and

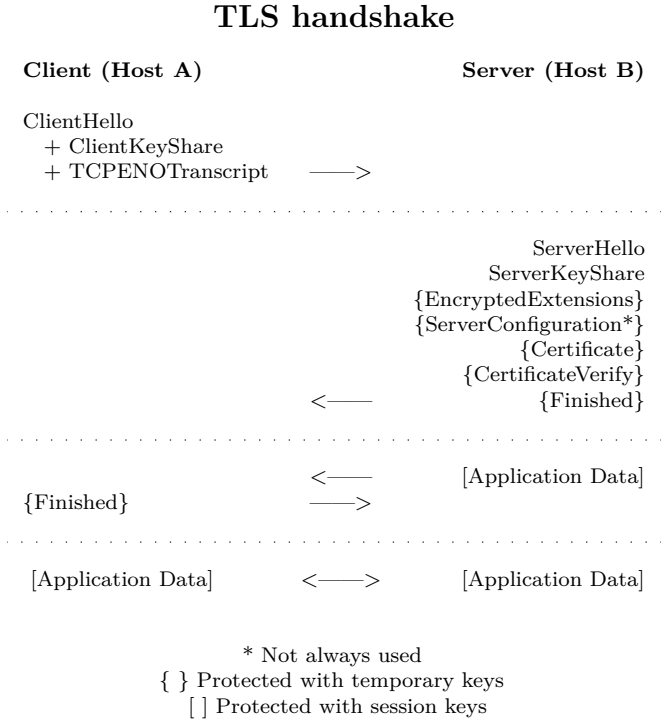


Figure 2.7: TLS (version for TCP-ENO) handshake [11]

the associated key(s) of some (or all) of them, `ClientKeyShare`. They will be used to protect the rest of the handshake with encryption (the `{ }` in Figure 2.7). Finally, the `TCP-ENO-Transcript` (Section 2.1.4) is sent too.

The server answers with `ServerHello` containing a random nonce, the selected cipher suite and ECDHE group. If there is not the key corresponding to the chosen ECDHE group in the `ClientKeyShare` (which is unlikely), the server must send a message, the `HelloRetryRequest`, asking the client to send a new `ClientHello` with this key. The `ServerKeyShare` corresponding to the chosen ECDHE group is then sent. All following handshake messages will be encrypted using keys derived from both `ClientKeyShare` and `ServerKeyShare`. These messages include :

- `EncryptedExtension`: (mainly) the response to the willingness to accept a raw public key rather than an X.509 certificate,
- `ServerConfiguration`: useful to resume this connection later (Section 2.3.3),
- `Certificate` and `CertificateVerify`: used to authenticate the server (with either a raw public key or a certificate, according to the `EncryptedExtension`),
- `Finished`: a MAC over the entire TLS handshake transcript up to this point.

At this point, the server can send application data, but the client must still send his `Finished` message before sending data himself.

2.3.2 Data protection

A layer is inserted between the application and TCP, the record layer. It splits the data stream of the application and puts each fragment in a `TLSPlaintext` structure. Then the record protection functions translate these TLSPlaintext structures into `TLSCiphertext` structures. An AEAD is used at this step, whose nonce is based on the data offset in the data stream (by means

of the 64-bit record sequence number). Both `TLSPlaintext` and `TLSCiphertext` structures contain the length of the plaintext/ciphertext, the TLS version and the `ContentType`. The latter specifies the type of the record (e.g., application data record, handshake record, alert record). Finally, the record layer sends these `TLSCiphertext` records to the TCP (or MPTCP) layer which will treat them as usual, like a regular data stream of an application.

At the other side, the record layer extracts the records from the data stream received by the TCP (or MPTCP) layer, and the deprotection functions reverse the process of the record protection functions (i.e., translate the `TLSCiphertext` records into `TLSPlaintext` records). The data are finally sent to the application.

2.3.3 Connection resumption

TLS enables two hosts that have already connected together to resume their connection, avoiding extra-handshakes and computation. This can be negotiated with the variable-length sub-option of the ENO option. The `ServerConfiguration` must be put in the extra-data of the sub-option. But hosts still need to do a whole handshake (a little bit different from the initial one), with `ClientHello`, `ServerHello` and `Finished` messages. To reduce the latency, TLS allows the client (resp. the server) to send data as soon as it has sent the `ClientHello` (resp. `ServerHello`), but the key used (the static secret) to protect them is less secure than usually (i.e., no forward secrecy). Moreover, the connection resumption can lead to replay attacks.

2.3.4 Keys management

The negotiation of the key agreement scheme is made in the TLS handshake, after the TCP-ENO handshake.

The key derivation is done by means of the HKDF key-derivation algorithm (as in tcpcrypt, Section 2.2.2). Two keys are derived: one for each direction (i.e., *client* $\rightarrow$ *server*, and *server* $\rightarrow$ *client*).

The version 1.3 of TLS has removed the rekeying mechanism from the version 1.2 [8] due to security issues. Therefore, if hosts want to renew the keys, they must close the connection and restart a new one (but they can use the connection resumption to make it faster, Section 2.3.3).

2.4 Conclusion

TLS and tcpcrypt provide encryption and authentication over TCP. But there was not a common way to handle these protocols since each one is designed in its own way, without any convention. A client that wanted to have a secure connection, either with tcpcrypt or TLS, had to try to establish a tcpcrypt connection with the server, and then a TLS connection if the server did not support the first one. Imagine what would become with several encryption schemes. This was really not efficient. Thanks to TCP-ENO, it is now easier to use encryption (and authentication) over TCP. Several encryption schemes can be easily negotiated. TLS and tcpcrypt have been adapted in order to be compliant with TCP-ENO.

This work made on TCP is a good base to start our work on MPTCP. We can learn from this experience (securing TCP) to avoid these mistakes. Moreover, we can base our protocol on these secure protocols to avoid making twice the same work.

Part II

Design

Chapter 3

MPTCPesn

The tcpinc IETF working group is in charge of TCP-ENO (Chapter 2.1), in order to provide an easy way to negotiate encryption schemes over TCP. This chapter extends this work to MPTCP. The protocol we have designed, MPTCP Encryption Scheme Negotiation (MPTCPesn), is the adaptation of TCP-ENO to MPTCP (and this chapter is thus based on the TCP-ENO draft [10]).

This chapter is divided in three parts. The first one concerns the MPTCP handshake when MPTCPesn is used. The second section details the sub-options used by MPTCPesn (the same as those of the TCP-ENO option). This chapter finishes with some requirements for the encryption schemes that can be negotiated with MPTCPesn.

3.1 MPTCPesn handshake

TCP-ENO is a standalone TCP option. MPTCP is negotiated itself by an option, the `MP_CAPABLE` option (Section 1.3.1). It seems better to use this option than the ENO option. Indeed, it is useless to negotiate encryption schemes that run on MPTCP without negotiating the use of MPTCP itself. This section first explains the modifications of the `MP_CAPABLE` option and shows its use in the MPTCP handshake. The fallback in case of failure, that can be further developed later, is explored just after that. Then it defines the MPTCPesn transcript of the handshake. Finally, it shows the weakness of MPTCPesn against active attackers.

3.1.1 `MP_CAPABLE` as the ENO option

The first step is to modify the `MP_CAPABLE` option by removing the `Sender's Key` and `Receiver's Key` (Figure 1.8). They are useless since the encryption scheme to be negotiated will perform a second handshake in order to establish a shared secret. This shared secret can also be used for the same purpose as MPTCP connection's keys, so that we do not need to exchange them. Thanks to this trick, some space is spared in the `MP_CAPABLE` option (at least 64 bits, which is really precious, as said in Section 1.2.1). Let us use this space to negotiate the encryption scheme and the general sub-option as in TCP-ENO (more details about these sub-options are given in Section 3.2). The modifications are depicted in Figure 3.1.

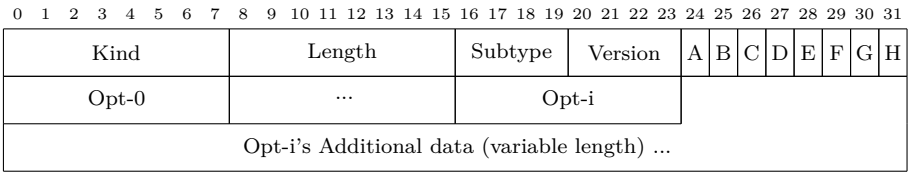


Figure 3.1: `MP_CAPABLE` option modified for MPTCPesn.

In order to differentiate this `MP_CAPABLE` option from the old one, we increment the version (in the `Version` Field).

This option is used exactly as the TCP-ENO option during the initial handshake (Section 2.1.1). An example is given in Figure 2.1. Note that the `MP_CAPABLE` option is 4-bytes long in the last ACK, and not 2-bytes long as the TCP-ENO option in TCP.

In case of simultaneous open, both hosts send the SYN segment. Upon reception, as usual, they send back a SYN-ACK segment. It must contain the same sub-options in the same order than in the SYN segment. In case of failure (e.g., if both hosts want to play the role of host A), the host that has detected the failure sends a SYN-ACK segment without the TCP-ENO option.

3.1.2 Fallback

If the MPTCPesn negotiation fails, the MPTCP connection is compromised because the keys are not sent in the MP_CAPABLE option. There are several cases, some of them are listed below, but a solution is proposed only for the first one. Further work can bring more solutions.

- The client sends a MPTCPesn MP_CAPABLE option in the SYN segment but the server, that knows MPTCPesn, does not want (or cannot) use it. It can respond with a usual MP_CAPABLE SYN-ACK with its key, and the client send its own key in the ACK segment. In conclusion, only the SYN segment is different, but the handling of the last ACK segment could be different.

- Same case than the previous one, except that the server does not know MPTCPesn. It thus strips the unknown option and the connection is reduced to a TCP connection. The client can choose to close this connection to start another one with MPTCP instead of MPTCPesn.

- The server responds to the SYN segment with an incorrect MP_CAPABLE in the SYN-ACK segment (e.g., none of the encryption specs is common with those of the SYN segment).

- A segment (SYN, SYN-ACK or ACK) encounters some issue during its transmission, like a middlebox or an attacker that strips the MP_CAPABLE from it.

3.1.3 MPTCPesn Transcript

In order to protect its encryption negotiation from unwanted modifications, TCP-ENO defines a negotiation transcript that will be used by the encryption specifications. An encryption specification must fail with high probability if the negotiation transcript is not the same on both hosts (i.e., SYN and SYN-ACK segments have been tampered or even forged by an attacker). It is composed of the TCP-ENO options exchanged during the handshake, in the SYN segments.

MPTCPesn defines its own transcript based on the exchanged `MP_CAPABLE` options. More specifically, it is constructed from the following, in order:

1. The `MP_CAPABLE` option in host A's SYN segment, including the first 4 bytes (i.e., kind, length, subtype and version fields, and the flags).

2. The `MP_CAPABLE` option in host B's SYN-ACK segment (SYN segment for simultaneous open), including the first 4 bytes (i.e., kind, length, subtype and version fields, and the flags).

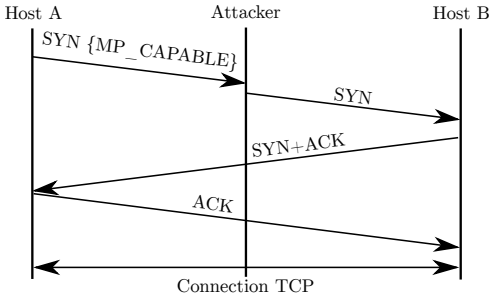


Figure 3.2: MP_CAPABLE removing attack on the initial MPTCPesn handshake

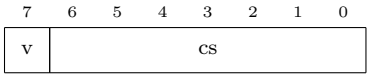


Figure 3.3: General format of TCP-ENO sub-options.

For the simultaneous open case, the `MP_CAPABLE` options are the same in the SYN segments than in the SYN-ACK segments of the same host. It is therefore useless to include the options of the SYN-ACK segments in the transcript.

Note that the transcript is not unique. It can be the same for several connections, but this is not an issue. We only want both hosts to agree on the options sent and received.

3.1.4 MPTCPesn weakness

The transcript enables hosts to detect any modification of the `MP_CAPABLE` options during the MPTCPesn handshake, but there is still one weakness: an attacker (or a middlebox) can easily remove the `MP_CAPABLE` options itself from the handshake (Figure 3.2). There will not be encryption spec negotiation and the connection will not be encrypted (and thus the transcript will not be used). In that case, the application may choose to close the connection in order to prevent attacks.

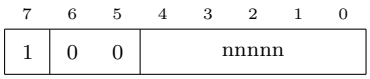
3.2 Sub-options

The sub-options are exchanged by means of the `MP_CAPABLE` option. The sub-options are 8-bits long. The first one, the `v` bit, determines if the sub-option contains additional data. The remaining seven-bits field is called the `cs` value. It is illustrated in Figure 3.3. These `cs` values can be divided in three categories, explained in the following sections : the general sub-options, the spec identifiers and the marker bytes.

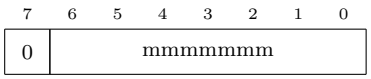
3.2.1 General sub-option

The general sub-option, already defined in Section 2.1.2 and represented in Figure 2.3, is used for general conditions that apply regardless of the negotiated encryption spec. There must be one and only one general sub-option in the `MP_CAPABLE` option. It contains 4 fields :

- **The four 0-bits** enable to recognize the general sub-option from other options (note that the `v` bit is included in this field and is set to 0).
- **The `zero (z)` bit** is reserved for future revisions of MPTCPesn (must be set to 0).
- **The `Application Aware (aa)` field** specifies if the application is aware of the use of MPTCPesn and if it can adapt its behavior to handle the encrypted MPTCP. This field also specifies if the receiver host application must be aware of it too. Here are the different values:



(a) Marker byte



(b) Second byte of the marker word. The first one is a marker byte

Figure 3.4: Format of the `marker byte` and `marker word` sub-options.

  - **00** Application is not aware of MPTCPesn.
  - **01** Application is aware of MPTCPesn, but does not request it from the other side.
  - **10** Reserved for future use but interpreted as the previous one (01).
  - **11** Application awareness is mandatory at both sides for use of MPTCPesn.

- **The `Role-override (b)` bit** is useful in case of simultaneous open, to break the symmetry in the role assignation (as previously explained in Section 2.1.3).

3.2.2 Spec identifiers

Spec identifiers enable hosts to negotiate encryption schemes. Each identifier (i.e., the `cs` field) represents a specific encryption scheme and can take values between 0x20 and 0x7f. More than one identifier can be reserved by an encryption protocol (like tcpcrypt) in order to specify, for example, which key-agreement scheme to use, or if we want to resume a previous connection. The requirements for the specs that can be negotiated are described in Section 3.3.

3.2.3 Variable-length sub-options

As an overview on this point has already been given in Section 2.1.5, we focus on the formats of the `marker byte` and `marker word`, shown in Figure 3.4. The `v` bit of the marker byte is set in order to advertise additional data. Both following bits are set to 0 in order to recognize this sub-option. The last 5 bits' value, plus one, is the length of the additional data. In summary, we have a marker byte indicating a length N (i.e., the `nnnnn` value is N-1), followed by an encryption spec (with the `v` bit set), itself followed by N bytes of additional data. The marker byte can thus advertise additional data length from 1 to 32 bytes.

If 32-bytes length is not enough, the marker byte is extended to a marker word, adding 7 bits to encode the length of the additional data. So, the length is represented by the `nnnnnnmmmmmmm` field (i.e., the concatenation of the `nnnnn` and `mmmmmmm` fields). It can be recognized from a marker byte followed by an encryption spec thanks to the `v` bit of the second 8-bits field, set to 0 (instead of 1 for an encryption spec).

3.3 Encryption specs requirements & Session ID

MPTCPesn, along with TCP-ENO, enables a large flexibility in the designs of encryption schemes. But TCP-ENO designers also try to keep away these design differences from the application writer. The encryption schemes that want to reserve some `cs` value must therefore fulfill a list of requirements (available in the TCP-ENO draft [10]).

In short, the specs must protect the TCP (for TCP-ENO) or MPTCP (for MPTCPesn) data stream with authenticated encryption, by means of protocols providing, at least, a security of 128 bits and forward secrecy some bounded, short time after the connection closure. Null cipher suite cannot be negotiated. If an attacker closes the connection prematurely, it must raise an error (i.e., the hosts will know there are some data lost). URG flag and urgent pointer must be protected or deactivated. Finally, the specs must define a `Session ID` for each connection (detailed below).

There is a last requirement that applies only to MPTCPesn. As the keys have been removed from the MPTCP handshake, encryption specs must define the MPTCP connection tokens and the Initial Data Sequence Numbers (IDSN). They must also define the value used for the MP_FASTCLOSE option (Section 1.3.5). Note that each host must know the remote token before establishing a new subflow, the local IDSN before transmitting data, and the value to put in the MP_FASTCLOSE.

The session ID (introduced above) identifies an encrypted connection and must be unique at any point in time with high probability (even if one end of the connection does not respect the specs and maliciously manipulates the protocol in order to create a duplicated session ID). It must be at least 33-bytes long, with the first byte equal to the spec identifier. The other bytes must be a collision-resistant hash of connection information, including fresh data contributed by both sides: public cryptographic parameters having a related private parameter (e.g., public keys) and the negotiation transcript (Section 2.1.4 for TCP-ENO and Section 3.1.3 for MPTCPesn). They must be computationally indistinguishable from random bytes. The session ID can be public.

3.4 Conclusion

TCP-ENO enables hosts to easily negotiate encryption schemes such as the well-known TLS and tcpcrypt protocols, during the TCP handshake. It provides security against forging and tempering attacks on the handshake (but not options dropping attacks) by means of a transcript that must be used by the encryption schemes. Moreover, all the encryption schemes that can be negotiated must fulfill a list of security requirements. But TCP-ENO can be used only with TCP. This chapter has designed MPTCPesn, the MPTCP equivalent of TCP-ENO. It is now possible to negotiate encryption schemes during the MPTCP handshake.

Chapter 4

MPTCPsec

There is now a means to negotiate encryption schemes: MPTCPesn. But they are not yet the negotiable encryption schemes themselves. We have thus designed MPTCPsec (for MPTCP secure), a cryptographic protocol that improves the reliability and the security of data transfers over MPTCP, in the presence of adversaries, and that can be negotiated by means of MPTCPesn.

MPTCPsec is application independent, fast and benefits from the MPTCP advantages (e.g., by protecting the influence of subflows on other ones). It is designed for MPTCP and it is even integrated in it. The protocol is based on tcpcrypt [12], and some parts of it are even identical (e.g., the handshake). In fact, it could also be adapted to be coupled with TLS [32], but we have chosen tcpcrypt because it cannot be run above MPTCP while TLS can. In other words, we give to hosts the choice between two different encryption protocols, with different properties and goals, instead of letting them choose between two versions of the same protocol (i.e., TLS above MPTCP and TLS in MPTCP).

The design of MPTCPsec is detailed in this chapter. Firstly, the security goals are fixed in Section 4.1, in terms of attacker models and common attacks on MPTCP that MPTCPsec should overcome. From that, a first sketch of the solution is made. The following section, Section 4.2, gives an overview of MPTCPsec, each part being explained in more details afterwards. Section 4.3 is all about the AEAD algorithms. It also gives a quite detailed pseudo code of the protocol. Then Section 4.4 and Section 4.5 detail the protections of the options, i.e., the DSS option for the first of both sections, and all other options for the second one. The protocol of the fragmentation and re-keying are respectively explained in Section 4.6 and Section 4.7. Then specific values and variables are defined, i.e., the IDSN and token in Section 4.8 and the spec identifiers negotiable with MPTCPesn in Section 4.9. Section 4.10 discusses the proposition of tcpcrypt's designers to replace the Keep-Alive mechanism by a re-keying mechanism. At this point, MPTCPsec is completely defined. We thus show that the goals are achieved in Section 4.11, and we compare our solution with TLS above MPTCP in Section 4.12. Some future work can be done, that could improve MPTCPsec, as detailed in Section 4.13. And a small conclusion is proposed in Section 4.14.

4.1 Security goals and first sketch

The main purpose of MPTCPsec is to overcome the security issues of MPTCP. We have aimed for MPTCPsec to be secure even in presence of an adversary that can read, drop, modify and even forge segments on all the subflows of the connection. But we have not aimed for MPTCPsec to ensure identity authentication, and it is therefore vulnerable to Man-In-The-Middle attacks (Section 1.1.6). This section presents the security weaknesses of MPTCP (the attacks that can happen) that MPTCPsec should overcome and then it gives a first sketch of the solution.

MPTCP uses multiple paths and needs segments to be in sequence at the TCP layer and at the MPTCP layer. It thus makes some attacks harder than in TCP, but not impossible. For instance, data is sent in clear and an attacker can therefore read it ; but in MPTCP, he can only read data sent on paths on which he is. It means that the attacker can read all data, but only if he is on all the paths used by the connection. In other words, even if some attacks are harder, MPTCP is as secure as TCP. Indeed, an attacker on all the subflows of an MPTCP connection is as powerful as an attacker on the sole flow of a TCP connection. Some common attacks are given below, but a deeper threat analysis is made in [36]. Attacks 1 and 2 enable the attacker to capture the data, attacks 3 to 5 enable him to modify the data and attacks 6 and 7 are mostly Denial of Service (DoS) attacks.

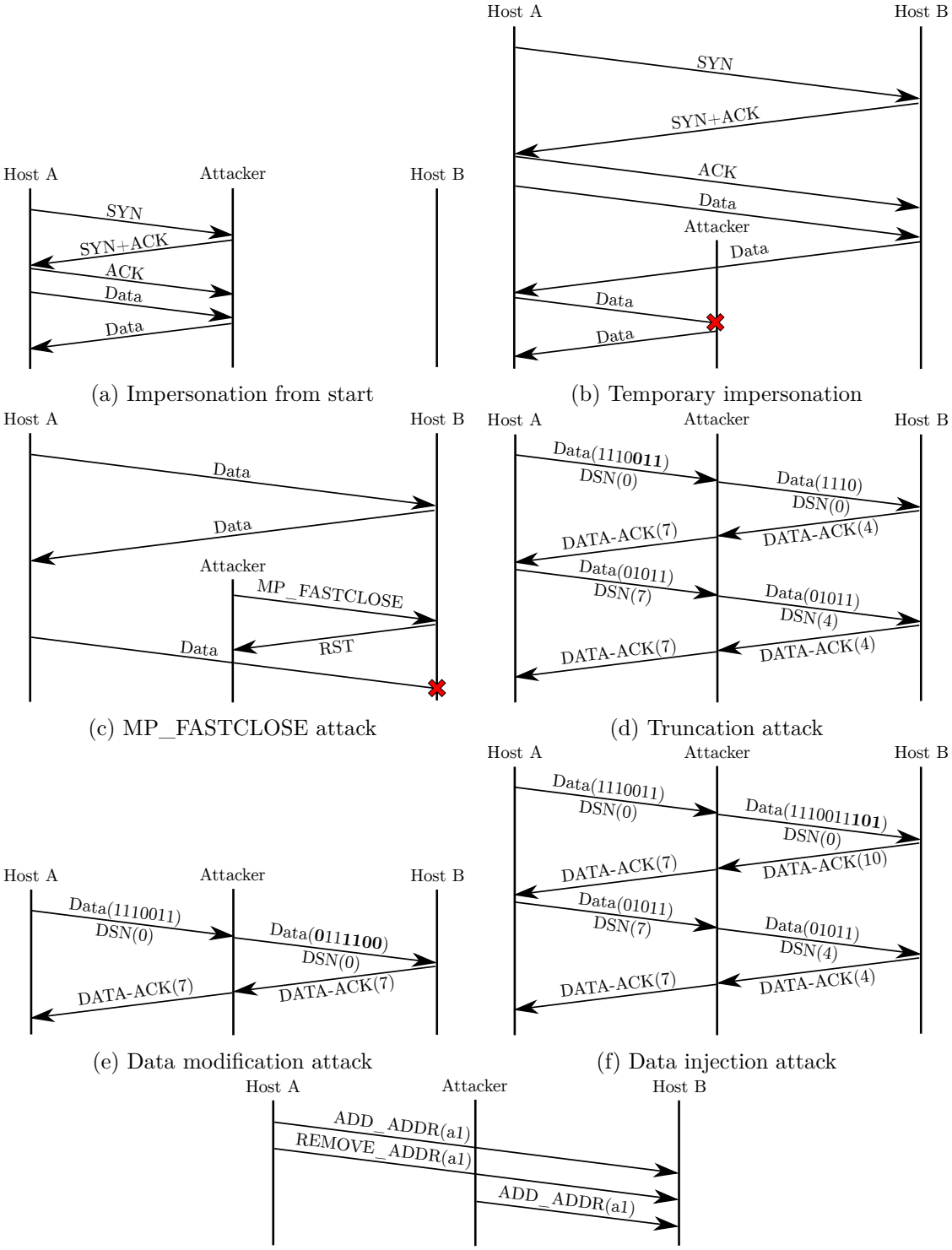
1. **Eavesdropping**: an attacker can read the data sent between hosts.
2. **Identity spoofing**: an attacker can impersonate a host whenever he wants during the connection. He can impersonate the client or the server, right from the start of the connection (Figure 4.1a) or only for a part of it (Figure 4.1b).
3. **Data truncation**: an attacker can prevent some data from reaching their final destination, i.e., the other host. He can do it either by closing the connection prematurely (Figure 4.1c) or by removing some data from the data stream and by modifying the DSNs of all subsequent segments (Figure 4.1d).
4. **Data modification**: an attacker can modify the data as he wants, i.e., he can change the bits (but without modifying the length of the message, as shown in Figure 4.1e).
5. **Data injection**: an attacker can inject data in the data stream. To do so, and as for the data truncation, he must modify the DSNs of all subsequent segments (Figure 4.1f).
6. **Segment replay**: an attacker can replay a segment already sent (or a part of it). As the replay attack of payload is already covered by the above attacks, we focus more on the replay of MPTCP options here. The attacker can replay options (Figure 4.1g) in whichever segment he wants.
7. **Option removal**: an attacker can remove a MPTCP option from a segment.

While designing MPTCPsec, we have tried to overcome all these issues. The first attack can be easily avoided with data encryption, and attacks from 3 to 5 with data authentication[1]. These two words (encryption and authentication) bring us to AEAD algorithms (Section 1.1.7). To protect the data from these four attacks, MPTCPsec can use them as the *plaintext* input of the AEAD. In order to protect the options from attacks 6 and 7, it is necessary to use them as associated data (encrypting them can cause some troubles, it is discussed later in Section 4.5). We still need to choose a unique nonce, and the 64-bit DSN seems perfect for this role (the 32-bit DSN must be forbidden because it is too short to be unique during connection lifetime). Indeed, the DSN could be the same for two different segments only if it has made at least one complete cycle (i.e., 10^{19} bytes of data sent without a re-keying), which is barely thinkable. The cipher produced by the AEAD is put in the payload of the segment.

The identity spoofing attack is overcome as soon as the connection is established, thanks to the authentication, but to be able to completely solve it (i.e., even during the connection establishment), we need certificates. The drawback of using certificates is that applications need to be modified/adapted. We have chosen to not use them, as we work with tcpcrypt.

For performance reasons, we have chosen to make the options authentication in another step than the data encryption. Indeed, in case of re-injection (data sent again but on another

[1]In order to prevent the third attack, the authentication of DSS and MP_FASTCLOSE options is also needed.



(a) Impersonation from start

(b) Temporary impersonation

(c) MP_FASTCLOSE attack

(d) Truncation attack

(e) Data modification attack

(f) Data injection attack

(g) Replay attack

Figure 4.1: Attacks on MPTCP

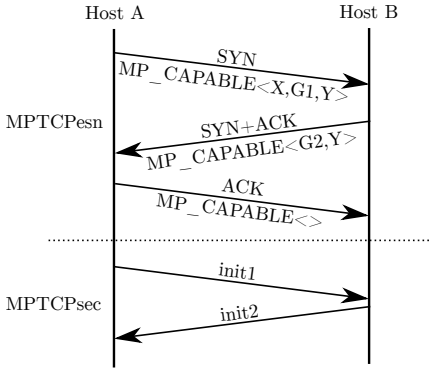


Figure 4.2: Connection establishment with an MPTCPesn handshake followed by an MPTCPsec handshake.

subflow), it is computationally expensive to encrypt the data again (whereas it is useless). But since options can change (e.g., the Subflow Sequence Number of the DSS), the authentication tag must be computed again. Two steps are needed.

In order to link the data with their header, MPTCPsec uses the tag produced by AEAD on the data as the *associated data* input of the AEAD that computes the tag for the options (themselves used as associated data too). The option's tag is put after the cipher block of the data (that includes the data tag).

4.2 Overview

This section gives an overview of MPTCPsec. The details and corner cases are kept for the later sections, as well as most of the explanations and motivations. It starts with the handshake and then explains how the data is protected. The protection of the MPTCP options is explained right afterwards. This section ends with the operation order that hosts must follow when sending and receiving segments.

4.2.1 Handshake

As only identifiers can be exchanged in the MPTCPesn's MP_CAPABLE of the MPTCP handshake, MPTCPsec needs to exchange control data again (including agreement on a shared secret). And as the MPTCP handshake is finished, this must be done in the data portion of MPTCP segments.

The MPTCPsec handshake and the keys derivation process are exactly the ones of tcpcrypt (Section 2.2.1). The handshake does not need to be modified as MPTCP is not even used at this time. Indeed, since the IDSN is still not established (MPTCPesn's requirements, Section 3.3), the DSS option cannot be used. The handshake is therefore made on TCP. MPTCPsec only needs to get back the negotiated shared secret at the end of the handshake. The whole connection establishment is represented in Figure 4.2.

The IDSN and token values are derived from the Session ID, itself derived as in tcpcrypt. It is explained in more details in Section 4.8. MPTCPesn also requires to define the value used for the MP_FASTCLOSE. It is discussed in Section 4.5.5.

Section 4.9 defines three spec identifiers that can be negotiated with MPTCPesn. As for tcpcrypt, they mainly define the key-agreement scheme to be used.

4.2.2 Data protection

Application data is encrypted and authenticated with AEAD algorithms, whose nonce is based on the DSN. Some variables must be added to this nonce to keep it unique in all cases, for instance when segments are re-injected. The inputs are details in Section 4.3.1. There is no associated data.

MPTCPsec does not use records like TLS and tcpcrypt do, but uses DSS options. A block of encrypted data is covered by one and only one DSS. The ciphertext length is given by the Data-Level Length field, and the usual control bits (e.g., the tcpcrypt rekey bit) can be added in the DSS option. It enables to fragment a ciphertext in several segments by covering all these segments by only one DSS (fragmentation is discussed in Section 4.6). But this also entails that all the fragments must be sent in the same subflow than the first one (i.e., the one with the DSS option).

4.2.3 Option protection

MPTCPsec authenticates some options but does not encrypt any of them. We could use them as associated data of the AEAD used for the data, but it would entail to recompute the whole ciphertext each time the segment is re-injected on another subflow, since the SSN of the DSS option would change. MPTCPsec therefore authenticates the options separately, so that only the options tag must be recomputed in case of re-injection. It forbids any change of the segments in case of re-transmission.

In order to link the data with its headers, MPTCPsec includes the data authentication tag in the computation of the options tag, which is made with the AEAD used for the data. It is then appended to the ciphertext in the payload (i.e., after the encrypted data and its authentication tag). Figure 4.3 represents this segment format (assuming that the AEAD algorithm used puts the tag at the end of the ciphertext).

The tag increases the DSN. It enables to know, in case of a re-keying mechanism (Section 4.7), if the options have been authenticated with old keys (DSN's option tag $<$ re-keying DSN) or new keys (DSN's option tag $\geqslant$ re-keying DSN).

Moreover, it makes replay attacks impossible. Indeed, since the options are associated to a given DSN, MPTCPsec only needs to check that the segment covering this DSN has not been already received (e.g., on another subflow) before applying the options. The DSN should make a whole cycle, without re-keying, to make a replay attack possible... and it is barely possible with 64-bits DSN. The options are thus applied once and only once.

A last point is that, since the option tag is covered by the DSN, it must be acknowledged (like regular data). The acknowledgment of the tag is, in fact, the acknowledgment of its options. And when an ACK is received, we want to know which options have been received by the other host. So even if the tag changes in case of re-injection (but not its size), we want its semantics to be the same. Therefore, MPTCPsec forbids to change the semantics of the options in case of re-injection. It disables the attack that consists of dropping segments with options that are not convenient for us in the hope that the sender will not send these options again (i.e., it sends the segment again, but with another options).

It involves that there is data (i.e., the options tag) in the segment as soon as there is at least one MPTCP option that needs to be authenticated. A DSS is thus needed, even if there is not application data in the segment. It is an issue for the pure ACK segment, which is discussed in Section 4.4.2. In the case of a DSS covering several segments, the options tag is put in the last segment and authenticates the options of all the segments covered by the DSS (explained in more details in Section 4.6).

Figure 4.3: Segment format (assuming that the AEAD algorithm used puts the tag at the end of the ciphertext), with *tag* the authentication tag produced by the AEAD.

Options	**Protection**	**Re-injection**
MP_CAPABLE	Included in the MPTCPsec transcript	Cannot be re-injected
MP_JOIN	HMAC exchanged during the handshake (in the options)	Cannot be re-injected
ADD_ADDR	Included in the options tag	Nothing can be changed
REMOVE_ADDR	Included in the options tag	Nothing can be changed
DSS	Included in the options tag	The Subflow Sequence Number must be modified. The m and a flags can be modified, and thus the Data ACK and the DSN can be truncated or not.
MP_FASTCLOSE	HMAC of the remote key with the local key (in the option)	Nothing can be changed
MP_PRIO	Included in the options tag	If the segment is sent to the concerned AddrID, it can be not specified, otherwise it must be specified.
MP_FAIL	Should not be used	

Table 4.1: This table is a summary of how the MPTCP options are protected and what can be or must be changed in case of re-injection. In fact, MPTCPesn is more in charge of the MP_CAPABLE option than MPTCPsec.

Table 4.1 gives a summary of how the options are protected and about what can be or must be changed in the options in case of re-injection (in order to keep the same semantic). The details are given in Section 4.5, except for the DSS option for which Section 4.4 is entirely dedicated. Note that with MPTCPsec, the options are sent reliably while they are not with MPTCP. As counterpart, the whole option processing could be slower and more constrained.

4.2.4 Operation order

When sending or receiving segments, a host must follow a precise order for the operations in order to ensure the secure properties aimed. Here is what a host does in case of:

- **Sending a segment**: first, the host executes the AEAD on the data to be sent. Then it chooses the subflow on which it will send the segment. It creates the header and computes the authentication tag for it. And finally it sends the segment on the subflow.
- **Re-transmitting a segment**: the host sends the segments again, with the same header and the same tags, on the same subflow.
- **Re-injecting a segment**: the host modifies the DSS and can or must modify the others

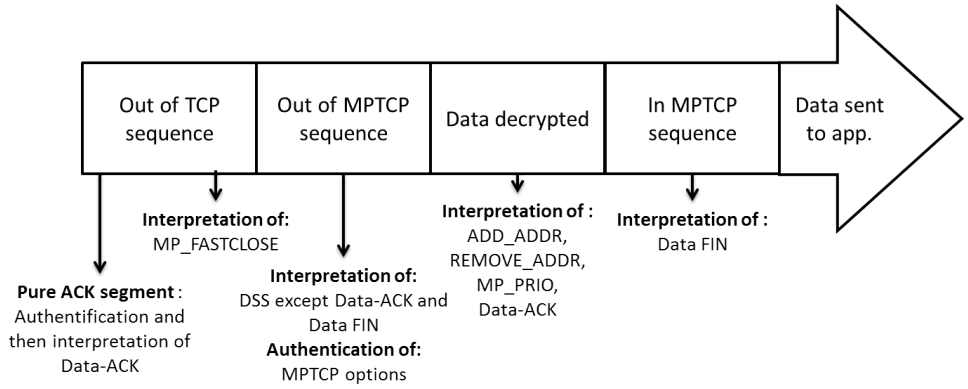


Figure 4.4: Timeline of the interpretation of the MPTCP options.

options too, such that their semantics are not modified (e.g., the MP_PRIO option needs to specify the Address ID it concerns if it is not sent to this Address ID). Then it recomputes the authentication tag of the options, but does not modify the ciphertext. In case of too small MTU, it fragments the segment (Section 4.6).

- **Receiving a segment**: the host checks if the segment is in the MPTCP window and not already received (otherwise it is dropped). Then it decrypts the ciphertext, checks the options tag, and if there is no error, it applies/treats the options (otherwise it closes the subflow). It sends the data to the application only when they are in sequence. In case of fragmented segment, it must wait to receive all the fragments before processing the segment. Then it follows the specifications given in Section 4.6.

When receiving a segment, a host must check both the authentication of the data and authentication of the options before applying the MPTCP options. Otherwise, an attacker could make replay attacks of the options. Indeed, if he modifies a bit in the ciphertext, the authentication of the option will still succeed and the option will be applied. But the subflow will be closed because the authentication of the data will fail. The segment will not be acknowledged, and sent on another subflow. The options will therefore be applied twice (or more).

A more precise order of the option interpretation is given in Figure 4.4. Pure ACK segments and MP_FASTCLOSE options do not contain MPTCP data (as explained in Section 4.4.2) and can thus be directly handled. The DSS option is partially interpreted before the option authentication check. Most MPTCP options, and the Data-ACK of the DSS, can be interpreted only after the data decryption, as explained above. Finally, the Data FIN must be in sequence before being interpreted.

4.3 AEAD

This section firstly describes precisely the inputs of the AEAD for the data encryption and authentication, and for the option authentication. Then, it discusses about what to do when receiving a segment with an incorrect authentication tag. It finishes with the discussion about using an AEAD instead of an HMAC to authenticate the options.

4.3.1 Inputs

The AEAD algorithms need four inputs, as explained in Section 1.1.7. This section specifies what precisely these inputs are for MPTCPsec. A pseudo-code for the encryption, decryption and authentication of segments is given at Figure 4.5.

```
1  def 64_bits_DSN(32_or_64_bits_DSN):
2      return the corresponding 64_bits_DSN
3  def 64_bits_Data_ACK(32_or_64_bits_Data_ACK):
4      return the corresponding 64_bits_Data_ACK
5  def extract_tag(ciphertext):
6      return the authentication tag included in the ciphertext
7  def subflow_id(subflow):
8      val trunc_HMAC_1 = first 32 bits of the SYN-ACK MP_JOIN HMAC of
           subflow
9      val trunc_HMAC_2 = first 32 bits of the ACK MP_JOIN HMAC of subflow
10     return subflow = initial_subflow ? 0x0 : {trunc_HMAC_1, trunc_HMAC_2}
11 def aead_encrypt(plaintext, key, unique_nonce, associated_data):
12     [...]
13     return ciphertext
14 def aead_decrypt(ciphertext, key, unique_nonce, associated_data):
15     [...]
16     return failure ? error : plaintext
17
18 def construct_segment(plaintext, subflow, encr_key):
19     val header = construct_header()
20     val auth_opt = MPTCP options of the header that must be authenticated
21     val DSN = 64_bits_DSN(header.DSS.DSN)
22     val ciphertext = aead_encrypt(plaintext, encr_key, {DSN, 0x0}, null)
23     val data_tag = extract_tag(ciphertext)
24     val option_tag = aead_encrypt(0x0, encr_key, {DSN, 0x1, subflow_id(
           subflow)}, {auth_opt, data_tag})
25     val payload = {ciphertext,option_tag}
26     return {header,payload}
27
28 def decrypt_segment(segment, subflow, decr_key):
29     val auth_opt = MPTCP options of the header that must be authenticated
30     val DSN = 64_bits_DSN(segment.header.DSS.DSN)
31     val ciphertext = segment.payload.ciphertext
32     val data_tag = extract_tag(ciphertext)
33     val option_tag = segment.payload.option_tag
34     val result = aead_decrypt(option_tag, decr_key, {DSN, 0x1, subflow_id
           (subflow)}, {auth_opt, data_tag})
35     if (result == error) : return option_tag_error => will close the
           subflow
36     val plaintext = aead_decrypt(ciphertext, decr_key, {DSN, 0x0}, null)
37     if (plaintext == error) : return data_tag_error => will close the
           subflow
38     Apply_options(segment.MPTCP.options)
39     return plaintext
40
41 def construct_pure_ack_segment(encr_key):
42     val header = construct_pure_ACK_header()
43     val Data_ACK = 64_bits_Data_ACK(header.DSS.Data_ACK)
44     val nonce = {Data_ACK, 0x2, header.DSS.reserved_bits_and_flags}
45     val ack_tag = aead_encrypt(0x0, encr_key, nonce, header.DSS)
46     return {header, ack_tag}
47
48 def check_pure_ack_segment(segment, decr_key):
49     val Data_ACK = 64_bits_Data_ACK(segment.DSS.Data_ACK)
50     val nonce = {Data_ACK, 0x2, segment.DSS.reserved_bits_and_flags}
51     return aead_decrypt(segment.ack_tag, decr_key, nonce, segment.DSS)
```

Figure 4.5: Pseudo-code for encryption, decryption and authentication of segments, given that {} are concatenations. For re-injection, it is nearly the same as `construct_segment()` except that the ciphertext has already been computed and that the DSN must be the same as in the first segment. And for the re-transmission, MPTCPsec forbids to change anything of the segment that is authenticated.

The symmetric keys are computed during the MPTCPsec handshake (Section 4.2.1) and are the same for the data authentication and for the options authentication. But they are different for the encryption and for the decryption (tcpcrypt derives one set of keys for each direction: *client* → *server* and *server* → *client*).

The unique nonce of the AEAD must be different for each different block of data encrypted with the same key. It means that we only need to ensure its uniqueness for data sent in one direction (e.g., *client* → *server*), since the other direction uses another key. MPTCPsec uses the 64-bits DSN for the AEAD nonce, as said previously. But we must add something because the AEAD is run twice for the same DSN: once for the data, and once for the options (i.e., the AEAD nonce is not unique). We can append a constant value to the DSN. MPTCPsec appends 0x0 when running the AEAD on the data, and appends 0x1 when running it on the options. Now, our AEAD nonce is thus the concatenation of the DSN and a constant value.

But this is not enough because segments can be re-injected on other subflows, giving DSS options with different Subflow Sequence Numbers but with the same DSN. The data to authenticate are thus different, but the AEAD nonce is the same. No unique subflow ID exists for now[2]. We need a way to compute a value for each subflow, unique in the scope of the sender but computable by both hosts, in order to incorporate it in the AEAD nonce. This value must be computed on fresh data coming from both hosts, and should ideally be collision resistant even if one host has poor randomness. The HMACs computed on the MP_JOIN nonces (and on the Address IDs, as explained later in Section 4.5.2) during the subflow's establishment seem perfect for this role. MPTCPsec defines the subflow ID as the concatenation of the first 32 bits of the MP_JOIN's HMAC in the SYN-ACK segment and the first 32 bits of the MP_JOIN's HMAC in the ACK segment. Since the initial subflow does not use MP_JOIN options and so nor HMACs, MPTCPsec uses the 64-bits constant value 0x0.

In summary, hosts keep in memory one 64-bits subflow ID per active subflow, which is the concatenation of the first 32 bits of both HMACs of the subflow's handshake. The subflow ID is concatenated to the AEAD nonce (i.e., the DSN and the constant value, as shown in the pseudo-code). It enables to modify the options (but not their semantics) when segments are re-injected (i.e., on another subflow), contrary to when they are re-transmitted (i.e., on the same subflow).

If the case of re-transmission was considered, it would be harder to define a unique AEAD nonce. It is for this reason and for performance reasons that MPTCPsec forbids to change anything of the segment when it is re-transmitted on the same subflow. There are some consequences, such as the fact that a delayed segment, re-transmitted many times, could transport outdated options. Another issue can arise if the MTU drops suddently. Indeed, the segment will need to be fragmented, but the options cannot be changed (but they can be spread among all the fragments) and other options cannot be added. This is a constrained fragmentation.

There is also the case of pure ACK segments (explained later in Section 4.4.2) for which the AEAD nonce is the 64-bits Data ACK, the reserved bits and the flags of the DSS. In order to avoid any collision with the Data ACK and the DSN, MPTCPsec appends 0x2 after the Data ACK, which is 64-bits long like the DSN.

Concerning the associated data, it is quite straightforward. There are no associated data for the encryption and authentication of the data. For the authentication of the options, there are put as associated data of the AEAD in the same order than in the segment header, followed by the data authentication tag (Section 4.5.1).

Finally, as the plaintext input is what we want to encrypt, we must use the data for this

[2]One could think that concatenation of both Address ID could give a unique subflow identifier. But unfortunately, several subflows can be opened with the same pair of Address ID, on different ports. And we cannot use the ports in the nonce since they can be changed by middleboxes.

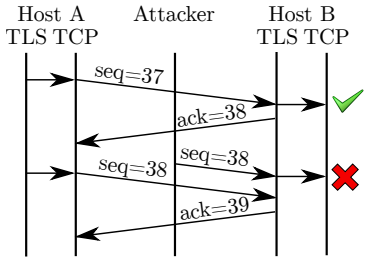


Figure 4.6: Example of segment accepted by TCP but not by TLS and that will thus never be sent again.

input when running the AEAD on the data. But for the AEAD used to authenticate the options, this is less obvious. Indeed, we do not need to encrypt something, but the AEAD has to have a plaintext input. MPTCPsec uses a constant value, 0x0, for the plaintext input and it uses the whole ciphertext as authentication tag. This value should be as small as possible (i.e., as many bytes as the minimum required by the AEAD algorithm used), because the ciphertext length and the computational cost increase with the plaintext length. The other host, upon reception of the segment, only needs to decrypt the ciphertext with the options as associated data. The value to encrypt can be, in fact, anything else and could be used for many purposes. We develop this idea in more details in Section 4.13.2.

4.3.2 Authentication failure

In case of authentication failure, TLS closes the connection. This choice is motivated by two reasons.

The first one is the oracle given to the attacker (if the connection is not closed). The attacker could send segments with different authentication tags until a segment is accepted. Even if the encryption scheme is theoretically secure, this can be dangerous (Section 1.1.1). This issue concerns MPTCPsec too.

The second reason comes from the separation of the TLS layer and TCP layer. The authentication check is made in the TLS layer. Even if this authentication fails for a segment, TCP has already accepted it and sent an acknowledgement to the other host (an example is given in Figure 4.6). So, it will never be transmitted again by the other host, which thinks that the segment has correctly arrived. These data are therefore definitely lost. This is useless to keep the connection open, so TLS closes it. But MPTCPsec authenticates the DSS of MPTCP, and the authentication check is done in the same layer as MPTCP. Both hosts are therefore always synchronised because if the authentication check fails, MPTCP does not send an acknowledgement and the other host knows that the segment is not correctly received. MPTCPsec is therefore not concerned by this issue.

In conclusion, MPTCPsec should close the subflow when the authentication fails, due to the oracle issue. But there is no need to close the whole connection.

Note that the DoS attack which enables this, is limited. Indeed, the attacker must be on the subflow to know the correct TCP sequence number and the port. Without that, he will not be able to send a segment wrongly authenticated but accepted by TCP (and so will not be able to force the closure of the subflow). This DoS attack is similar to the RST attack. An attacker only needs the TCP sequence number, the port and the IP address to forge a valid RST segment which will close the TCP connection (i.e., the subflow itself). So MPTCPsec should either close the subflow in case of AEAD failure and not try to protect itself from RST attacks, or it should not close the subflow in case of AEAD failure and protect itself from RST attacks. One of them

without the other one is useless.

4.3.3 Options authentication

MPTCPsec uses AEAD algorithms to authenticate the options, instead of an HMAC (Section 1.1.4) like the well-known HMAC_SHA256 (see [30] for precise specifications) that is specially designed to produce authentication tags. We explain in more details how to do so and the reasons of this choice.

AEAD algorithms impose a plaintext input that will be encrypted. We do not have data to encrypt when authenticating the options. But we can simply use a constant or even random value, as small as possible to minimize the computational cost and the ciphertext length. The whole ciphertext is treated as the authentication tag.

At first glance, one could think that since AEAD algorithms are designed to authenticate but also encrypt, they should be slower than HMAC_SHA256. But this is not true. The Intel processors now integrate the AES-NI instruction set whose purpose is to improve the speed of applications performing encryption and decryption using AES. Thanks to that, AES-GCM is quicker than HMAC_SHA256, as confirmed by our performance tests results (shown in Figure 4.7). Indeed, a HMAC needs to run twice the hash function. HMAC_SHA256's throughput is at most the half of the sha256's throughput (in grey in Figure 4.7). Moreover, we launch the AEAD on small data (at most 40 bytes of options, and a plaintext as small as possible), so only values before 2^6 bytes matter.

For CPUs without the AES-NI instruction set, AES-GCM is slower than HMAC_SHA256, and even sometimes too slow to be used. But CHACHA20_POLY1305 has been developed to be fast and can be used on such CPU (for instance, Firefox uses this combination of AES-GCM for CPU with the AES-NI instruction set and CHACHA20_POLY1305 for those without, as stated in [37]).

Using only one AEAD makes things simpler (instead of one AEAD and one HMAC algorithms), as well as for the design than for the implementation.

4.4 DSS and ACK

This section is all about the DSS. The first part discusses the authentication of the DSS while the second part details how to handle the acknowledgment segments.

4.4.1 DSS authentication

MPTCPsec authenticates the DSS option. The Checksum of the DSS option is therefore useless.

MPTCPsec uses 64-bits DSN (or Data ACK for ACK segments, as explained in Section 4.4.2) for the AEAD nonce in order to prevent it from making a whole cycle without doing any re-keying mechanism in the meantime. Otherwise, MPTCPsec would be weakened and replay attacks could occur. But the 32-bits DSN (and Data ACK) can be used in the DSS. The exact format of the AEAD inputs is explained in Section 4.3.1.

The DSS is secure. Forgery attacks cannot happen thanks to the authentication tag and replay attacks are harmless thanks to MPTCP (a DSS out of sequence is simply ignored, and as soon as an authenticated option is sent, the DSN is increased thanks to the tag). One could say segments can be replayed after 2^{32} bytes, because the 32-bits DSN of the segment would be in

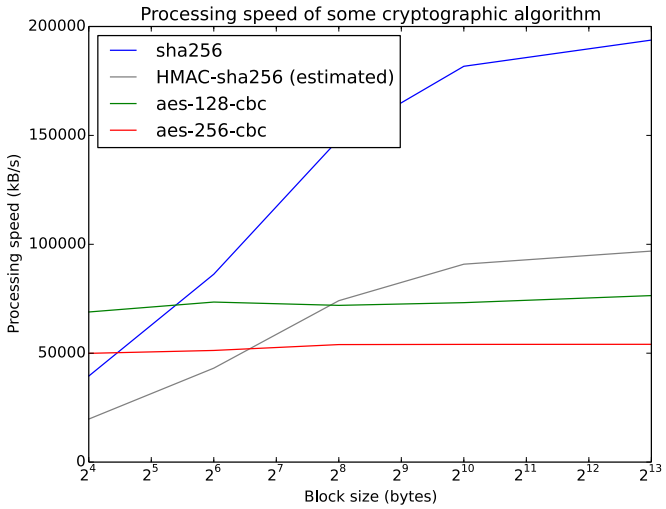


Figure 4.7: Performance tests made with the openssl [38] command-line "openssl speed [algo]" on sha256, aes-128-cbc and aes-256-cbc algorithms, with different block sizes from 16 bytes to 8192 bytes. The x axis values are the block size in byte, with a logarithmic scale, and the y axis values are the processing speeds, in kB/s of data processed. The HMAC-sha256 processing speed is estimated as the half of the sha256 processing speed. The computation was made on an Intel(R) Core(TM) i3-2350M CPU @ 2,3 GHz.

sequence again. But the segment would be authenticated with a wrong 64-bits DSN and the verification of the authentication tag would fail.

4.4.2 Acknowledgments

There is an important issue for the acknowledgments. With the current protocol, as ACK segment contains a DSS option, they also contain an authentication tag, in the payload. Thus, they always contain data (there are no more pure ACK segments), and data must be acknowledged. In other words, each ACK segment must be acknowledged itself by another ACK segment. This is an endless loop. To solve this issue, in case of a pure ACK segment (i.e., a segment that contains neither data nor other MPTCP options than the DSS), MPTCPsec still puts the options tag in the payload but does not map it with the DSN. It means that the segment can be easily replayed. But since this segment is a pure ACK segment, this is obviously not an issue. It also means that we cannot use the DSN anymore for the AEAD nonce because the segment is not linked to a specific DSN. MPTCPsec uses the 64-bits Data ACK instead of the DSN to authenticate a pure ACK segment. Moreover, in order to be compliant with rekeying (Section 4.7), it must be authenticated with the key used to encrypt the data it acknowledges.

It raises another issue: the Data ACK can be on 32 bits or on 64 bits, and two different pure ACK segments (not re-transmitted ones) can acknowledge the same data. In other words, we have two different associated data for the same nonce (the 64-bits Data ACK). Yet, we need a nonce unique for each authentication of different data. This issue can be solved if MPTCPsec adds the `a` flag of the DSS to the nonce. But in order to ensure security even if the host makes weird things like changing the reserved bits of the DSS, MPTCPsec concatenates all the flags and the reserved bits of the DSS to the nonce when authenticating a pure ACK segment.

A last point needs to be discussed. Even if the tag is not covered by the DSN, it is covered by the TCP Sequence Number. Therefore, if the TCP sending window of a host falls to 0, it cannot send pure MPTCP ACK segments anymore. The connection will not be frozen since TCP

ACK segments can still be sent (they do not need tags in the payload). The TCP window will thus be freed and the host will be able to send MPTCP ACK segments. But it could, perhaps, slow down the data flow. A simple solution is to reserve a portion of the TCP sending window exclusively for MPTCP ACK segments (for 5 or even 10 of them). Hosts could also ensure that the portion of the TCP window reserved for the MPTCP ACK segment is used only when new data is acknowledged. In that case, space for only one or two MPTCP ACK segments should be enough (i.e., one or two options authentication tag, whose length depends of the AEAD chosen). Note that this solution is optional, but if it is not implemented, MPTCPsec must enable pure TCP ACK segments (i.e., without any MPTCP option, without any data and without any tag), otherwise the connection could be frozen.

4.5 MPTCP options

This section discusses the security of the MPTCP options, firstly by completing the overview given in Section 4.2.3, and then case-by-case. The Table 4.1 gives a summary of how the options are protected and about what can be or must be changed in case of re-injection (in order to keep the same semantic). The DSS option is explained in Section 4.4, and it is MPTCPesn that is in charge of the MP_CAPABLE option.

4.5.1 General case

MPTCPsec authenticates some options but does not encrypt any of them. The DSS and the MP_JOIN options cannot be encrypted with MPTCPsec. The DSS, with the Data-Level Length, indicates where the ciphertext finishes and also where is the option tag (it could even be on another segment in case of fragmentation, as explained in Section 4.6). So, it is impossible to decrypt a segment without reading the DSS option. And as the MP_JOIN opens a new subflow, the host cannot know which MPTCPsec connection this segment belongs to if it cannot read the MP_JOIN option. For the other options, there is no obvious reason not to encrypt them, but it can cause some trouble with middleboxes (it is more detailed in the possible improvements, in Section 4.13.1). We have chosen to avoid to complicate MPTCPsec by encrypting some options but not all of them.

MPTCPsec authenticates the options in another step than encrypting the data, so that hosts only need to recompute the authentication tag of the options in case of re-injection. In case of re-transmission, MPTCPsec forbids to change anything of the segment. In order to link the data with their header, MPTCPsec authenticates the options along with the data authentication tag. The set of AEAD algorithms that can be chosen to encrypt the data is thus limited to those enabling the retrieval of the authentication tag (e.g., by defining its position in the ciphertext). MPTCPsec proposes three AEAD algorithms that enable to get the tag from the ciphertext (as a reminder, these algorithms are negotiated during the MPTCPsec handshake). The algorithms AEAD_AES_128_GCM and AEAD_AES_256_GCM specified in [21] and the algorithm AEAD_CHACHA20_POLY1305 specified in [22]. The authentication tag of the options is appended to the ciphertext of the data, in the payload. Figure 4.3 represents this segment format (assuming that the AEAD algorithm used, puts the tag at the end of the ciphertext).

We still need to choose an algorithm to produce this authentication tag. One could thing to a HMAC (Section 1.1.4) like the well-known HMAC_SHA256 [30], specially designed for that purpose. But MPTCPsec uses the same AEAD algorithm as the one used to encrypt the data. This choice is motivated in Section 4.3.3. The exact format of the AEAD algorithms inputs and the pseudo-code for encrypting and decrypting segments are given in Section 4.3.1.

local nonce	remote nonce	0x1 (MP_JOIN subtype)	(res.)	B	Address ID

Figure 4.8: Format of the input of the HMAC in the MP_JOIN (in MPTCPsec). The `(res.)` and `B` bits, and the `Address ID` are the ones received by the other host.

4.5.2 MP_JOIN

Since MP_JOIN already uses authentication in its handshake, MPTCPsec does not authenticate the MP_JOIN option. But the `B` bit and the `Address ID` fields must be included in the HMAC to prevent attackers from modifying them. It is also better to include the reserved flags (for future use). Each host includes, in the HMAC that it sends, the `B` bit, the reserved flags and the `Address ID` it has received from the other host.
The keys used for the HMAC are no more exchanged during the first handshake in MPTCPsec. MPTCPsec provides these authentication keys (the same as the ones used for the authentication of other options).

Without other modifications, this gives a (semi-)oracle to the attacker (only a truncated HMAC of a chosen value). It is always better to avoid any kind of oracle (Section 1.1.1) and there is an easy solution to prevent the attacker from controlling the whole input of the HMAC function. Instead of doing the HMAC on the sender's nonce only, we use the concatenation of both nonces: first the local one and second the remote one. We also add the subtype (MP_JOIN) in order to be sure the attacker cannot use these HMAC in another context. Figure 4.8 depicts the HMAC input format. The oracle still exists, but is now really hard to exploit for the attacker.

Note that the HMAC of the server is truncated. It eases the impersonation of the server by the attacker, even if only a brute force attack is possible. Fortunately, this attack can only lead to a DoS attack because even if the MP_JOIN handshake succeeds, the data will be encrypted and unreadable for the attacker. The client will wait for the timeout before retransmitting these data on another subflow. In order to make an armful DoS attack, the attacker should be able to successfully impersonate the server for plenty of MP_JOIN handshakes, which is not trivial. Indeed, supposing a 40 GB/s connection, it would take 35 seconds for the attacker to have 50% chance to send a segment with the good truncated HMAC. Moreover, the attacker must wait for the client to send a SYN packet. In conclusion, the truncation of the HMAC is really not an issue.

4.5.3 ADD_ADDR

MPTCPsec authenticates the ADD_ADDR option.

The ADD_ADDR option cannot be replayed since its authentication tag includes the DSN. A replayed ADD_ADDR will therefore either be out-of-sequence or have a wrong authentication tag.

If the ADD_ADDR was not authenticated, this would lead to a weakness against forgery attacks. But these attacks would only be DoS attacks (and cannot totally block the connection). There are three possible cases:

- The advertised address is one of the other host. The host will send a MP_JOIN and will receive a RST because this address is not available. The loss of performance is negligible since this MP_JOIN is sent on a new and unused subflow.
- The address does not belong to the other host but to the attacker instead. The attacker will

not be able to complete the MP_JOIN handshake, and the subflow will not be established. Again, the loss of performance is negligible since this MP_JOIN is sent on a new and unused subflow.

- The attacker removes the option (e.g., in order to prevent the establishment of subflows on paths not controlled by the attacker). The connection could therefore be tied to one subflow (the starting one).

Note that the ADD_ADDR2, defined in [39], cannot be used because MPTCPsec does not define the keys used for its HMAC. Anyway, the ADD_ADDR2 is no more useful since MPTCPsec already provides authentication of the ADD_ADDR option. It provides it in an even better way than the ADD_ADDR2 because an attacker cannot remove the option without removing all the segments of the data flow, otherwise the authentication tag would become incorrect.

4.5.4 REMOVE_ADDR

MPTCPsec authenticates the REMOVE_ADDR.

If the REMOVE_ADDR was not authenticated, the security of MPTCPsec would not be compromised. Indeed, REMOVE_ADDR is only a notification. There are three possible cases:

- REMOVE_ADDR is invalid (i.e., the address specified has not been previously advertised by an ADD_ADDR and it is not currently used): the segment is ignored
- REMOVE_ADDR is valid and the address is used by a subflow: the host sends a Keep-Alive and it will close the connection only if it receives a RST. If the attacker can send this RST, he can directly send this RST instead of passing by the REMOVE_ADDR.
- REMOVE_ADDR is valid and the address is not used by a sublow: the address is removed from the list of the addresses on which the host can establish new subflows.

This authentication is still useful because it makes harder DoS attacks. Without authentication, an attacker can forge or replay the REMOVE_ADDR in order to force hosts to send lots of Keep-Alive (for more details, see 4.10), or in order to remove the addresses as soon as they are advertised by an ADD_ADDR.

4.5.5 MP_FASTCLOSE

The key used by the MP_FASTCLOSE option is no more available with MPTCPsec (as said previously). Hosts obviously cannot send the keys used for authentication and encryption because the attacker would be able to decrypt all the previously sent messages. MPTCPsec sends instead the HMAC of the remote key, done with the local key. This HMAC must be truncated to 192 bits and put in the MP_FASTCLOSE. There is no more space than 192 bits available in the header because TCP needs 12 bytes for the timestamp option and for the first 4 bytes of the MP_FASTCLOSE. TCP does not need the SACK option since the MP_FASTCLOSE attempts to close the data stream abruptly. These two options are explained in Section 1.2.1.

MPTCPsec does not use SHA-1 (define in [30]) because it is not secure[3]. The HMAC algorithm is negotiated by means of the MPTCPesn sub-options (this document proposes some sub-options and their related HMAC in 4.9).

Since this value is only known by the other host, there is no need for MPTCPsec to authenticate this option.

[3]Google finds a collision and states in `https://sites.google.com/site/itstheshappening/` that SHA-1 is no more secure.

4.5.6 MP_PRIO

MPTCPsec authenticates the MP_PRIO option. Otherwise, an attacker could make a DoS attack by setting all the subflows as back-up paths except one of his choice, in order to reduce the connection to a single subflow. More than that, as he can choose the sole working subflow, he could choose one he is on in order to reduce even more the throughput by dropping most of the segments on this subflow (but not all, otherwise the subflow will be closed and the back-up ones will be used).

4.5.7 MP_FAIL

MPTCPsec should not use this option. It should close the subflow, send a REMOVE_ADDR, or even close the MPTCPsec connection, instead of sending this option. This choice is based on several observations, listed below. The first one concerns the case of a connection with more than one subflow, in which the MP_FAIL option purpose is to synchronize the DSN (i.e., the other host must discard all data following the data sequence number specified). The other observations concern the case of a single-subflow connection, in which the MP_FAIL option advertises a fallback.

- There cannot be false positives since all segments are authenticated. All segments accepted by MPTCPsec, i.e., with a correct authentication tag, have certainly not been modified. There is therefore no need to synchronize the DSN, which is the main purpose of the MP_FAIL option.
- It is not possible to fall back to MPTCP from MPTCPsec due to the fact that the keys in the MP_CAPABLE have been replaced by the MPTCPesn.
- It is not possible to fall back to tcpcrypt (or TLS, according to the scheme negotiated here) because if MPTCP has failed, MPTCPsec could have failed too. Thus, there is no shared secret.
- It is not reasonable to fall back on TCP and to rely on the scheme negotiated over TCP. Indeed, if a host tries to negotiate an encryption scheme with MPTCPesn, it is highly likely that it has not tried to negotiate encryption scheme over TCP (due to the small space available in the header for the options).
- It does not seem reasonable to fall back to an unsecured TCP when a host has tried to negotiate a secure MPTCP connection. It is like giving a lizard to someone who has bought a dragon !

Despite all this, if this option is used, it must be at least protected. It must be authenticated. And so, if this option is received, it must be included in the authentication tag verification (but it should not be applied).

4.6 Fragmentation

So far, MPTCPsec has transformed the MPTCP stream mode into a message mode. But this is less efficient. For example, MPTCP enables to cover several segments with only one DSS. And in case of re-injection of a segment on another subflow, if this subflow has a smaller MTU, the segment is fragmented on several smaller segments. It introduces some difficulties with MPTCPsec. This section tries to make MPTCPsec more compliant with stream mode by enabling fragmentation of the messages.

The first problem arises from the fact that the tags are put in the payload and are covered by the DSN mapping. If a segment is fragmented, we cannot treat each fragment separately as

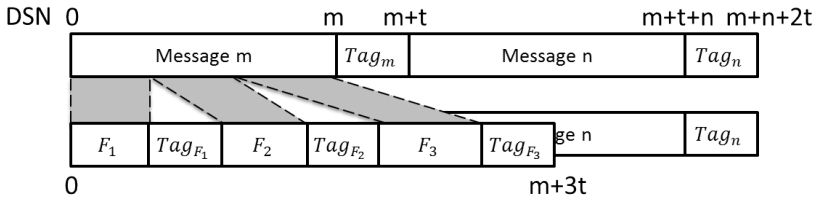


Figure 4.9: DSN overlapping issue when fragments are encrypted and authenticated separately. The encrypted messages m and n are respectively of size m and n bytes, the size of the segment tags are t bytes, F_i is the i^{th} fragment of the encrypted message m and Tag_i is the tags of segment i.

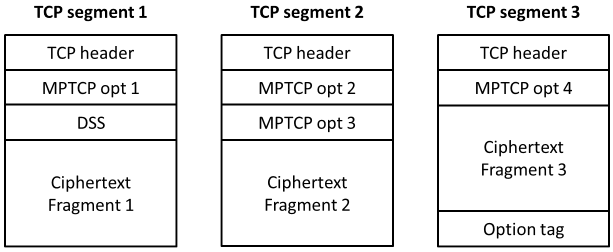


Figure 4.10: Example of ciphertext fragmented in three segments. The option tag is made on the MPTCP options, concatenated as follows (in this order): {opt 1, DSS, opt 2, opt 3, opt 4}

usual segments because the resulting additional tags (two per fragment instead of two for the whole segment) cannot be mapped by the DSN. Figure 4.9 gives an example of two segments mapped by the DSN, and it shows the DSN overlapping when the first segment is fragmented. In other words, we cannot use more tags than the two ones of the initial segment.

MPTCPsec keeps the same ciphertext (composed of the encrypted data and the data tag) and fragment it. Thanks to this, the data are encrypted and authenticated only once in all cases, but data can be decrypted only when all fragments are received. This also requires that each ciphertext is always covered by one and only one DSS option and that a DSS option always covers only one ciphertext, otherwise MPTCPsec could not distinguish ciphertexts from each other. The length of the ciphertext is indeed given by the Data-Level Length field of the DSS option covering it (minus the size of the option tag).

But what for the option tag ? And here comes the second problem because the options of all fragments must be authenticated with only one tag. So, MPTCPsec must remember all the options sent with the fragments until sending the last one. In the last fragment, MPTCPsec puts at the end of the ciphertext the data tag, computed from all the options sent, in the same order than in the fragments (i.e., in the same order than in the segments' headers, with the options of the first fragment before the ones of the second fragment). Due to that solution, each TCP segment can contain only one ciphertext (or a fragment of it), otherwise MPTCPsec could not know to which ciphertext the options of the segment belong (i.e., in which option that MPTCPsec should include them). This means that the re-assembling mechanism of TCP must be disabled. Another issue is that all fragments must be received before applying any of the MPTCP options.

Figure 4.10 gives an example of a ciphertext fragmented in three segments. The MPTCP options i are MPTCP options other than the DSS option. The option tag is made on the MPTCP options, concatenated as follows (in this order): {opt 1, DSS, opt 2, opt 3, opt 4}. The data tag is included somewhere in the ciphertext (depending on the AEAD chosen).

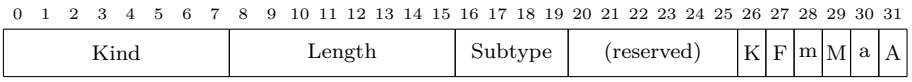


Figure 4.11: Format of the 4 first bytes of the DSS option for the tcpcrypt adaptation. K is the bit for the re-keying mechanism.

4.7 Re-keying mechanism

Tcpcrypt enables hosts to refresh their keys during the connection, as explained in Section 2.2.4. To inform the other host of the re-keying, tcpcrypt uses one bit of the control byte (in the record), the rekey bit. But MPTCPsec does not use records anymore. We can move this bit in the DSS option. Between the `subtype` field and the flags, there are some unused bits. MPTCPsec defines a flag next to the `F` bit: the `K` bit. This is represented in Figure 4.11.

We define the re-keying DSN as the DSN specified by the DSS when the K bit is set. Data sent before a re-keying (i.e., data DSN $<$ re-keying DSN) must always be encrypted and authenticated with the old keys, even if they are reinjected on another subflow. It holds also for the authentication of the options. The tag of the options increases the DSN. It enables to know if the options have been authenticated with old keys (options tag DSN $<$ re-keying DSN) or new keys (options tag DSN $\geqslant$ re-keying DSN). If a DSN is in a DSS with the re-key bit set, in case of retransmission or reinjection, all DSS options specifying this DSN must have the re-key bit set too.

Since MPTCPsec segments can be processed even when they are not in sequence, the mechanism is more complicated than in tcpcrypt. While designing it, under the form of a state machine, we had three goals:

- Data flow is not impacted by the re-keying mechanism, except if hosts want to launch a second re-keying whereas the first is not finished. Indeed, hosts will need to wait for the first one to finish before sending more data.
- Only one bit in the DSS is used. These bits are precious, it is therefore better to minimize the number of them used for the re-keying.
- At most 2 sets of keys (i.e., 4 keys) have to be kept in memory, as well as at most 2 DSNs (32-bits DSNs is enough). It enables to avoid an attacker to make lots of connections and launch lots of re-keying mechanisms in order to fill the memory. Indeed, the remote host cannot flush the old keys of its memory until having received and sent all the data with lower DSNs than the ones of the re-keying.

The state machine we have designed is available in Figure 4.12. The main idea is that, when Host A launches a re-keying (S1→S2), it sends segments with the K bit set (S2 and S3) until being sure that Host B has received at least one of them (S2→S3 or S4→S5). In other words, until it receives an acknowledgment of one of these segments (Data-ACK $\geq$ LD). Then it sends segments with the K bit unset (S3 and S5). In the meantime, when it receives a segment with the K bit set, it knows that Host B has started the rekeying too (S2→S4 or S3→S5). It keeps in memory the corresponding DSN (RD) and updates it if it receives a K bit set with a lower DSN. When it will have received all the segments with a lower DSN (All DSN $<$ RD), it means that Host B does not need the old $B \rightarrow A$ key anymore. Host A can flush its old $A \rightarrow B$ key from its memory if Host B has received all the segments encrypted with it (Data-ACK $\geq$ LD). There is one more condition before Host A can launch again a re-keying. It must have received at least one segment with the K bit unset and with the new keys (K=0 & DSN $>$ RD). Otherwise, if it launches a new re-keying, it could not know if a segment received with the K bit set is an old

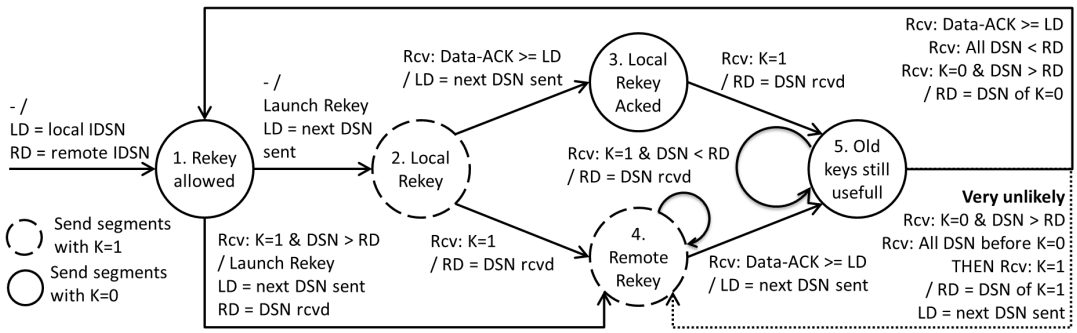


Figure 4.12: State machine of the re-keying mechanism. Arrow labels are constructed as follows: `[events] / [actions]`. All `[events]` must be fulfilled before doing `[actions]`. `LD` (resp. `RD`) is a Local DSN (resp. Remote DSN) to keep in memory. `Rcv: [prop]` is the reception of a segment having the properties `[prop]`.

one or the response of Host B to the new re-keying. Once these three conditions are fulfilled, it can start a new re-keying (S5→S1).

It is the same reasoning for Host B. This comes from the fact that we have designed our state machine to handle simultaneous re-keying, i.e., both hosts want to launch a re-keying at the same time. A host that receives a segment with the K bit set therefore becomes itself a host that wants a re-keying, except that it has already received the segment with the K bit set (S1→S4) while Host A still waits for it.

It is unlikely that the dashed arrow in the figure will be used. Indeed, it can happen if Host B launches a new re-keying whereas Host A still cannot do it. But Host A is ready to do a new re-keying. Indeed, if Host B has fulfilled all the conditions, the sole condition that Host A could have not fulfilled is the reception of an acknowledgment of one of its segments with the K bit unset (Data-ACK $\geq$ LD). But if Host B launches the re-keying (and respects the states machine), Host A can be sure that Host B has received at least one of its segments with the K bit unset. And all its conditions are thus fulfilled.

Note that the re-keying DSN is the DSN of the first segment with the K bit set.

4.8 IDSN and token value

MPTCPesn specifies (Section 3.3) that encryption specifications must define the MPTCP connection tokens and the Initial Data Sequence Numbers (IDSN). This section defines these values.

MPTCPesn, along with TCP-ENO [10], requires encryption schemes to define a 33-bytes (or more) initial session ID. It is pseudo-random (except the first byte) and public. Its first byte is the sub-option of the encryption scheme used. Since tokens and IDSNs are public, MPTCPsec defines them as the remaining bytes of this session ID. In MPTCPsec, this Session ID is defined during the handshake (i.e., the tcpcrypt handshake, and we are sure it does it because TCP-ENO requires encryption scheme to define a Session ID too). MPTCPsec decomposes the session ID as follows (by bits):

- 0-7: sub-option of the encryption scheme used
- 8-63: host A's token
- 64-127: host B's token

- 128-191: IDSN $A \rightarrow B$
- 192-255: IDSN $B \rightarrow A$

With this token derivation, hosts cannot avoid an already used token (unlike MPTCP). Therefore, MPTCPsec has to use 64-bits tokens in order to avoid collisions with high probability.

In the case of a connection resumption, if a new session ID is defined (as in tcpcrypt), MPTCPsec must use it to define the new tokens and IDSNs as above.

4.9 Spec Identifiers

MPTCPsec currently defines 3 spec identifiers that can be negotiated with MPTCPesn. These identifiers can be used to negotiate the use of the MPTCPsec, each one specifying a different key-agreement scheme. More spec identifiers can be easily defined later on. Here are the values common to the three spec identifiers (they are defined in Section 2.2.2):

- **Extract**: HKDF-Extract-SHA256
- **CPRF**: HKDF-Expand-SHA256
- $\boldsymbol{Len_{N_A}}$, the length of the random nonce of host A: 32 bytes
- $\boldsymbol{Len_{N_B}}$, the length of the random nonce of host B: 32 bytes
- $\boldsymbol{K_{len}}$, the length of the key: 32 bytes
- **HMAC**: HMAC-SHA224

The proposed key-agreement schemes (defined in [34]), next to the names of the corresponding spec identifiers, are the following:

- `MPTCPSEC_TCPCRYPT_ECDHE_P256`: ECDHE-P256
- `MPTCPSEC_TCPCRYPT_ECDHE_P521`: ECDHE-P521
- `MPTCPSEC_TCPCRYPT_ECDHE_Curve25519`: ECDHE-Curve25519

4.10 Keep-Alive

Tcpcrypt [12] recommends to change the Keep-Alive by a rekeying mechanism in order to prevent an attacker to extend a connection with spoofed Keep-Alive. It says that "Such an attack might prevent a process with sensitive data from exiting, giving an attacker more time to compromise a host and extract the sensitive data.". In MPTCP (and MPTCPsec), when a host receives a REMOVE_ADDR, it sends a Keep-Alive on the subflow(s) to see if the address(es) is (are) actually unreachable. One could say that MPTCPsec does not need anymore this Keep-Alive since options are sent reliably, but they are not processed in the same order as they are sent. Thus, as Keep-Alive is not a costly operation, MPTCPsec still sends them. But MPTCPsec should not replace these Keep-Alive by a rekeying mechanism. Otherwise, it could reduce the throughput in case of several REMOVE_ADDR sent in a short time. Indeed, there can be at most one rekeying launched at a time (Section 4.7), and hosts should wait until all the rekeyings (except the last) have finished before sending data again.

Concerning regular Keep-Alives (i.e., the ones sent after a fixed time without any fresh data received), MPTCPsec should replace them by a rekeying mechanism. The keys will thus be regularly changed and the attacker will not be able to find them.

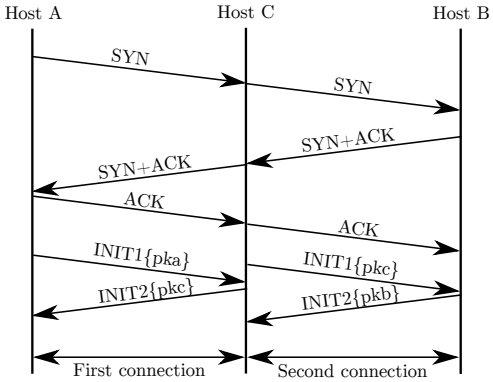


Figure 4.13: Attacks on the initial MPTCPsec handshake, which is made on the initial subflow.

4.11 Security goals achieved

All the security goals have been met and this section proves it. In the first part, we show how the attacks on MPTCP cannot be done anymore on MPTCPsec, and against which model attacker MPTCPsec is secure. In the second part, we present the analyse that we have done with the security tool scyther [40] on the MPTCPesn and MPTCPsec handshake.

4.11.1 MPTCPsec more secure than MPTCP

This section shows how the attacks on MPTCP become harmless with MPTCPsec and concludes by specifying against which attacker model MPTCPsec is strong.

Let us take each attack of Section 4.1 one by one:

1. **Eavesdropping**: once the connection between hosts is established, MPTCPsec is secure against eavesdropping. The cipher cannot be decrypted without the secret key. An attacker will never be able to read data exchanged by hosts whatever he does during the connection.

2. **Identity spoofing**: MPTCPsec ensures the identity of the segments' sender (i.e., data authenticity) by means of authentication tag, once the connection is established. But during the connection establishment, MPTCPsec is not secure against active attackers. Indeed, as MPTCPsec does not use certificates, it is vulnerable to Man-In-The-Middle attacks (explained in Section 1.1.6 and shown in Figure 4.13).

3. **Data truncation**: MPTCPsec is secure against truncation attacks. In other words, either all data sent will eventually reach the destination or the hosts will be aware of the failure. Since all data (and the DSN) sent are authenticated, the sole way to make a truncation attack in MPTCPsec would be to close the connection on behalf of one of the hosts and without raising an error. But we can prove that it is not possible for an attacker, whatever he tries, to close the connection:

   - `RST and FIN` (Section 1.2.3): if an attacker succeeds to close all the TCP subflows (with RST or FIN), the MPTCP connection itself will not be closed. Therefore, each host knows that there can be still pending data (and new subflows will be created).
   - `DATA_FIN` (Section 1.3.4): The DATA_FIN is a flag in the DSS option. MPTCP does not provide any protection against DATA_FIN forging, but MPTCPsec authenticates the DSS. MPTCPsec is thus protected against this kind of attacks.
   - `MP_FASTCLOSE` (Section 1.3.5): in MPTCP, the attacker needs to know the key which is sent in clear (only) during the initial handshake. MPTCP is therefore not totally secure against MP_FASTCLOSE forging. MPTCPsec removes this weakness since it

sends a shared secret instead, never sent before and specially designed for this purpose (Section 4.8).

MPTCPsec is therefore secure against truncation attacks.

4. **Data modification**: In MPTCPsec, data cannot be modified without raising an error since they are authenticated. An attacker cannot even replace some data by another authenticated cipher of a previous segment. Indeed, the data are authenticated with their DSN as associated data.

5. **Data injection**: The attacker cannot authenticate new data, and he cannot re-inject data previously sent for the same reason as above (i.e., data are linked to their DSN).

6. **Segment replay**: Again, options and data are linked to their DSN by means of the authentication tag and therefore, cannot be replayed. Indeed, either they would be out of sequence (if the attacker has kept the same DSN) or the authentication tag would be incorrect (if the attacker has modified the DSN). Even options sent without data have their unique DSN since the authentication tag is put in the payload (Section 4.5).

7. **Option removal**: Options are authenticated and removing one of them leads to an incorrect authentication tag. Even if a segment without data is stopped by an attacker, as the tag is covered by the DSN, the expected receiver will notice that a segment has been lost.

In conclusion, MPTCPsec is secure against any attacker that can freely see, drop, modify and forge any segment on the network (except against the Man-In-The-Middle at the connection establishment) and is strong against a wide range of DoS attacks. Note that if any of the hosts is compromised and/or malicious, the whole connection is compromised. Moreover, a weak pseudo-random generator at either host could compromise MPTCPsec's security.

4.11.2 Scyther tool

Scyther [40] is a tool that enables to verify the security of cryptographic protocols. It performs a formal analysis under the perfect cryptography assumption, i.e., assuming that all cryptographic functions (e.g., hash functions) are perfect. Protocols can be described in the Scyther input language. Claims can be added to the protocol and Scyther will try to find attacks that falsify these claims.

We have used Scyther to analyse the MPTCPesn and MPTCPsec handshakes and the sending of the first data. The code we have made and the results are available in Appendix A. As we expected, the Man-In-The-Middle (MITM) attack is possible and therefore no property is verified (Figure A.1). Indeed, the MITM can do whatever he wants. We have thus modified two lines (in comments in the code) in order to analyse the handshakes without considering the MITM attacks (i.e., we ensure for both hosts that the received public key is the one of the other host). The following properties have been tested:

- **Data secrecy** ensures that data can be read only by both sides of the connection.
- **Data agreement** ensures that data received is the one sent by the other side of the connection.
- **Aliveness** ensures that whenever Host A plays the client role apparently with Host B, Host B has played one of the role (client or server) at least once, but not necessarily at the same time (and both can have played the client role).

- **Weak agreement** is stronger than Aliveness, since it ensures that both hosts have apparently played their role with the other one (but again, not necessarily at the same time).
- **Non-injective agreement** ensures weak agreement and also that both hosts agree on all the values they have exchanged.
- **Non-injective synchronisation** adds to non-injective agreement that the sending and receiving events happen in the right order (i.e., a host cannot receive a message that has not been already sent by the other host).

MPTCPsec, without considering the identity authentication weakness (i.e., MITM attack), ensures[4] all these properties except the last one (Figure A.2). Indeed, an attacker can easily send the handshakes' ACK segments before hosts do it themselves (i.e., preplay attack). These segments (among others) can be easily guessed and forged, and hosts will not notice that it is not their own ACK segments (identical in term of semantic) that have been received. This issue can be armful in the case of a client paying the server for the time of its connection. If the attacker is able to guess the MPTCPesn handshake segments that the client will send (which is not really hard), he can start the MPTCPesn connection long before the client really does. The server will thus charge the client much more than he could expect. Note that the attacker cannot easily guess the first segments of the MPTCPsec handshake since they contain random nonces. The Non-injective synchronisation weakness is therefore armful only on the MPTCPesn handshake.

The results are those we expected and are quite satisfying. The main and nearly the sole weakness is that MPTCPsec does not use certificates.

4.12 Comparison with TLS above MPTCP

This section compares our solution, MPTCPsec, with TLS above MPTCP. It first presents the main achievements of MPTCPsec, that come from the integration of it in MPTCP while TLS runs above MPTCP. Then it compares the handshake of MPTCPsec and TLS.

4.12.1 Main improvements

TLS above MPTCP does not protect the MPTCP connection itself while MPTCPsec does. With TLS, an attacker can replay, forge and remove MPTCP options as he wants because they are not authenticated. Moreover, if he is on all the subflows and knows the MPTCP keys (sent in clear in the MP_CAPABLE), he can close the connection without raising an error at the MPTCP layer (by sending an MP_FASTCLOSE on one subflow and RST segments on all others). TLS detects that the connection has been closed prematurely, but cannot know the source of the failure (i.e., the other host, an attacker or even the local MPTCP configurations) while MPTCPsec can. In fact, with MPTCPsec, it is even impossible the failure comes from an attacker because the value of the MP_FASTCLOSE is never sent in clear (Section 4.5.5). At most, an attacker can close all the subflows, but not the MPTCP connection itself.

With MPTCPsec, an attacker can make DoS attack only on the subflows he controls. In other words, from one subflow, he cannot interfere with another one. With TLS, for instance, an attacker can prevent the use of path he does not control by means of MP_PRIO options (by setting them as back-up subflows, Section 1.3.6).

Another issue with TLS is the handling of an AEAD failure (Section 4.3.2). TLS has to close the whole connection. It is therefore really easy for an attacker to make a DoS attack: he

[4] We have tested with up to 8 runs, and there is no attacks within these bounds.

only needs to change any single bit, in any segment of any subflow to close the whole MPTCP connection. However, MPTCPsec only close the subflow that detects the AEAD failure, so even if the attacker is on all the subflows, he cannot close the MPTCP connection itself.

A last point is that TLS must wait for data to be delivered by MPTCP, in sequence, to decrypt them. It means that if a segment is lost, the host must wait for it before decrypting the other segments (with a higher DSN). This can lead to CPU idle time until the lost segment is finally received, followed by a peak of computation. Indeed, since the segments following the lost one are already received by MPTCP, they will be all delivered by MPTCP to TLS (and thus decrypted by TLS) at the same time. This issue is solved by MPTCPsec because it can decrypt segments as soon as they are received, even if they are not in sequence.

4.12.2 TLS and tcpcrypt handshakes comparison

The main difference between TLS and MPTCPsec handshakes (i.e., TLS and tcpcrypt handshakes) is that the first one provides authentication of the server via certificates (but this is not mandatory) while the second does not. As counterpart, TLS needs the application to be adapted while MPTCPsec operates in the kernel only (but if needed, it can communicate with the application). This results from their respective objectives: simplicity and efficiency for MPTCPsec, high security for TLS.

The MPTCPsec handshake is a two-way handshake (i.e., both init messages) while the one of TLS is a three-way handshake. Indeed, in TLS (version 1.3) the server must wait for the Finished message of the client before beginning to send data while MPTCPsec enables to send data right after the init2 message. The TLS handshake is also quite heavier than the one of MPTCPsec because there is much more data to send and also because it is partially encrypted with keys derived from an ephemeral secret (which brings more connection confidentiality). But TLS has one more drawback compared to MPTCPsec: the ClientKeyShare (Section 2.3.1) provided by the client can be refused by the server. In that case, the server sends a HelloRetryRequest and the client must restart the handshake from scratch, with an appropriate ClientKeyShare. In other words, the three-way handshake sometimes becomes a five-way handshake.

MPTCPsec's handshake is thus less secure but faster than TLS's handshake.

4.13 Future work

This section discusses about the possible improvements of MPTCPsec. The first point is about encrypting or not the MPTCP options. The second point is about the length of the data, which is visible by everyone. Then this section treats the DoS attacks made by modifying the window size advertised on TCP. The SYN flooding attacks ends this section.

4.13.1 MPTCP options encryption

As said in Section 4.5.1, MPTCPsec does not encrypt the MPTCP options. This choice is motivated by simplicity. In order to be able to take a good decision, many performance tests and measures are required in a real network. For each option, we should make loads of connections with and without encryption to be able to compare the behavior of middleboxes in both cases. Indeed, middleboxes could remove the option because they cannot read it or because it seems strange.

Another point of interest: the performance. We could think that the computational cost would be the same with or without encryption since options are already authenticated with AEAD algorithms. But encrypting these options would maybe force to break some optimization

made by the MPTCP implementation. Again, loads of performance tests must be made, with an optimized implementation of the options encryption.

If the behaviour of the middleboxes is not worse with encryption, and if the computational cost increase is acceptable, it would be better to encrypt the options because it would bring more confidentiality. With the ADD_ADDR and REMOVE_ADDR, an attacker is indeed able to associate distinct IP addresses in the network to the same client/server. For instance, a client opening several websites with its browser establishes different MPTCP connections and each of them has different subflows. The attacker would be able to know that all the IP addresses belong to the same client. The other options also carry information about the connection, and it is always better, in term of security, to hide at most any kind of information.

4.13.2 AEAD plaintext input for options authentication

To authenticate the options, MPTCPsec currently uses an AEAD whose plaintext input is the constant 0x0 (Section 4.3.1). The output is a ciphertext of constant length (because the input length is constant too). The whole ciphertext is used as options authentication tag and is put in the payload (Section 4.2.3).

This plaintext input could be used for many purposes. For example, it could be used to solve the above issue, about encrypting the options (Section 4.13.1). Indeed, the options that need to be encrypted could be used as plaintext input instead of being placed in the header. So that they would be in the payload instead of the header and middleboxes could not see them.

But even better than that, we could use this input to somehow expand the header space, or to exchange control information. As shown in the pseudo-code in Figure 4.5, retrieving this value is really easy. We only need to have a way to specify the length of the ciphertext produced (i.e., the length of the options tag), since the input length would change. We could use the DSS's checksum field that is no more useful (Section 4.4.1) for that purpose.

4.13.3 Lengths visible

The DSS option has only been authenticated, not encrypted. The attacker can see the Data-Level Length, the DSN,... He can thus know exactly the size of the encrypted data sent[5]. Cryptographers do not like that ! Depending on the application, the size of the data can give enough information to guess the information exchanged. For example, if the attacker knows that a given segment contains a password, he can easily know the length of this password, which is obviously bad.

TLS1.3 [32] proposes a solution to this issue by letting the application (or the implementation itself) choose a padding. MPTCPsec could do the same, but such a mechanism requires to send the true length of the data (without the padding). The Data-Level Length of the DSS cannot help us since it specifies the total length of the payload, padding included. MPTCPsec should therefore add and encrypt, in the segment, the true length of the data. For the tcpcrypt adaptation of MPTCPsec, this length could be appended at the beginning of the plain text and encrypted with it.

[5]In fact, this is not always true. Most AEAD algorithms use padding in order to make the plaintext length a multiple of the block size. In these cases, the attacker cannot know exactly the size of the application data. But he still can have an estimation, which can be bad too. For instance, a client tries to get confidential data. If he has not access to it, the answer of the server will be short (e.g., "you cannot access these data"). Otherwise, all the data will be sent. The attacker can therefore know if the client has access to these data. Moreover, the ciphertext (without the tag) produced by gcm-aes is of the same length as the plaintext. The size of the data can thus be precisely known.

4.13.4 Window modification

An attacker can modify the header of TCP segments. So, he can modify the window size in order to reduce the amount of data transmitted over a subflow. Some middleboxes also optimize this value, and thus modify it in the header.

MPTCPsec does not need the `checksum` in the DSS option. This space can be used to advertise the window size so that it will be authenticated along with the DSS option. As all the subflows on MPTCP share the same receiver window size (and thus all the subflows advertise the same window value), it seems acceptable to let the MPTCP layer advertise this information. With this solution, the window size advertised by TCP should be ignored by MPTCPsec.

Concerning the TCP Window Scale option, it is more tricky. Indeed, the MPTCPsec handshake is done after the use of this option, which makes impossible to authenticate this option. A way to protect it would be to include this option in the MPTCPesn transcript either directly or by transforming this option in a general suboption of the MPTCPesn option. The first proposition will cause the connection to fail if a middlebox changes this TCP option while the second one is strange in the sense that this option has nothing to do with MPTCPesn, in the MP_CAPABLE option.

4.13.5 SYN flooding

There is currently no solution to tackle SYN flooding issues. And unfortunately, this DoS attack is easier against MPTCPsec than MPTCP because there is more information to keep in memory for each connection establishment (e.g., the MP_CAPABLE of the SYN and SYN-ACK segments). Perhaps MPTCPsec could use a similar system to SYN cookies (see [41]).

4.14 Conclusion

Through this chapter we have designed MPTCPsec, a cryptographic protocol, adapted to MPTCP and negotiable with MPTCPesn, that enables two hosts to securely and efficiently communicate together, even in presence of active attackers that control all the network. Data sent cannot be read by an attacker, nor be truncated, modified or replayed without both hosts noticing it. The sending and receiving of options are more constrained, but it enables to send them reliably and to avoid an attacker to interfere with subflows he is not on. Indeed, they are attached to a DSN (via their tags), they must be acknowledged and they cannot be interpreted before decrypting the (optional) data of their segments. Moreover, an attacker cannot close the MPTCP connection (but he can close all the subflows). Finally, MPTCPsec enables concurrent encryption and decryption, no matter the order (i.e., data can be decrypted in another order than they were encrypted).

Part III

Implementation

Chapter 5

MPTCPesn

This chapter presents the Linux kernel[1] implementation of MPTCPesn (Chapter 3 for the design). The code is available at `https://bitbucket.org/mptcpsecteam/mptcpsec` (branch mptcpsec_v1.0). First, it details the different configurations of MPTCP and MPTCPesn that can influence the result of the negotiation (Section 5.1). Then, it describes the paths taken by the segments of the TCP three-way handshake and identifies where the MPTCPesn negotiation has been inserted (Section 5.2). After that, some suggestions for future development are discussed (Section 5.3). Finally, the methodology used to assess the MPTCPesn's functioning is detailed (Section 5.4).

5.1 Different configurations

There are different points of configuration in MPTCPesn's implementation that can influence the behaviour of the negotiation :

- Enabling MPTCP
- The version of MPTCP : only version 2 can use MPTCPesn
- The list of encryption schemes that can be negotiated
- The security preference which can be set to `MUST_ENCRYPT`, `TRY_ENCRYPT` or `NOT_ENCRYPT`
- The application aware setting (Section 3.2.1)

The first two settings (already introduced by the MPTCP's implementation) can be changed with sysctl[2] variables. These variables are common to all the applications running on the machine. The encryption schemes that are acceptable can be set via another sysctl variable (in the future, a socket option could enable an application to select only a fraction of the encryption schemes allowed on the machine-level). Nevertheless the choice whether to encrypt and authenticate data or not, in the kernel, is application-dependent. For instance, an application already running above TLS would be slowed down if a second encryption was performed inside the kernel. The implementation provides a socket option that can set the security preference of the application. The last setting is needed by the MPTCPesn protocol and can be set via a socket option (by default, it is set to *application not aware*).

The negotiation has three possible outcomes: a successfully chosen encryption scheme (like MPTCPsec), a fallback to unencrypted and unauthenticated MPTCP (or TCP) or a connection

[1] We have modified the branch `mptcp_trunk` (version of 9th February 2016) of linux kernel (version 4.1) implementation of MPTCP 0.90 [42].

[2] see `http://linux.die.net/man/8/sysctl`

	Not aware	Aware	Must be aware
Not aware	v	v	x
Aware	v	v	v
Must be aware	x	v	v

Table 5.1: Compatibility between two hosts' application aware settings

	No encryption	Try encryption	Must do encryption
No encryption	/	Fallback	Failure
Try encryption	Fallback	Encryption	Encryption
Must do encryption	Failure	Encryption	Encryption

Table 5.2: Compatibility between two hosts' security preferences (assuming no incompatibility of application aware settings or of encryption scheme lists). The case where both hosts have the security preference set to "No encryption" cannot happen, because none of them would have started a MPTCPesn negotiation.

failure (Section 3.1.2). The fallback to unencrypted and unauthenticated MPTCP is not implemented yet, thus it currently leads to the connection failure as well.

Table 5.1 shows the compatibility between two hosts' application aware settings. An incompatibility results into a connection failure. It could result into a fallback to regular MPTCP (not implemented yet) if neither of hosts' security preferences are set to MUST_ENCRYPT. Note that incompatible hosts' encryption scheme lists (i.e., no encryption identifier is present in both lists) will cause the same results. The compatibility between hosts' security preferences are shown in Table 5.2.

5.2 Negotiation flow

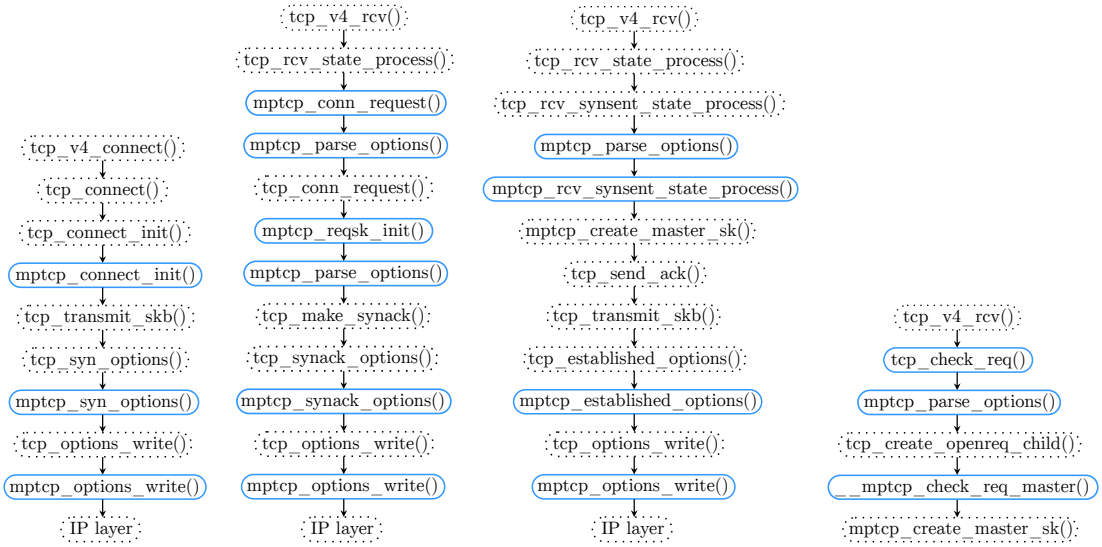
This section introduces how the MPTCP negotiation works in the Linux kernel and presents our modifications to integrate MPTCPesn. It goes through the main functions of the negotiation flow of MPTCPesn (Section 3.1.1), represented in Figure 5.1.

5.2.1 SYN sending

Let us see what happens between the connection request and the sending of the SYN segment (Figure 5.1a). The first function called is `tcp_v4_connect()` (or the IPv6 equivalent), it initiates the connection. Then `tcp_connect()` initializes the SYN segment. The `tcp_connect_init()` function is in charge of initializing the TCP sock (Figure 5.2 shows the position of this kernel equivalent to the socket in the data stream), while the MPTCP equivalent function (`mptcp_connect_init()`) handles the initialization of the MPTCP key and token. This last function has been modified in order to initialize the list of encryption schemes that will be sent.

The next function, `tcp_transmit_skb()`, fills the TCP header of the segment and sends it to the IP layer. It calls `tcp_syn_options()` that computes the header and its TCP options while `mptcp_syn_options()` takes care of the MPTCP options. We have modified the latter in order to initialize the MPTCPesn sub-options[3] in the MP_CAPABLE option, the general one and the encryption schemes identifiers from the previously computed list (that is cut

[3]The variable length sub-options (Section 3.2.3) are not yet implemented since we does not need them for MPTCPsec.



(a) SYN sending (b) SYN reception (c) SYN+ACK reception (d) ACK reception

Figure 5.1: Main functions of the negotiation flow (modified functions are surrounded by a solid line)

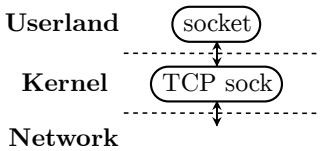


Figure 5.2: The TCP sock is the Kernel structure that stores all the information of the TCP connection. It is between the socket and the network in the data transfer, connection establishment/closing,...

if there is not enough space left in the header). After that, `tcp_transmit_skb()` calls the function `tcp_write_options()`, calling itself `mptcp_write_options()`, in order to actually fill the segment structure (i.e., write in the segment the header and options computed just before). Of course, this last function has also been modified because the `MP_CAPABLE` option format has been changed. Finally, the segment is sent to the IP layer of the kernel.

5.2.2 SYN reception

This section presents the processing of the SYN segment when it arrives on the server (Figure 5.1b). Every reception of a segment is handled in `tcp_v4_rcv()` (or in its IPv6 equivalent). Since the connection is not established yet, it calls `tcp_rcv_state_proccess()`. This function has a different behaviour depending on the state of the TCP sock, which is *LISTENING* in our case, so the next function called is `mptcp_conn_request()`. The latter one must check if a MP_CAPABLE option is present in the SYN segment. It involves a parsing of the options with `mptcp_parse_options()`, modified to enable the recognition of the MPTCPesn MP_CAPABLE option. If it is present, it continues the MPTCP processing, otherwise there is a fallback to TCP. We have modified `mptcp_conn_request()` in order to reject the connection request if it is incompatible with the sock or host settings (Section 5.1 details the settings that can influence this decision).

If the segment passes this check, `tcp_conn_request()` creates a structure called `request sock`. This structure is lighter than a sock and is only converted to a sock upon the ACK reception. It makes SYN flooding attacks[4] less effective. `mptcp_reqsk_init()` is in charge of initializing MPTCP variables (including the MPTCP key and token). It is thus the function where we have initialized the list of encryption schemes that will be sent in the SYN+ACK segment.

The next step is to send the SYN+ACK segment by calling `tcp_make_synack()`. It fills the TCP header by calling similar functions as in the SYN sending. Indeed, the functions `tcp_synack_options()` and `mptcp_synack_options()` compute the header and its options, and then `tcp_write_options()` and `mptcp_write_options()` actually fill the segment structure with these options. Finally, the segment is sent to the IP layer of the kernel.

5.2.3 SYN+ACK reception

This section explains the processing of the SYN+ACK segment (Figure 5.1c). This processing starts in `tcp_v4_rcv()` (or in its IPv6 equivalent), that calls itself `tcp_rcv_state_proccess()` as in the SYN reception (the connection is still not established). But this time, the state of the TCP sock is *SYN_SENT*, so the next function called is `tcp_rcv_synsent_state_process()`. The latter parses the header (involving a call to `mptcp_parse_options()`, modified by us in order to retrieve the MPTCPesn variables), and if a MP_CAPABLE option is observed, it calls the function `mptcp_rcv_synsent_state_process()`. This last function has been modified in order to check that the SYN+ACK segment conforms sock and host settings.

If settings are compatible, `mptcp_create_master_sk()` is called. As its name suggests, this function is responsible for the creation of the master sock of MPTCP. Indeed, in MPTCP, there is not only one sock like in TCP (Figure 5.3) :

- The meta sock (already created) is the interface used by the application's socket. All data sent or received by the application pass through it.

[4] In such attack, the attacker sends many SYN segments with spoofed IP address in order to prevent a server from processing valid connection requests. More details can be found in [41].

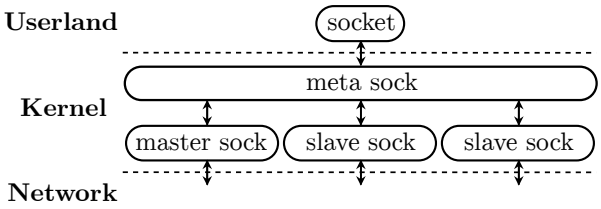


Figure 5.3: `Socks` used in MPTCP

- The master sock (created here) is a TCP sock that is in charge of the first TCP subflow established.
- The slave socks are also standard TCP socks but they handle the subflows created with MP_JOIN option.

This function also creates the control block of MPTCP: `mptcp_cb`.

Once these structures are initialized, `mptcp_rcv_synsent_state_process()` sets the control block's variables related to MPTCPesn, based on the parsing's results obtained with the previous call to `mptcp_parse_options()`.

The last step is to send the ACK segment. This work is started by `tcp_send_ack()` that calls `tcp_transmit_skb()` like in the SYN sending. But this time, the TCP header is computed by `tcp_established_options()` and `mptcp_established_options()`. The latter has been modified for MPTCPesn. Then, `tcp_write_options()` and `mptcp_write_options()` fill the segment structure. Finally, the segment is sent to the IP layer of the kernel.

5.2.4 ACK reception

The last step for the server is to process the ACK segment (Figure 5.1d). Its handling starts in `tcp_v4_rcv()` (or in its IPv6 equivalent). The next function called is `tcp_check_req()`, responsible for the segments processing when the request sock is in the *SYN_RECV* state. This function performs some checks on the received segment. We have thus modified this function in order to check that the `MP_CAPABLE` option on the ACK segment is compatible with MPTCPesn. For this check, this function calls `mptcp_parse_options()`.

After that, the function `tcp_create_openreq_child()` creates a TCP sock (that will become the `meta sock`) from the `request sock`. The call to `__mptcp_check_req_master()` finishes the initialization of the new sock created and calls itself `mptcp_create_master_sk()` in order to create the master sock and the MPTCP control block. `__mptcp_check_req_master()` has been modified in order to initialize MPTCPesn variables inside the `meta sock` and the MPTCP control block.

5.3 Future developments

The implementation of MPTCPesn can be further developed by implementing simultaneous opening case (Section 3.1.1), the fallback to regular MPTCP (Section 3.1.2) and the use of variable length sub-options in the MP_CAPABLE (Section 3.2.3). Moreover, TCP fastopen and TCP SYN cookies should be deactivated when using encryption. Indeed, TCP fastopen enables to send data before the TCP handshake is finished, which is incompatible with the principle of MPTCPesn. And TCP SYN cookies enable a stateless TCP handshake for the server, but the transcript of the handshake is supposed to be kept by the server in order to build the MPTCPesn transcript (Section 3.1.3).

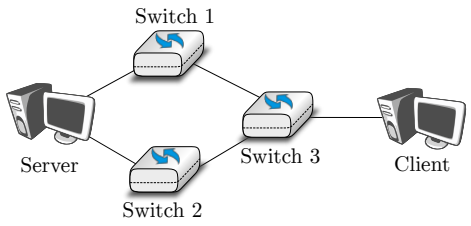


Figure 5.4: Topology used for the tests

5.4 Functionality tests

In this section, we first describe the tools and then the tests[5] used to assess our MPTCPesn implementation functioning.

5.4.1 Test environment

Let us first describe the tools :

- **Mininet** [43] has enabled us to emulate a network. Indeed, it provides a very handy python API that enables to build a customized network topology and to launch scripts on a particular node (host, router, switch,...). We have used a VM image[6] with Mininet already installed.
- **Scapy** [44] has enabled us to forge segments (in particular buggy segments) and also to automatically analyze network traces.

We have used the topology shown in Figure 5.4. This topology is interesting to observe MPTCP because it has two possible paths and because we can capture the traffic either on both paths (`switch 3`) or on only one of them (`switch 1` or `switch 2`).

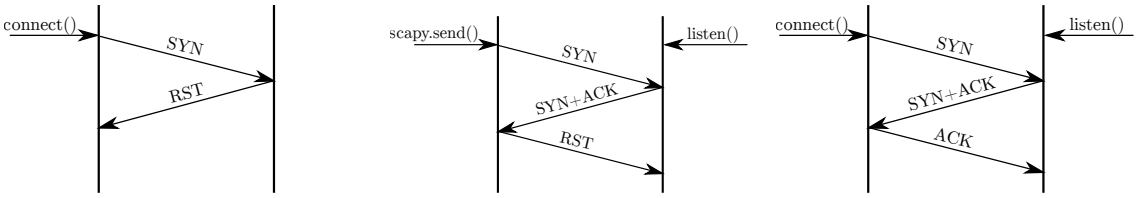
5.4.2 Tests performed

As explained in Section 5.1, there are different configurations in our MPTCPesn's implementation that can influence the behaviour of the negotiation. Hosts' configurations can be compatible or make the connection fail. A failure can appear at any of the three steps of the three-way handshake. For this reason, for each compatible configuration combination, the correct behaviour of the handshake has been checked progressively (Figure 5.5):

- In order to check the SYN sending, we request the client to connect to a non-listening server (Figure 5.5a).
- The check of the SYN+ACK sending is made with the help of Scapy that forges a SYN segment for the server (Figure 5.5b).
- And after these tests, we launch a full connection (Figure 5.5c).

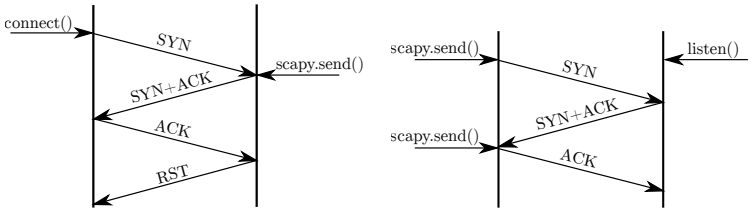
At each step of the process, the network trace is verified automatically with Scapy and we check that the encryption scheme negotiated is the same for both hosts.

We have encountered a problem for testing hosts with different sysctl values. Indeed, the sysctl settings that we use, are common to all processes running on the VM. We have thus used Scapy to emulate a host with a different sysctl configuration during the handshake. Two tests were needed to replace the last one (Figure 5.6). In the first case (Figure 5.6a) Scapy is used to emulate the server and in the second case (Figure 5.6b), the client.



(a) SYN sending (b) SYN+ACK sending (c) ACK sending

Figure 5.5: Step-by-step checks of the correctness of MPTCPesn negotiation



(a) ACK sending (b) ACK parsing

Figure 5.6: Two additional checks to replace the check shown in Figure 5.5c are represented. They are performed with the help of Scapy.

Incompatible hosts' configurations are tested with similar scenarios as in Figure 5.5b, Figure 5.6a and Figure 5.6b, but a RST is the expected answer.

5.5 Conclusion

This chapter has described our MPTCPesn implementation. First, it has detailed the configuration parameters of MPTCPesn. We have then detailed how to modify the MPTCP negotiation flow in order to incorporate MPTCPesn in it. The validation test we have performed have confirmed the correctness of our implementation. Even if some part are still missing (e.g., the fallback), our implementation is ready to negotiate encryption schemes. We can thus, on this basis, begin the implementation of MPTCPsec, starting by ensuring privacy to MPTCP.

[5]The code is made available at `https://bitbucket.org/mptcpsecteam/mptcpsec-functional-tests`
[6]available at `https://github.com/mininet/mininet/wiki/Mininet-VM-Images`

Chapter 6

MPTCPsec - Privacy

This chapter and the next one present the implementation of MPTCPsec (its design is described in Chapter 4). Through these chapters, the modifications made in order to obtain MPTCPsec from MPTCP are described. This chapter focuses on the modifications to achieve privacy. In order to do so, data need to be sent encrypted and authenticated in the network and, upon reception, data must be decrypted and their authentication verified. The code is available at `https://bitbucket.org/mptcpsecteam/mptcpsec` (branch mptcpsec_v1.0).

This chapter begins with the main problem we have encountered for the cryptographic structures initialization (Section 6.1). After that, it describes how the encryption and authentication have been inserted in the data sending (Section 6.2). Then, the insertion of the decryption and authentication verification is described (Section 6.3). Finally, the methodology used to assess the correctness of these modifications is presented (Section 6.4).

6.1 Initialization of cryptographic structures

We have used the GNU/Linux CryptoAPI [45] for the cryptographic algorithms implementations. In this kernel version, among the AEAD algorithms proposed in Section 4.5.1, only GCM-AES128 and GCM-AES256 [21] are available. Chacha20-Poly1305 [22] has been added afterwards to the Linux Kernel.

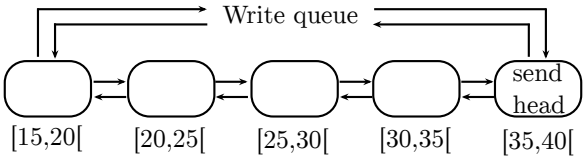
In order to be able to call Crypto API functions, a special structure has to be initialized. This structure is connection-dependent and can thus be created only after the reception of the last segment of the MPTCPsec handshake segment (but also before any data exchange). Nevertheless, in Linux, the TCP segment handling is performed inside an interrupt handler. Since an interrupt handler is meant to be quickly executed, it does not allow to use a scheduler. The problem comes from the fact that the initialization of the crypto structure requires to sleep. The solution is to append the initialization process to a `Work Queues` [46]. The jobs queued in this structure are executed later in process context (i.e., scheduling is allowed).

6.2 Output flow

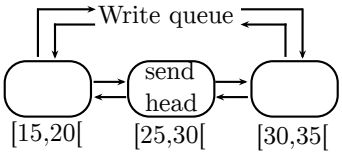
This section describes the mechanisms behind the sending of data in MPTCP and then describes the insertion of the encryption and the authentication in these mechanisms.

6.2.1 MPTCP output flow

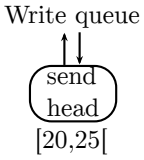
As mentioned previously in Section 5.2.3, MPTCP's implementation relies on several TCP socks (Figure 5.3). As a reminder, there are the meta sock (kind of interface for the application



(a) Write queue of the meta sock



(b) Write queue of the master sock



(c) Write queue of a slave sock

Figure 6.1: Example of state of the MPTCP output flow where the intervals are the Data Sequence Numbers covered by the segments

socket through which all data pass), the master sock (sock of the first subflow) and the slave socks (socks of additional subflows).

When the application wants to send data with its socket, it sends them to the meta sock, on the kernel side. The meta sock creates one or several segments for these data and stores them, in order, in its Write queue (Figure 6.1a). A Write queue is a doubly linked list storing the segments that have not been acknowledged yet. In the figure, you can see the pointer `send_head` that usually points to the first never sent segment. In the meta socket, this pointer has a slightly different meaning since segments are not sent on the network, but a copy of each segment is made and passed to the Write queue of a subflow sock (Figure 6.1b and Figure 6.1c). Therefore `send_head` in the meta sock points to the first segment not yet copied to one subflow queue.

In Figure 6.1, the segment covered by DSN from 30 to 35 has been passed to a subflow sock (the master sock) but this latter has not sent the segment yet. The segment covering bytes 35 to 40 is not yet transferred to a subflow.

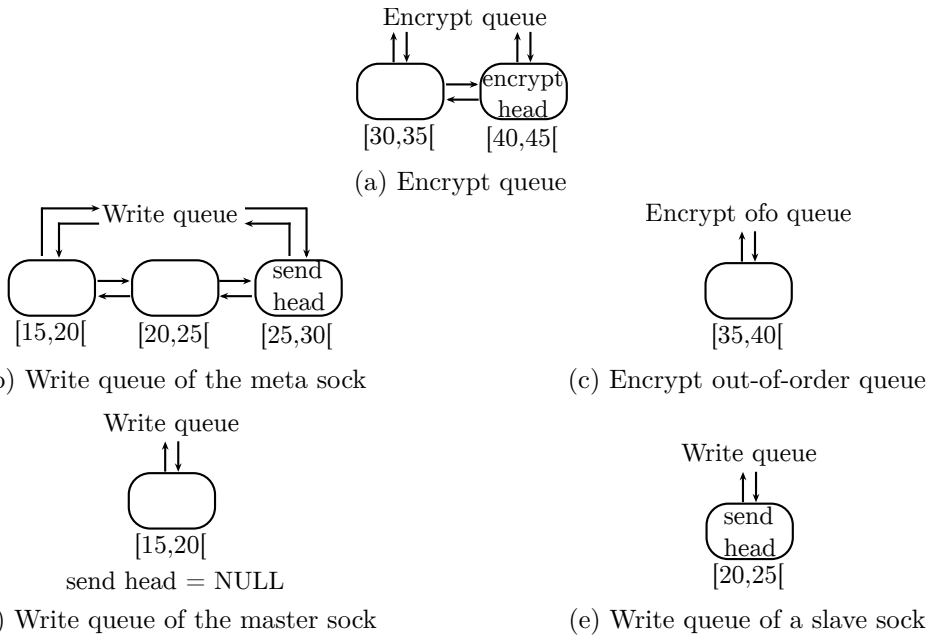
As soon as a segment is acknowledged at the TCP level, it is removed from the subflow Write queue. But it has to be acknowledged on the MPTCP level in order to be removed from the Write queue of meta sock.

6.2.2 Modified MPTCP output flow

The AEAD encryption and authentication operation must be done asynchronously. This is forced by the GNU/Linux CryptoAPI but this is the most efficient choice anyway. It is indeed a very costly operation. Nevertheless, the direct consequence is that we are not guaranteed that the first execution of the AEAD algorithm will be finished before the next ones. It requires the creation of two new lists:

- An `Encrypt queue` (Figure 6.2a) that acts a bit like a Write queue. The difference is that it contains all segments that are not yet encrypted (instead of not yet sent), either because the AEAD algorithm has still not been launched (the first of these segment is pointed by `encrypt_head`) or because the AEAD is still running.
- An `Encrypt out-of-order queue` (Figure 6.2c) stores segments that are encrypted and authenticated while previous ones are not.

When data arrives from the application's socket, segments are created and are put on the `Encrypt queue`. When the segment is sufficiently filled, the negotiated AEAD algorithm is launched on the segment's payload and `encrypt_head` points the next segment.



(a) Encrypt queue

(b) Write queue of the meta sock

(c) Encrypt out-of-order queue

(d) Write queue of the master sock

(e) Write queue of a slave sock

Figure 6.2: Example of state of the modified MPTCP output flow where the intervals are the Data Sequence Numbers covered by the segments

As soon as the authentication and encryption of a segment finishes, it is removed from the `Encrypt queue`. If it is not the first of the `Encrypt queue`, it is put in the `Encrypt out-of-order queue`. Otherwise it is appended at the end of the meta sock's Write queue (Figure 6.2b) and the segments of the `Encrypt out-of-order queue` that become *in sequence* are also moved to the meta sock's Write queue.

The Write queues of the meta sock and subflow socks (Figure 6.2b, Figure 6.2d and Figure 6.2e) are handled exactly like regular MPTCP.

One could argue that the `Encrypt out-of-order queue` is not needed for the correct sending of the segments. Indeed, the segments have already their DSN. But the TCP subflow sequence numbers are chosen depending on the order on the meta sock's Write queue. We think that this reordering could result in a performance drop.

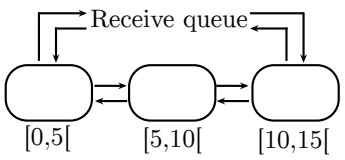
Another one could argue that the data encryption and authentication could be made on the subflow level, just before sending the packet. Indeed, no additional queues would be needed but at each TCP retransmission and reinjection, data should be encrypted and authenticated again, which is really not efficient.

6.3 Input flow

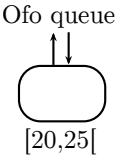
This section describes the mechanisms behind the reception of data in MPTCP and then describes the insertion of the encryption and the authentication in these mechanisms.

6.3.1 MPTCP input flow

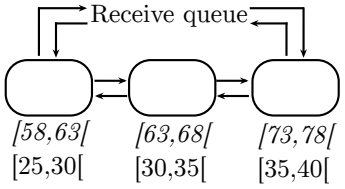
As shown in Figure 5.3 and said previously, MPTCP has several types of socks. The master sock and the slave socks are the first ones to receive data. The segments are reordered in function of their Subflow Sequence Numbers. Then, they are passed to the meta sock where they are reordered according to their DSN. And finally, the data can be retrieved by means of the application's socket.



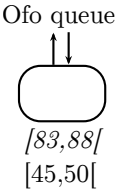
(a) Receive queue of the meta sock



(b) Out-of-order queue of the meta sock



(c) Receive queue of a subflow sock



(d) Out-of-order queue of a subflow sock

Figure 6.3: Example of state of MPTCP input flow's structures. In Figure 6.3c and Figure 6.3d, the intervals in italic contain SSN while the other ones contain DSN.

In these socks, there are two (doubly linked) queues of segments : the Receive queue which stores all segments that are in sequence (w.r.t. DSN for meta sock and w.r.t. SSN for master/slave socks) and the Out-of-order queue which stores the segments inside the receiving window but not in sequence.

in Figure 6.3, you can see an example of a possible state of these queues. When new segments arrive, they are queued in the subflow's Receive queue if they are received in sequence. Otherwise, they are queued in the subflow's Out-of-order queue. After that, the segments inside the subflow's Receive queue are transferred to the meta sock's Receive queue if they are in sequence (w.r.t. DSN). Otherwise, they are put inside the meta sock's Out-of-order queue. Then, data can be read from the meta sock's Receive queue by the application's socket.

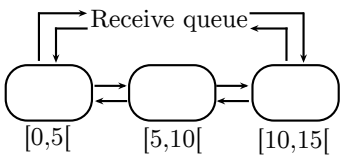
6.3.2 Modified MPTCP input flow

As for the AEAD encryption and authentication, the AEAD decryption and verification must be done asynchronously. Moreover, any segment must be dropped if its verification fails (and its subflow must be closed). For these reasons, two queues must be added in the MPTCP level: the `Decrypt queue` that stores the segments in sequence (w.r.t. DSN) which are not yet decrypted and the `Decrypt out-of-order queue` for those not in sequence.

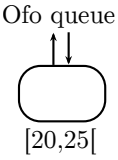
In Figure 6.4, you can see an example of a possible state of the queues of the input flow. When new segments arrive, they are queued in the subflow's Receive queue if they are received in sequence. Otherwise, they are queued in the subflow's Out-of-order queue. After that, the segments inside the subflow's Receive queue are transferred to the `Decrypt queue` if they are not in sequence (w.r.t. DSN) or to the `Decrypt out-of-order queue` otherwise. In both cases, the AEAD decryption and authentication check is run. Nevertheless, these segments cannot be already in the `Decrypt queue`, in the `Decrypt out-of-order queue` or in the meta sock's Out-of-order queue.

When data of a segment of the `Decrypt queue` are decrypted and if it is the segment just after the last one of the meta sock's Receive queue, it can be transferred directly to this last queue. Otherwise, it is put on the meta sock's Out-of-order queue. Then, decrypted data can be read from the meta sock's Receive queue by the application's socket.

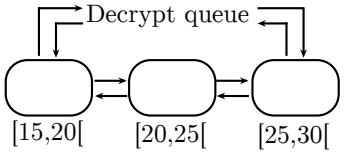
In case of an authentication check failure, the segment is dropped. If it was in the `Decrypt queue`, the next segments of this queue are transferred to the `Decrypt out-of-order queue`



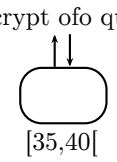
(a) Receive queue of the meta sock



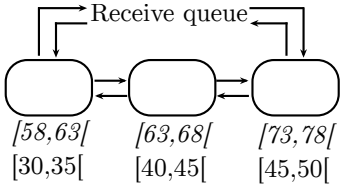
(b) Out-of-order queue of the meta sock



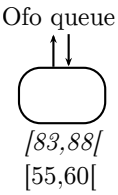
(c) Decrypt queue



(d) Decrypt out-of-order queue



(e) Receive queue of a subflow sock



(f) Out-of-order queue of a subflow sock

Figure 6.4: Example of state of the modified MPTCP input flow's structures. In Figure 6.4e and Figure 6.4f, the intervals in italic contain SSN while the other ones contain DSN.

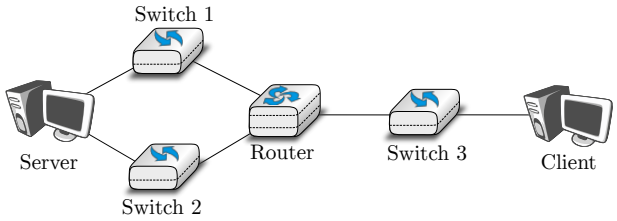


Figure 6.5: Topology used for the tests.

since they are not in sequence anymore.

6.4 Functionality tests

We have taken almost the same topology and tools as for MPTCPesn's tests (Section 5.4.1). A router has been added in the middle of the network and can thus play the role of attacker. The Figure 6.5 shows the modified topology. Checks on this part are based on basic file transfer (of different sizes). Data captured inside the network trace must be encrypted and the file at the output must be equal to the initial one.

Moreover, in order to have more realistic networks, Mininet's emulated links can be configured to emulate delays or losses. This is useful to check that the re-transmissions and re-injections of TCP and MPTCP are not broken by our modifications.

The last thing to check is the behaviour of a host when it receives a segment with an incorrect authentication tag on one of its subflows: it should close this subflow but it should still succeed in transferring the end of the file (Section 4.3.2).

6.5 Conclusion

Through this chapter, we have seen how MPTCP could achieve privacy by inserting AEAD algorithm in the data sending and reception. We have encountered the problem of the GNU/Linux CryptoAPI [45] that has asynchronous operations, and we have solved it with Work Queues and additional segments queues. The implemented solution has been detailed. The validation tests performed have validated our implementation. The following chapter, Chapter 7, details the modifications needed in order to achieve Data Integrity and resistance to some DoS attacks.

Chapter 7

MPTCPsec - Data integrity and DoS resistance

This chapter presents the required modifications in order to achieve data integrity and DoS resistance, starting from a MPTCP version which has already achieved privacy (the one of the previous chapter). It presents both properties here because the implementation choices made for data integrity have a direct impact on the DoS resistance's implementation. The code is available at `https://bitbucket.org/mptcpsecteam/mptcpsec` (branch mptcpsec_v1.0). As an estimation of our complete code length, the diff given by git has a length of 5000 lines.

In order to achieve integrity, data cannot be reordered and they cannot be truncated without the application knowing about the truncation. For this, DSS and MP_FASTCLOSE options must be authenticated (their design is presented respectively in Section 4.4 and in Section 4.5.5).

In order to achieve DoS resistance, we have to protect the other MPTCP options: ADD_ADDR, REMOVE_ADDR, MP_PRIO and MP_JOIN. For instance, if the ADD_ADDR option was not protected, an attacker could prevent the addition of uncontrolled subflows.

As explained in the design part, DSS, ADD_ADDR, REMOVE_ADDR and MP_PRIO options are protected with the same tag. Thus, the choice of the place where authentication is performed, must take all of them into account. MP_JOIN and MP_FASTCLOSE options are not modified in our implementation.

First, this chapter details in Section 7.1 how the authentication is inserted in the sending of data for all the options (except for the MP_JOIN and MP_FASTCLOSE). Then, the insertion of the authentication check for these options is described in Section 7.2. Finally, the methodology used to assess the correctness of these modifications is presented in Section 7.3.

7.1 Output flow

This section discusses the authentication of the DSS, ADD_ADDR, REMOVE_ADDR, MP_PRIO options. These MPTCP options are all written in the segment just before the sending to the IP layer of the subflow. This makes sense to authenticate them (with the the GNU/Linux CryptoAPI) at that moment, after their encoding in the segment. Moreover, this strategy has the advantage to be compliant with the asynchronous behaviour of the cryptography API. Indeed, sending a segment with SSN 2 before the one with SSN 1 is not a problem for the receiver that can reorder them quickly.

In case of re-transmission or re-injection, the semantic of the segment's options must not be changed and options must not be removed as stated in Section 4.2.3. We have not implemented it but we have a suggestion to do it. We can add a pointer inside the TCP control block of `socket buffer` (the structure representing a segment). This pointer points to a `mptcp_options`

structure that represents the semantic of the options. Note that the space inside this control block is limited but there is just enough room for a pointer. As presented in Section 6.2, in MPTCP (and also in MPTCPsec), a segment can be present simultaneously on the meta sock's Write queue and on some subflow socks' Write queue. But the `mptcp_options` structure must be freed only when there is no more `socket buffer` that points to it. This problem is solved by putting a ref counter in the structure and to free it only when this ref counter reaches 0. The same kind of strategy is used for freeing socks.

Another issue to tackle in this part is the sending of a pure ACK segment (discussed in Section 4.4.2). The options tag of a pure ACK segment counts in the TCP subflow sequence numbers. Therefore, such a segment has to be queued on the subflow sock's Write queue but not on the meta sock's Write queue and to be transmitted in sequence. Some performance issues may result from this basic strategy. In order to speed-up the acknowledgement, we can send directly a TCP ACK segment (without MPTCP Data-ACK) and try to change the Data-ACK of an unsent segment[1]. If no unsent segment is available, then and only then, we create this pure ACK segment.

Regular MPTCP may choose to include the options ADD_ADDR, REMOVE_ADDR and/or MP_PRIO in ACK segments. But, as mentioned in Section 4.2.3, for MPTCPsec, the option tag must be counted in the DSNs. This means that it is sent after all the segments which are not yet acknowledged. In order to speed-up the sending of options, we can try to register the options we want to sent to the `mptcp_options` structure of an already created segment. We take the first segment that has not been copied to some subflow sock's Write queue yet. If such a segment does not exist, we create a new segment and queue it to the meta sock's Write queue.

At the moment, we are forcing one DSS by segment (in the implementation). This limitation can be corrected in the future by the implementation of the fragmentation (Section 4.6).

7.2 Input flow

Section 4.2.4 of MPTCPsec design requires to interpret the options after the option check and data decryption. Thus, our implementation launches the option authentication check before the data decryption. Then, the interpretation of the options is made (except for the Data FIN field of the DSS that can only be interpreted in sequence w.r.t. DSN). For the moment, the interpretation of the Data FIN field of the DSS was moved but not those of the other options.

The second case of authentication check is the pure ACK segment (i.e., without data and without other MPTCP options than the DSS, as explained in Section 4.4.2). These segments should be fast and they should therefore be checked and interpreted upon reception. Before checking the tag (costly operation), the host should check that this pure ACK segment acknowledges new data (cheap operation). And if it is the case, only at that time, it should check the tag and then apply the Data ACK (if the tag is correct). The tag of a replayed pure ACK segment is therefore not verified since the segment does not bring any new information.

7.3 Functionality tests

We have taken the same topology and tools as for functionality tests made in the previous chapter (topology is shown in Figure 6.5). Checks on this part are based (as for Privacy part's tests) on basic file transfer (of different sizes). Tests for option authentication have been tested

[1]The segment must be unsent, otherwise the same nonce (based on the DSN) will be used for two different AEAD operations and this can lead to security issues.

by the success of the file transfer but also by the correct reaction to a forged MPTCP option (i.e., the subflow on which the wrong segment was sent has been closed). Some delay and losses were also introduced in order to check the correct retransmissions and reinjections.

7.4 Conclusion

This chapter has described the implementation of data integrity and DoS resistance. We have actually implemented most of the needed modifications. For the other ones, suggestions have been made, based on our experience about MPTCP implementation. Our implementation correctness has been confirmed with the validation tests.

Our implementation still needs the shared secret derivation and fragmentation to fit the MPTCPsec specifications (Chapter 4). By lack of time, we have preferred to focus on performance evaluation to validate our implementation. Chapter 8 presents the results.

Chapter 8

Performance measures

The previous chapters of this part have presented the implementation of MPTCPesn and MPTCPsec. This chapter presents tests that measure the performance of our implementation by comparing it to MPTCP and to the OpenSSL implementation [38] of TLS 1.2 [8] running above MPTCP[1]. Section 8.1 compares the throughput of these implementations. Then, an estimation of the processing time is made in Section 8.2.

8.1 Throughput measures

This section describes the first experiment. The topology used is shown in Figure 8.1. Table B.1 and Table B.2 describe the hardware and the configurations of, respectively, the machines and the interfaces.

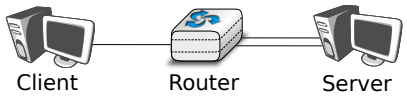


Figure 8.1: Topology used for the experiments

Figure 8.2 shows what experiment has been realized. The client application sends a given amount of data (from 100MB to 1GB) from memory (not from disk) to the server application by blocks of 4096 bytes that stores them in memory. There is no interruption of the data transfer. Each time that 4096 bytes is sent (resp. received), a time measure is taken by the client application (resp. server application).

The experiments has been made on applications using OpenSSL implementation of TLS 1.2 with MPTCP enabled, on applications activating MPTCPsec (by using a socket option) and on applications with only MPTCP. Several transfer sizes were tested and experiments were repeated 15 times (the mean of these measures was taken). Figure 8.3 shows the throughput with different transfer sizes. Only data transfer is considered, the TCP handshake and the shared secret negotiation are not taken into account.

This graph shows that the throughput is independent of the transfer size in the three cases (this is logic for big size of transferred data). The fluctuations observed are probably due to other traffic on the network: other hosts are connected to the router. The three protocols have a throughput that is near the limit of 100Mb/s. MPTCP achieves the best/highest throughput. Since it is the only one out of the three protocols that does not encrypt data, this makes sense.

[1]The code that we have written for the experiments is available at `https://bitbucket.org/mptcpsecteam/mptcpsec-performance-tests`

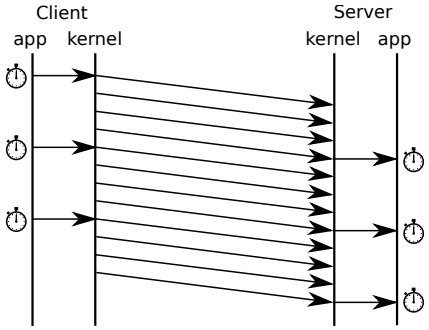


Figure 8.2: This figure describes the type of data transfer made and what time measure we consider.

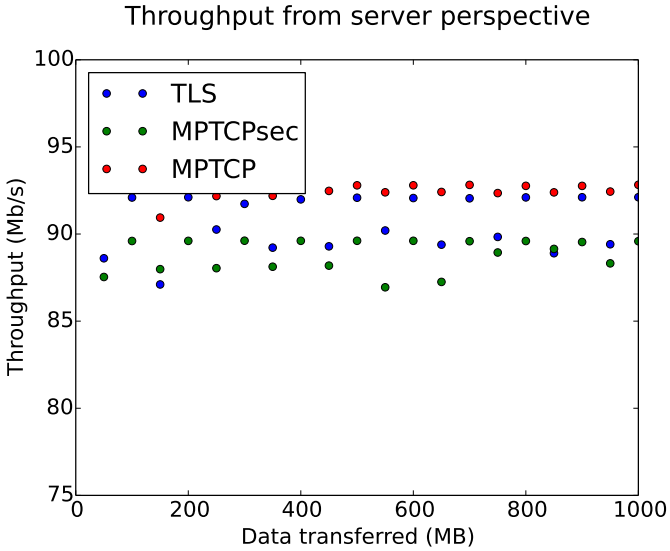


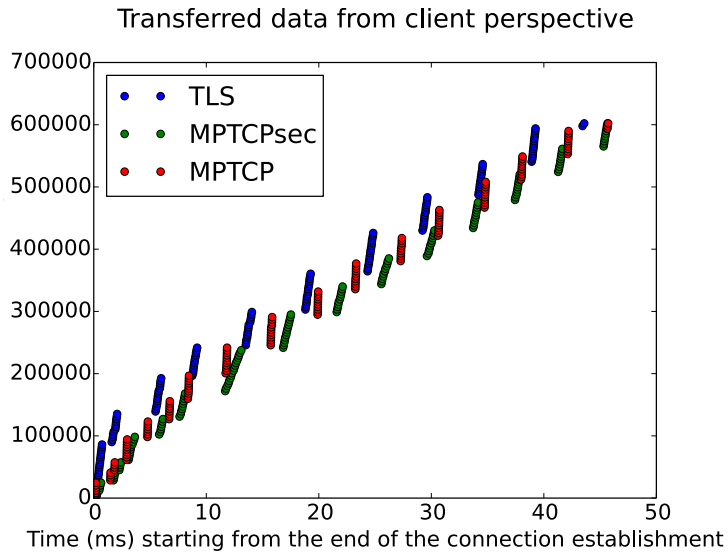
Figure 8.3: Comparison of the throughput of MPTCP, TLS, MPTCPsec implementations: measures are taken by the server's application.

TLS and MPTCPsec have almost the same throughput which is still a great achievement for a first implementation. Nevertheless, MPTCPsec throughput is a bit lower.

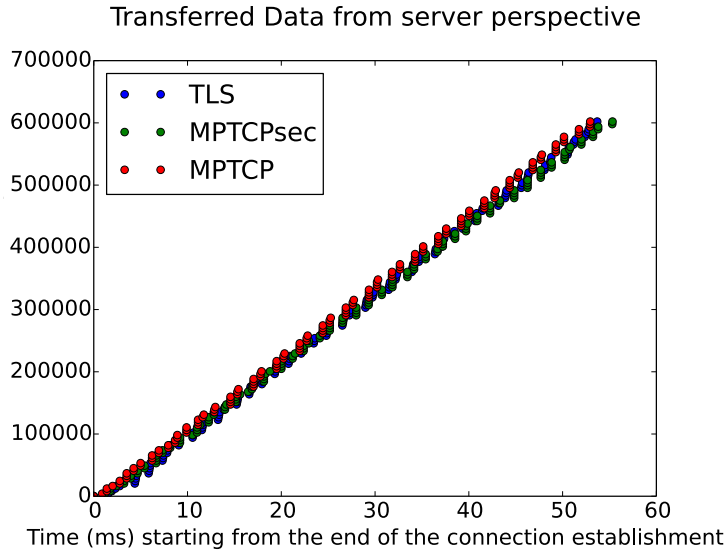
Figure 8.4a and Figure 8.4b are issued from the same experiment with a transfer of 600kB. In these graphs, this is not the throughput that is shown but the data transferred in function of time elapsed. Figure 8.4a, we can observe stairs[2]. As stated previously, the client application took a time measure each time that 4096 bytes were sent but these bytes are perhaps not yet received by the server. Thus, according to this graph, the client application manages to send several times 4096 bytes until it is blocked for a while before the sending can resume. Indeed, MPTCP, like TCP, stops sending beyond a given amount of unacknowledged data. This unacknowledged data is stored inside the sending buffer. The buffer can only be emptied by acknowledgments. Therefore, the stop of the sending is due to the sending buffer limit while its resumption is linked to the reception of an acknowledgment from the server. It seems that OpenSSL implementation of TLS has selected a larger sending buffer size which explains that data, in TLS, leaves the application faster than in MPTCPsec or MPTCP. Nevertheless, on the server side (Figure 8.4b), the MPTCP data transfer is the first to end. Then, TLS data transfer ends, followed closely by MPTCPsec.

The difference between TLS and MPTCPsec is probably due to optimizations made by OpenSSL that we do not apply in MPTCPsec. For future work, it could be interesting to look at

[2]Actually these are not perfect stairs, they look more like Hocaille stairs.



(a) Data sent



(b) Data received

Figure 8.4: Data sent/received in function of time elapsed

the code of OpenSSL in order to find which optimizations can be adapted to MPTCPsec.

8.2 Processing time estimation

This section describes the second experiment. The topology used is shown in Figure 8.1. The hardware and the configurations of the machines and the interfaces are the same as previous section and they are described in Table B.1 and Table B.2.

Figure 8.5 shows what experiment has been realized. The point of this experiment is to look at the processing time of data without taking into account the time elapsed on the link or by being blocked while waiting for acknowledgments. The client application sends a given amount of data (from memory), below the receiving window (thus, from 1kB to 26kB), to the server application in one block (that stores it in memory). Once the server application has received all the data, it sends the same amount of data to the client application. A network capture of the

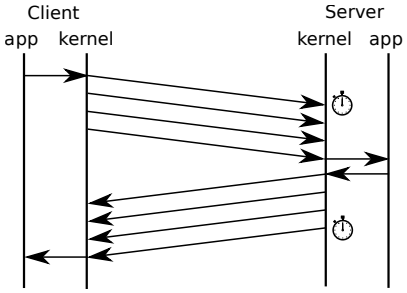


Figure 8.5: This figure describes the type of data transfer made and what time measure we consider.

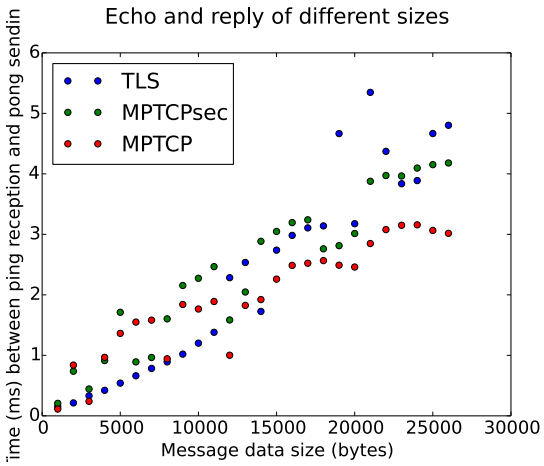


Figure 8.6: Time elapsed between the ping reception and pong sending (8.5 shows the)

transfer is made by the server with tcpdump [47] that stores the capture inside a tmpFS [48] partition. A tmpFS partition is mounted on RAM and this makes file operations performed by tcpdump much faster. The processing time has been estimated by looking at the time elapsed between the reception of the first segment of application data sent by the client and the sending of the last segment of application data sent by the server.

The experiments were made on applications using OpenSSL implementation of TLS 1.2 with MPTCP enabled, on applications activating MPTCPsec (by using a socket option) and on applications with only MPTCP. Several transfer sizes have been tested, experiments were repeated 25 times and the median was selected for each transfer size. The median has been chosen instead of the mean in order to limit the influence of the outliers. The results are shown in Figure 8.6. In this graph, we can see that MPTCP, TLS and MPTCPsec have very closed results. A surprising result is that TLS performs slightly better than MPTCP with small transfer sizes. This could be due to some optimizations of the MTPCP connection made by OpenSSL implementation. MPTCPsec performs as good as MPTCP for small transfer sizes. AEAD operations, run on small data, are negligible compared to the data handling while this difference become less negligible for bigger data sizes. For biggest data sizes in this figure, TLS is comparable and sometimes worse than MPTCPsec.

8.3 Conclusion

Through this chapter, we have performed tests that measure the performance of our implementation by comparing it to MPTCP and to the OpenSSL implementation [38] of TLS 1.2 [8]

running above MPTCP. We have first presented the throughput achieved by our MPTCPsec implementation and we have compared it to the one of OpenSSL implementation (TCP handshake and shared secret negotiation excluded). We have shown tiny differences of throughput between this first implementation of MPTCPsec and the already optimized OpenSSL implementation. Then we have tried to take a time measurement that is dependent of processing time of the host and not of network time. In this case, we have also observed that MPTCPsec measures are very closed to TLS ones.

Conclusion

In this document, we have first laid the foundations, i.e., TCP, MPTCP, the cryptography bases, TCP-ENO, tcpcrypt and TLS. From this, we have designed MPTCP Encryption Scheme Negotiation (MPTCPesn), a mechanism that enables easier negotiation of encryption schemes over MPTCP. It is for MPTCP what TCP-ENO represent for TCP. Designed to be modular and easy to update, it offers a good basis for the development of cryptographic protocols for MPTCP. Indeed, no protocol exists to secure MPTCP connections, except TLS that can be run above them. We have thus designed MPTCPsec, the first cryptographic protocol specially designed for MPTCP (and even integrated in it), negotiable with MPTCPesn. MPTCPsec is application independent, fast and it takes advantage of the MPTCP's properties (e.g., having several subflows). MPTCPsec provides the same security properties than TLS, except the identity authentication (i.e., certificates). But it is resistant to many more DoS attacks than TLS.

MPTCPsec brings three main improvements, compared to simply running TLS above MPTCP (or any other protocol that is not integrated in MPTCP). Firstly, the whole connection does not need to be closed when an attack is detected on a subflow (i.e., authentication failed). Only this subflow is closed, and another one can be opened to replace it. It is a great improvement since an attacker can no more close the whole connection by modifying only one byte of any segment in any subflow. Furthermore, and this is the second improvement, an attacker is unable to interfere with a subflow without being on the path used by this subflow, even if he controls other subflows. He can neither prevent the establishment of subflows he does not control. This is due to the authentication of the MPTCP options. In fact, the most he can do is to close the subflows he is on, or to drop some or all the segments of these subflows (but they will be re-injected on other subflows). The third improvement is that segments can be decrypted as soon as they are received, even if a segment with lower DSN is not already received.

Once the design has been finished, we have implemented the main parts of our protocol, directly in the Linux kernel, in order to properly incorporate it in the MPTCP implementation. First of all to prove to ourselves that our design makes sense and then to be able to compare our solution with TLS above MPTCP with some performance tests. Both objectives are fulfilled and the results are very satisfying. Especially as our implementation is not fully optimized, contrary to TLS implementation (OpenSSL) that has been optimized and improved for several years now. Our most important achievements are that MPTCPsec is a little faster in term of data processing time and that the data flow continues even if an attacker is on one subflow and tries to modify segments (while TLS connection is closed). The results also show that even if MPTCPsec has a little smaller throughput than TLS, it remains very competitive.

Imagine what MPTCPsec could be after two or three years of work in order to improve it and to optimize its implementation, while after less than a year it is already competitive with TLS, in term of security as much as performance, and while it already overtakes TLS in term of resistance against DoS attacks. Exchange of certificates could be added to overcome the main weakness of MPTCPsec: the Man-In-The-Middle. We hope our work has convinced you that MPTCP is a really strong foundation for cryptographic protocol, mainly because if the connection encounter an attacker on its path, it can get around by establishing new subflows.

Bibliography

[1] J. Postel, "Transmission control protocol," STD 7, RFC Editor, September 1981. http://www.rfc-editor.org/rfc/rfc793.txt.

[2] F. Gont, "Icmp attacks against tcp," RFC 5927, RFC Editor, July 2010. http://www.rfc-editor.org/rfc/rfc5927.txt.

[3] A. Ramaiah, R. Stewart, and M. Dalal, "Improving tcp's robustness to blind in-window attacks," RFC 5961, RFC Editor, August 2010.

[4] A. Heffernan, "Protection of bgp sessions via the tcp md5 signature option," RFC 2385, RFC Editor, August 1998.

[5] J. Touch, A. Mankin, and R. Bonica, "The tcp authentication option," RFC 5925, RFC Editor, June 2010.

[6] G. Lebovitz and E. Rescorla, "Cryptographic algorithms for the tcp authentication option (tcp-ao)," RFC 5926, RFC Editor, June 2010.

[7] A. Bittau, D. Boneh, D. Giffin, M. Hamburg, M. Handley, D. Mazieres, and Q. Slack, "Cryptographic protection of tcp streams (tcpcrypt)," Internet-Draft draft-bittau-tcpinc-tcpcrypt-03, IETF Secretariat, July 2015. http://www.ietf.org/internet-drafts/draft-bittau-tcpinc-tcpcrypt-03.txt.

[8] T. Dierks and E. Rescorla, "The transport layer security (tls) protocol version 1.2," RFC 5246, RFC Editor, August 2008. http://www.rfc-editor.org/rfc/rfc5246.txt.

[9] Y. Sheffer, R. Holz, and P. Saint-Andre, "Summarizing current attacks on tls and dtls," Internet-Draft draft-sheffer-uta-tls-attacks-00, IETF Secretariat, February 2014. http://www.ietf.org/internet-drafts/draft-sheffer-uta-tls-attacks-00.txt.

[10] A. Bittau, D. Boneh, D. Giffin, M. Handley, D. Mazieres, and E. Smith, "Tcp-eno: Encryption negotiation option," Internet-Draft draft-ietf-tcpinc-tcpeno-01, IETF Secretariat, February 2016. http://www.ietf.org/internet-drafts/draft-ietf-tcpinc-tcpeno-01.txt.

[11] E. Rescorla, "Using tls to protect tcp streams," Internet-Draft draft-ietf-tcpinc-use-tls-01, IETF Secretariat, May 2016. http://www.ietf.org/internet-drafts/draft-ietf-tcpinc-use-tls-01.txt.

[12] A. Bittau, D. Boneh, D. Giffin, M. Hamburg, M. Handley, D. Mazieres, Q. Slack, and E. Smith, "Cryptographic protection of tcp streams (tcpcrypt)," Internet-Draft draft-ietf-tcpinc-tcpcrypt-01, IETF Secretariat, February 2016. http://www.ietf.org/internet-drafts/draft-ietf-tcpinc-tcpcrypt-01.txt.

[13] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "Tcp extensions for multipath operation with multiple addresses," RFC 6824, RFC Editor, January 2013. http://www.rfc-editor.org/rfc/rfc6824.txt.

[14] D. Boneh and V. Shoup, "A graduate course in applied cryptography (version 0.2)," August 2015.

[15] A. Kerckhoffs, "La cryptographie militaire. journal des sciences militaries," *IX (38)*, vol. 5, 1883.

[16] E. Rescorla, "Diffie-hellman key agreement method," RFC 2631, RFC Editor, June 1999. http://www.rfc-editor.org/rfc/rfc2631.txt.

[17] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J.-K. Tsay, "Efficient padding oracle attacks on cryptographic hardware," in *Advances in Cryptology–CRYPTO 2012*, pp. 608–625, Springer, 2012.

[18] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1," in *Advances in Cryptology—CRYPTO'98*, pp. 1–12, Springer, 1998.

[19] H. Krawczyk, M. Bellare, and R. Canetti, "Hmac: Keyed-hashing for message authentication," RFC 2104, RFC Editor, February 1997. http://www.rfc-editor.org/rfc/rfc2104.txt.

[20] D. McGrew, "An interface and algorithms for authenticated encryption," RFC 5116, RFC Editor, January 2008. http://www.rfc-editor.org/rfc/rfc5116.txt.

[21] K. Igoe and J. Solinas, "Aes galois counter mode for the secure shell transport layer protocol," RFC 5647, RFC Editor, August 2009. http://www.rfc-editor.org/rfc/rfc5647.txt.

[22] Y. Nir and A. Langley, "Chacha20 and poly1305 for ietf protocols," RFC 7539, RFC Editor, May 2015. http://www.rfc-editor.org/rfc/rfc7539.txt.

[23] J. Postel, "User datagram protocol," STD 6, RFC Editor, August 1980. http://www.rfc-editor.org/rfc/rfc768.txt.

[24] M. Allman, V. Paxson, and E. Blanton, "Tcp congestion control," RFC 5681, RFC Editor, September 2009. http://www.rfc-editor.org/rfc/rfc5681.txt.

[25] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "Tcp selective acknowledgment options," RFC 2018, RFC Editor, October 1996.

[26] V. Jacobson, B. Braden, and D. Borman, "Tcp extensions for high performance," RFC 1323, RFC Editor, May 1992. http://www.rfc-editor.org/rfc/rfc1323.txt.

[27] J. Postel, "Tcp maximum segment size and related topics," RFC 879, RFC Editor, November 1983. http://www.rfc-editor.org/rfc/rfc879.txt.

[28] J. Mogul and S. Deering, "Path mtu discovery," RFC 1191, RFC Editor, November 1990. http://www.rfc-editor.org/rfc/rfc1191.txt.

[29] M. Handley and E. R. and, "Internet denial-of-service considerations," RFC 4732, RFC Editor, December 2006.

[30] D. Eastlake and T. Hansen, "Us secure hash algorithms (sha and sha-based hmac and hkdf)," RFC 6234, RFC Editor, May 2011. http://www.rfc-editor.org/rfc/rfc6234.txt.

[31] K. B. Egevang and P. Francis, "The ip network address translator (nat)," RFC 1631, RFC Editor, May 1994. http://www.rfc-editor.org/rfc/rfc1631.txt.

[32] E. Rescorla, "The transport layer security (tls) protocol version 1.3," Internet-Draft draft-ietf-tls-tls13-12, IETF Secretariat, March 2016. http://www.ietf.org/internet-drafts/draft-ietf-tls-tls13-12.txt.

[33] J. Touch and W. Eddy, "Tcp extended data offset option," Internet-Draft draft-ietf-tcpm-tcp-edo-05, IETF Secretariat, April 2016. `http://www.ietf.org/internet-drafts/draft-ietf-tcpm-tcp-edo-05.txt`.

[34] A. Langley, M. Hamburg, and S. Turner, "Elliptic curves for security," RFC 7748, RFC Editor, January 2016.

[35] H. Krawczyk and P. Eronen, "Hmac-based extract-and-expand key derivation function (hkdf)," RFC 5869, RFC Editor, May 2010. `http://www.rfc-editor.org/rfc/rfc5869.txt`.

[36] M. Bagnulo, "Threat analysis for tcp extensions for multipath operation with multiple addresses," RFC 6181, RFC Editor, March 2011.

[37] T. Taubert, "A fast, constant-time aead for tls : Chacha20/poly1305 cipher suites in firefox 47." `https://timtaubert.de/blog/2016/04/a-fast-constant-time-aead-for-tls/`.

[38] I. Ristic, "Openssl cookbook," *Feisty Duck*, 2013.

[39] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, and C. Paasch, "Tcp extensions for multipath operation with multiple addresses," Internet-Draft draft-ietf-mptcp-rfc6824bis-05, IETF Secretariat, January 2016. `http://www.ietf.org/internet-drafts/draft-ietf-mptcp-rfc6824bis-05.txt`.

[40] C. J. Cremers, "The scyther tool: Verification, falsification, and analysis of security protocols," in *Computer aided verification*, pp. 414–418, Springer, 2008.

[41] W. Eddy, "Tcp syn flooding attacks and common mitigations," RFC 4987, RFC Editor, August 2007.

[42] C. Paasch, S. Barre, *et al.*, "Multipath TCP in the Linux Kernel." Available from `http://www.multipath-tcp.org`.

[43] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, p. 19, ACM, 2010.

[44] P. Biondi, "Scapy." `http://http://secdev.org/projects/scapy/`, August 2007.

[45] G. License, "The gnu/linux cryptoapi," 2003.

[46] M. Wilcox, "I'll do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers," in *Linux. conf. au*, 2003.

[47] S. McCanne, C. Leres, and V. Jacobson, "Tcpdump," *Lawrence Berkeley National Laboratory, fipV/ftp. ee. lbl. gov/tcpdump. tar. Z*, 2001.

[48] P. Snyder, "tmpfs: A virtual memory file system," in *Proceedings of the Autumn 1990 EUUG Conference*, pp. 241–248, 1990.

Appendices

Appendix A

Scyther

The code we have implemented to model the MPTCPesn and MPTCPsec handshakes are available below. The results are also provided in this appendix. Figure A.1 shows the result when the Man-In-The-Middle attack is considered. Figure A.2 shows the results when we enforce identity authentication. This is made by replacing, in the code below, Line 109 by Line 108, and Line 176 by Line 175

Listing A.1: Scheduler

```
1  /*
2   * MPTCPesn and MPTCPsec handshakes
3   */
4
5  /* MPTCPesn handshake */
6
7
8  //=============================   GLOBAL DEF   ==================================
9
10
11 // ----------------------------   MPTCPesn handshake
       ------------------------------------------
12
13 usertype MP-CAPABLE;
14
15 // sub-options
16 usertype Sub-opt; // encryption spec
17 const MPTCPsec, other-sub-opt : Sub-opt;
18 usertype Gen-sub-opt; // general sub-option
19
20 // Syn, syn-ack and ack & transcript
21 macro syn = mp-capable-syn, MPTCPsec, other-sub-opt, gen-sub-opt-syn;
22 macro synack = mp-capable-synack, encry-spec-choosen, gen-sub-opt-synack;
23 macro ack = mp-capable-synack; // it is the confirmation of the synack
24 macro transcript = syn, synack;
25
26 // ----------------------------   init1 & init2 handshake
       ------------------------------------------
27
28 // Constants
29 usertype Number;
30 const INIT1-MAGIC, INIT2-MAGIC : Number;
31
32 // Symmetric ciphers that can be negotiated
33 usertype Sym-cipher;
34 const sym1, sym2 : Sym-cipher;
35
36 // init1 and init2 messages
37 macro init1 = INIT1-MAGIC, sym1, sym2, N-client, exp(g,X);
```

```
38 macro init2 = INIT2-MAGIC, sym3, N-server, exp(g,Y);
39
40 // ------------------------  Key generation
   ------------------------------------------------
41
42 // Diffie-Hellman key agreement
43 const exp: Function;
44 const g;
45 usertype Prime;
46
47 protocol @DH(X) {
48     role X {
49         var Y: Agent;
50         recv_!1(X,X, exp(exp(g,X),Y));
51         send_!2(X,X, exp(exp(g,Y),X) );
52     }
53 }
54
55 // Cryptographic functions
56 hashfunction Extract, CPRF;
57 const CONST-KEY-A, CONST-KEY-B : Number;
58
59 // ------------------------  Data transmission
   ------------------------------------------------
60
61 usertype Data;
62 hashfunction Auth-tag;
63
64
65 //=========================  PROTOCOL  ==============================
66
67
68 protocol Handshakes-MITM(Client,Server)
69 {
70
71 //-------------  CLIENT  ------------------------------------------------------
72
73     role Client
74     {
75
76     // - - - - - - - - - -  MPTCPesn handshake - - - - - - - - - - - - - - - - -
77
78         fresh gen-sub-opt-syn : Gen-sub-opt;
79         fresh mp-capable-syn: MP-CAPABLE;
80         var gen-sub-opt-synack : Gen-sub-opt;
81         var encry-spec-choosen : Sub-opt;
82         var mp-capable-synack : MP-CAPABLE;
83
84         send_esn-syn(Client, Server,  syn);
85         recv_esn-synack(Server, Client, synack);
86         send_esn-ack(Client, Server, ack);
87
88     // - - - - - - - - - -  Init1 & init2 exchange  - - - - - - - - - - - - - -
89
90         fresh N-client : Nonce;
91         fresh X : Prime;
92         var Y : Prime;
93         var sym3: Sym-cipher;
94         var N-server: Nonce;
95
96         send_init1( Client, Server, init1 );
97         recv_init2( Server, Client, init2 );
98
```

```
99
100     // - - - - - - - - - -  Key generation  - - - - - - - - - - - - - - - - - - - -
101
102         /* If we do not consider the Man-In-The-Middle attack,
103         * the key-agreement scheme can be supposed to be secure.
104         * We will therefore use the long-term symmetric key K(Client, Server)
105         * as the result of the key-agreement (the PMS). The following line must
106         * thus be changed by this one :
107         */
108         // macro PMS = k(Client, Server);
109         macro PMS = exp(exp(g,Y),X);
110         macro PRK = Extract(PMS, transcript, init1, init2, N-client); // the
                Pseudo-random-key
111         macro k-ab = CPRF(PRK, CONST-KEY-A); // key for data from Client to Server
112         macro k-ba = CPRF(PRK, CONST-KEY-B); // key for data from Server to Client
113
114     // - - - - - - - - - -  Data transmission  - - - - - - - - - - - - - - - - - -
115
116         fresh data1 : Data;
117         var data2 : Data;
118         var auth-tag2;
119
120         claim(Client, Running, Server, data1);
121         macro auth-tag1 = Auth-tag(k-ab,data1);
122         send_data1(Client, Server, {data1}k-ab, auth-tag1);
123         recv_data2(Server, Client, {data2}k-ba, auth-tag2);
124         match(auth-tag2, Auth-tag(k-ba,data2));
125
126     // - - - - - - - - - -  Claim  - - - - - - - - - - - - - - - - - - - - - - - -
           - - - - - - - - - - - - - - - - - - -
127         claim(Client, Secret, data1);
128         claim(Client, Secret, data2);
129         claim(Client, Commit, Server, data2);
130
131         claim(Client, Alive);
132         claim(Client, Weakagree);
133         claim(Client, Niagree);
134         claim(Client, Nisynch);
135
136     }
137
138 //-------------- SERVER ------------------------------------------------------
139
140     role Server
141     {
142
143     // - - - - - - - - - -  MPTCPesn handshake - - - - - - - - - - - - - - - - - -
144
145         var mp-capable-syn : MP-CAPABLE;
146         var gen-sub-opt-syn : Gen-sub-opt;
147         var MPTCPsec, other-sub-opt, encry-spec-choosen : Sub-opt;
148         fresh gen-sub-opt-synack : Gen-sub-opt;
149         fresh mp-capable-synack : MP-CAPABLE;
150
151         recv_esn-syn(Client, Server, syn);
152         match(encry-spec-choosen, MPTCPsec);
153         send_esn-synack(Server, Client, synack);
154         recv_esn-ack(Client, Server, ack);
155
156     // - - - - - - - - - -  Init1 & init2 exchange  - - - - - - - - - - - - - - - -
157         fresh N-server: Nonce;
158         fresh Y : Prime;
159         var X : Prime;
```

```
160         var sym1, sym2, sym3: Sym-cipher;
161         var N-client: Nonce;
162 
163         recv_init1( Client, Server, init1 );
164         match(sym3, sym2); // we choose one sym-cipher
165         send_init2( Server, Client, init2 );
166 
167     // ---------- Key generation  --------------------
168 
169         /* If we do not consider the Man-In-The-Middle attack,
170         * the key-agreement scheme can be supposed to be secure.
171         * We will therefore use the long-term symmetric key K(Client, Server)
172         * as the result of the key-agreement (the PMS). The following line must
173         * thus be changed by this one :
174         */
175         // macro PMS = k(Client, Server);
176         macro PMS = exp(exp(g,Y),X); // it is equivalent to exp(exp(g,X),Y)
177         macro PRK = Extract(PMS, transcript, init1, init2, N-client); // the
                Pseudo-random-key
178         macro k-ab = CPRF(PRK, CONST-KEY-A);  // key for data from Client to
                Server
179         macro k-ba = CPRF(PRK, CONST-KEY-B); // key for data from Server to Client
180 
181     // ---------- Data transmission  ------------------
182 
183         var data1 : Data;
184         fresh data2 : Data;
185         var auth-tag1;
186 
187         recv_data1(Client, Server, {data1}k-ab, auth-tag1);
188         claim(Server, Running, Client, data2);
189         macro auth-tag2 = Auth-tag(k-ba,data2);
190         send_data2(Server, Client, {data2}k-ba, auth-tag2);
191         match(auth-tag1, Auth-tag(k-ab,data1));
192 
193     // ---------- Claim  ----------------------------
194         claim(Server, Secret, data1);
195         claim(Server, Secret, data2);
196         claim(Server, Commit, Client, data1);
197 
198         claim(Server, Alive);
199         claim(Server, Weakagree);
200         claim(Server, Niagree);
201         claim(Server, Nisynch);
202     }
203 }
```

Scyther results : verify

Claim				Status		Comments	Patterns
Handshakes_MITM	Client	Handshakes_MITM,Client2	Secret data1	Fail	Falsified	At least 1 attack.	1 attack
		Handshakes_MITM,Client3	Secret data2	Fail	Falsified	At least 1 attack.	1 attack
		Handshakes_MITM,Client4	Commit Server,data2	Fail	Falsified	At least 1 attack.	1 attack
		Handshakes_MITM,Client5	Alive	Fail	Falsified	At least 1 attack.	1 attack
		Handshakes_MITM,Client6	Weakagree	Fail	Falsified	At least 1 attack.	1 attack
		Handshakes_MITM,Client7	Niagree	Fail	Falsified	At least 1 attack.	1 attack
		Handshakes_MITM,Client8	Nisynch	Fail	Falsified	At least 1 attack.	1 attack
	Server	Handshakes_MITM,Server2	Secret data1	Fail	Falsified	At least 1 attack.	1 attack
		Handshakes_MITM,Server3	Secret data2	Fail	Falsified	At least 1 attack.	1 attack
		Handshakes_MITM,Server4	Commit Client,data1	Fail	Falsified	At least 1 attack.	1 attack
		Handshakes_MITM,Server5	Alive	Fail	Falsified	At least 1 attack.	1 attack
		Handshakes_MITM,Server6	Weakagree	Fail	Falsified	At least 1 attack.	1 attack
		Handshakes_MITM,Server7	Niagree	Fail	Falsified	At least 1 attack.	1 attack
		Handshakes_MITM,Server8	Nisynch	Fail	Falsified	At least 1 attack.	1 attack

Done.

Figure A.1: Result of the Scyther analyze when considering the Man-In-The-Middle attack (i.e., without identity authentication).

Scyther results : verify

Claim				Status		Comments	Patterns
Handshakes_without_MITM	Client	Handshakes_without_MITM,Client2	Secret data1	Ok		No attacks within bounds.	
		Handshakes_without_MITM,Client3	Secret data2	Ok		No attacks within bounds.	
		Handshakes_without_MITM,Client4	Commit Server,data2	Ok		No attacks within bounds.	
		Handshakes_without_MITM,Client5	Alive	Ok		No attacks within bounds.	
		Handshakes_without_MITM,Client6	Weakagree	Ok		No attacks within bounds.	
		Handshakes_without_MITM,Client7	Niagree	Ok		No attacks within bounds.	
		Handshakes_without_MITM,Client8	Nisynch	Fail	Falsified	At least 1 attack.	1 attack
	Server	Handshakes_without_MITM,Server2	Secret data1	Ok		No attacks within bounds.	
		Handshakes_without_MITM,Server3	Secret data2	Ok		No attacks within bounds.	
		Handshakes_without_MITM,Server4	Commit Client,data1	Ok		No attacks within bounds.	
		Handshakes_without_MITM,Server5	Alive	Ok		No attacks within bounds.	
		Handshakes_without_MITM,Server6	Weakagree	Ok		No attacks within bounds.	
		Handshakes_without_MITM,Server7	Niagree	Ok		No attacks within bounds.	
		Handshakes_without_MITM,Server8	Nisynch	Fail	Falsified	At least 1 attack.	1 attack

Done.

Figure A.2: Result of the Scyther analyze when disabling the Man-In-The-Middle attack, by ensuring host that they have received the right public key.

Appendix B

Test environment

Figure B.1 and Figure B.2 stores the hardware and the configurations of, respectively, the hosts and the interfaces. These hosts and interfaces are used in the performance measures made in Chapter 8.

Host	Client	Server
RAM (GB)	4	2
CPU	Intel(R) Core(TM i5-2450M CPU)	Intel(R) Core(TM) CPU
Core number	4	2
Clock rate (GHz)	2.9	2.13
OS	Linux 4.1	Linux 4.1
Distribution	Ubuntu 14.04	Fedora 23
OpenSSL version	1.0.1f 6 Jan 2014	1.0.2h-fips 3 May 2016
mptcp_checksum	0	0
mptcp_path_manager	fullmesh	fullmesh
mptcp_scheduler	default	default
mptcp_version	2	2
AEAD algorithm	AES128-GCM	AES128-GCM

Table B.1: Hardware and configuration of the hosts

Interface	eth0	eth1	eth2
Host	Client	Server	Server
Max speed (Mb/s)	100	100	100
MTU (B)	1500	1500	1500
MTU used (B)	1500	1420	1420
Generic segmentation offload	x	x	x
TCP segmentation offload	x	x	x
Large reassemble offload	x	x	x
Generic reassemble offload	x	x	x
Scatter-gather	x	x	x
Checksum offload (reception)	v	x	v
Checksum offload (transmission)	v	x	v

Table B.2: Hardware and configuration of the Ethernet interfaces

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**