Securing MultiPath TCP: Design & Implementation

Mathieu Jadin*, Gautier Tihon, Olivier Pereira* and Olivier Bonaventure*
*ICTEAM, Université Catholique de Louvain, Louvain-la-Neuve, Belgium
Email: `firstname.lastname@uclouvain.be`

***Abstract*—MultiPath TCP (MPTCP) is a recent TCP extension that enables hosts to send data over multiple paths for a single connection. It is already deployed for various use cases, notably on smartphones. In parallel with this, there is a growing deployment of encryption and authentication techniques to counter various forms of security attacks. Tcpcrypt and TLS are some of these security solutions.**

In this paper, we propose MPTCPsec, a MultiPath TCP extension that closely integrates authentication and encryption inside the protocol itself. Our design relies on an adaptation for the multipath environment of the ENO option that is being discussed within the IETF tcpinc working group. We then detail how MultiPath TCP needs to be modified to authenticate and encrypt all data and authenticate the different TCP options that it uses. Finally, we implement our proposed extension in the reference implementation of MultiPath TCP in the Linux kernel and we evaluate its performance.

I. INTRODUCTION

The Transmission Control Protocol (TCP) [1] is the standard reliable transport protocol. It is used by a wide variety of applications. TCP was designed when hosts had a single interface and TCP connections are bound to the IP addresses of the communicating hosts. Despite that today's hosts, e.g. smartphones or laptops, have several network interfaces, once a TCP connection has been created over one interface, all the data belonging to this connection must be exchanged over this interface.

Multipath TCP [2] is a major extension to TCP that solves this limitation. With Multipath TCP, hosts can use different interfaces or paths to exchange the data belonging to a single connection. Multipath TCP has already seen significant adoption since it is used on all Apple devices to support the Siri voice recognition application. In Korea, high-end smartphones use Multipath TCP to combine WiFi and LTE and achieve higher bandwidth [3].

During the last decade, security has been a growing concern for Internet protocols and various protocols have been extended to include encryption and authentication techniques. The pace of deployment of these techniques has further increased during the recent years [4]. The most successful deployments are in the application layer with protocols like SSH or TLS [5]. In the network and transport layers, encryption and authentication have been less successful. The usage of IPsec remains restricted to some Virtual Private Networks (VPN). In the transport layer, progress has been slower. TCP has been tuned to reduce its vulnerability to some segment injection attacks [6, 7]. Two extensions add authentication to the TCP protocol [8, 9]. However, they assume that a secret is shared between the communicating hosts, which limits their applicability to very specific use cases like the protection of BGP sessions. Another approach is `tcpcrypt` [10]. This recent TCP extension adds opportunistic authentication and encryption to TCP. In contrast with application layer solutions such as TLS, it does not use certificates to authenticate the server, but enables the client and the server to securely negotiate keys. This prevents attacks such as pervasive monitoring [4] but does not address man-in-the-middle attacks. The IETF `tcpinc` working group is currently defining a TCP extension heavily inspired by `tcpcrypt` [10].

Multipath TCP was not designed in order to be more secure than TCP. While some mechanisms protect control information in the protocol specification [11, 12], they assume that attackers cannot capture nor modify the initial TCP handshake. The encryption and authentication techniques used by application layer protocols such as Transport Layer Security (TLS) [5] or SSH can protect data stream but they remain vulnerable to attacks that affect the underlying TCP (or MPTCP) protocol (e.g., [13]).

In this paper, we propose the first security scheme specially designed for Multipath TCP: **MPTCPsec** (MPTCP secure). MPTCPsec brings two main improvements compared to using application layer security protocols such as TLS or SSH above MPTCP:

- MPTCPsec can detect and recover from packet injection attacks by stopping to use a compromised path, in contrast with application layer protocols like TLS that just close the underlying TCP connection when they detect an attack.
- MPTCPsec protects the application data (with authentication and encryption) and authenticates the TCP options that are used to control the Multipath TCP connection. This prevents various types of attacks against the protocol itself where an attacker could force data to be sent only over paths that he controls.

We implement MPTCPsec within the reference implementation of Multipath TCP in the Linux kernel implementation [14]. Our implementation uses the Linux crypto API and performs all cryptographic operations inside the kernel. Our measurements show that it provides good performance and reacts correctly to packet injection attacks.

This paper is organised as follows. We first describe the key parts of Multipath TCP in Section II. We then detail

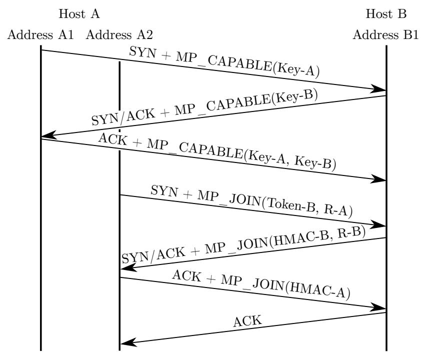

Host A
Address A1
Address A2
Host B
Address B1
SYN + MP_CAPABLE(Key-A)
SYN/ACK + MP_CAPABLE(Key-B)
ACK + MP_CAPABLE(Key-A, Key-B)
SYN + MP_JOIN(Token-B, R-A)
SYN/ACK + MP_JOIN(HMAC-B, R-B)
ACK + MP_JOIN(HMAC-A)
ACK

Figure 1. MPTCP three-way handshake (between addresses A1 and B1) and subflow establishment (between addresses A2 and B1). `R-A` (resp. `R-B`) is the random number of Host A (resp. Host B). `HMAC-A` (resp. `HMAC-B`) is equal to HMAC(R-A+R-B, Key-A+Key-B) (resp. HMAC(R-B+R-A, Key-B+Key-A)).

the main design choices of MPTCPsec in Section III. Section IV describes our implementation of MPTCPsec in the Linux kernel and we demonstrate in Section V. Finally, we compare MPTCPsec with related work in Section VI.

II. Multipath TCP

MultiPath TCP [2] was initially designed with multihomed hosts in mind. A typical example is a smartphone that wants to send the data belonging to one connection over both its WiFi and LTE interfaces. Each Multipath TCP connection is composed of a set of TCP connections that are called *subflows* in [2]. A subflow is created on each path that is used to support a given connection. The number of subflows that compose a Multipath TCP connection is not constant. Subflows can appear and disappear during the lifetime of a connection. For example, a smartphone can create a new subflow each time it attaches itself to a new WiFi access point. By using several subflows, Multipath TCP supports both mobility [15] and resource pooling [16]. A detailed overview of Multipath TCP may be found in [17]. We focus here on the TCP options that are used by Multipath TCP and are relevant for understanding the security of Multipath TCP [11, 12].

A. Connection and subflow establishment

A Multipath TCP connection starts with a three-way handshake. The `Multipath Capable` (`MP_CAPABLE`) option is used, as shown in the top of Figure 1, to negotiate the usage of MPTCP and exchange 64-bit keys. These keys are sent in clear during the handshake of the first subflow[1]. These keys are then used for three different purposes. Firstly, MPTCP hosts derive a token that uniquely identifies the Multipath TCP connection from a hash of the key [2]. Secondly, the `Initial Data Sequence Number` (IDSN) of a connection is also derived from these keys. Thirdly, the keys are used to authenticate the establishment of subflows with the `MP_JOIN` option.

[1]They also appear in RST segments that carry the `MP_FASTCLOSE` option see Section II-D.

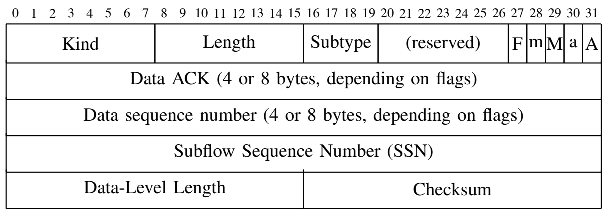


Figure 2. Format of the `DSS` option.

The `Join` (`MP_JOIN`) option is used to attach new subflows to an existing Multipath TCP connection. This option appears during the three-way handshake that establishes the additional subflow. An example is shown in Figure 1. When a host receives a `SYN` segment with the `MP_JOIN` option, it extracts its Token to map the subflow establishment attempt to an existing Multipath TCP connection (Token B in Figure 1). To prevent attacks, the client and the server authenticate the establishment of additional subflows with a HMAC computed over random numbers exchanged during the handshake (`R-A` and `R-B` in Figure 1) with the keys (Key-A and Key-B in Figure 1) that were exchanged during the initial TCP handshake.

B. Data transfer

The `Data Sequence Signal` (`DSS`) option, shown in Figure 2, enables the transmission of data over different subflows. In a nutshell, Multipath TCP uses two levels of sequence numbers: (i) the regular TCP sequence numbers at the subflow level and (ii) the Data Sequence Numbers (DSN) at the connection level. Each byte sent over a Multipath TCP connection is identified by one DSN that indicates its position in the bytestream. The DSN option maps these DSNs onto the TCP sequence numbers that are used in the TCP headers of the different subflows. Each `DSS` option maps a block of `Len` bytes from the DSN space to the subflow's sequence number. It also contains a cumulative acknowledgement field (Data ACK in Figure 2) and an optional Checksum that is used detect middlebox interference [2].

Thanks to the `DSS` option, it is possible to retransmit over a subflow data initially transmitted over another subflow. When a packet is lost, Multipath TCP first retransmits it over the same subflow. This retransmitted packet contains the same `DSS` option as the original one. If a subflow fails, unacknowledged data previously sent on this subflow must be retransmitted on another subflow. This is called a `re-injection` [2]. In this case, a new `DSS` option is computed to map the retransmitted data onto the new subflow.

The `DSS` option contains a series of flags whose semantics is detailed in [2]. In this paper, we only consider the `DATA_FIN` (`F` flag in Figure 2). It marks the end of the bytestream and plays for a Multipath TCP connection the same role as the `FIN` flag in TCP.

C. Address advertisement

To establish subflows between their different interfaces, Multipath TCP hosts must learn the IP addresses of the remote host. Multipath TCP relies on two TCP options for this purpose. The `Add Address` (`ADD_ADDR`) and `Remove Address` (`REMOVE_ADDR`) options enable hosts to advertise and remove addresses on which subflows can be established. These options are typically included in pure acknowledgements. The host that receives an option can decide to create a subflow or not towards the advertised address. The ability to advertise addresses creates a security threat that is discussed in [11, 12]. The latest Multipath TCP draft [18] has added a HMAC in the `ADD_ADDR` option to authenticate it.

D. Connection release

A Multipath TCP connection can be closed gracefully by using the `DATA_FIN` flag or abruptly with the `Fast Close` (`MP_FASTCLOSE`) option. When a host sends a `RST` segment containing the `MP_FASTCLOSE`, it requests the abrupt release of the corresponding Multipath TCP connection (and also of all the corresponding subflows). This creates a security risk if an attacker has been able to send a spoofed RST segment containing this option. To mitigate this risk, the `MP_FASTCLOSE` option must be authenticated by including the security key that has been exchanged during the establishment of the initial subflow.

III. Design of MPTCP Secure

Multipath TCP was designed as a replacement for TCP. From a security viewpoint, the security objectives were that Multipath TCP should not be worse than regular TCP.

Possible attacks on MPTCP fall under 3 categories: eavesdropping, data modification (including injection and truncation) and Denial of Service. The latter group includes all attacks consisting in MPTCP option forgery, replay and removal. For instance, an attacker, if he controls one of the subflows, can block new subflows by removing any `ADD_ADDR` option or by forging `REMOVE_ADDR` options to remove addresses.

Single-path security solutions existing today (like TLS or SSH) tackle eavesdropping and data modification but they do not tackle DoS attacks since they were designed to work above TCP (and they cannot be modified to protect MPTCP options without being merged to the TCP stack).

Moreover, a new risk of denial of service is added by security solutions initially designed to work above TCP: if an attacker modifies data of one segment and successfully injects it (that's easy if he controls one of the paths used by MPTCP), an authentication error is raised at the security layer (e.g., in TLS or SSH) and the entire MPTCP connection is closed. Indeed, the segment was already accepted and acked by MPTCP and the other host will never retransmit it.

This problem can be solved by checking the validity of a segment before acknowledging it and if invalid, closing the attacked subflow. This is possible only if we design a secure protocol inside MPTCP (instead of above), taking advantage of the multipath environment. We not only address eavesdropping and data modification but we also use the specificity of Multipath TCP to provide solutions to many Denial of Service attacks (i.e., by MPTCP option protection and avoidance of the attacker by using several subflows), resulting in more robust connections than what would be obtained from the use of an application layer security protocol.

In general, three phases can be observed in protocols using cryptography to carry data above TCP. This section is thus organized around these phases:

- The TCP connection establishment (Section III-A). We use it to negotiate MPTCP and extend TCP-ENO [19] to negotiate encryption in MPTCP.
- The secure handshake (Section III-B): this step is used to define unique session identifiers, to create the keying material that is needed for the authenticated encryption scheme used for data transfer, and possibly to check identities (e.g., through certificates).
- The protected data transfer (Section III-C): MPTCP secure (MPTCPsec) chooses to protect both data and MPTCP options.

A. Encryption suite negotiation

TCP does not include encryption and authentication. Within the `tcpinc` working group, the IETF is working on TCP-ENO [19], a generalized and backward compatible mechanism for incrementally deploying encryption. TCP-ENO is a TCP option that can be used in the SYN segment to negotiate the utilisation of a specific encryption/authentication scheme for a TCP connection. Its first use is to support `tcpcrypt` as described in [20], but the specification is open to other protocols that can meet similar security requirements, like TLS 1.3. We extend TCP-ENO to use it within a Multipath TCP connection.

MPTCP is negotiated by using the `MP_CAPABLE` option, which must be included once in each segment of any successful three-way handshake. In MPTCP, this option is also used to negotiate (in clear) keys that will be used to protect some of the options, on other subflows in particular. Exchanging keys in clear is obviously insecure when massive surveillance is a central concern. Therefore, we remove the keys from the `MP_CAPABLE` option and use the spared space to encode our MultiPath TCP Encryption Scheme Negotiation (MPTCPesn) option. Our MPTCPesn relies on the chosen encryption scheme to derive the necessary keys.

Figure 3 shows the modified `MP_CAPABLE` option. It uses the traditional Type-Length-Value format imposed by TCP. As stated in [2], the Kind indicates MPTCP type while the Subtype indicates which MPTCP option it is (in this case, it indicates the subtype of `MP_CAPABLE`). In order to indicate a MPTCPesn negotiation, the version field value is changed (into second version). This modified `MP_CAPABLE` option contains a list of sub-options, whose format is identical to TCP-ENO sub-options (see [19] for more details). One of the sub-options is used to perform out-of-band signalization to applications that would be aware of MPTCPsec, while the others are used for the negotiation of cipher suites.

Our `MP_CAPABLE` option is used as in MPCTP and TCP-ENO: it is used in the three segments of the TCP handshake, and provides a transcript and connection roles that can later

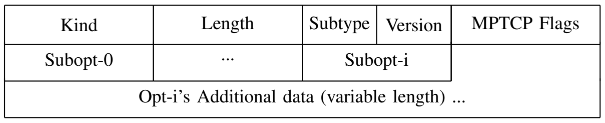


Figure 3. `MP_CAPABLE` option modified for MPTCPesn

be used to authenticate the negotiation. Furthermore, it aims at negotiating an encryption spec meeting all the requirements included in TCP-ENO, including to provide an authenticated encryption mechanism with minimum 128-bit security, and a unique session ID. In addition to defining the session ID, MPTCPesn also requires the encryption scheme to provide the MPTCP tokens and IDSN that are necessary for the MPTCP connection.

As in TCP-ENO, our `MP_CAPABLE` option does not prevent an on-path attacker (or a middlebox) from removing the `MP_CAPABLE` option itself from the handshake. In this case, no encryption will be negotiated and the application may choose to close the connection in order to prevent attacks.

B. Secure handshake

The MPTCP Secure (MPTCPsec) design can be used with the shared secret establishment method belonging to another secure protocol (e.g., tcpcrypt [20] and TLS 1.3 [21]). This shared secret establishment method must create keying material (and possibly nonces or IVs) that can be used for AEAD algorithms [22], which are required to be used for securing data in TCP-ENO. We describe below how we use this keying material in MPTCPsec.

Once the TCP three-way handshake is over, the chosen secure protocol can be run above the freshly established TCP subflow (since the token is not yet chosen, other subflows cannot be established). The difficulty is to introduce the MPTCPesn transcript in the keying material generation. TLS (in [23]) has already proposed a solution by providing an interface to use only its handshake to derive some keying material and by letting the application add some context to the computation: we can thus give the MPTCPesn transcript. Tcpcrypt [20] does not have this problem since its handshake already includes the TCP-ENO transcript.

From this keying material, must be derived the keys and the session ID, which TCP-ENO requires to contain at least 33 bytes. There is no additional operation to do in tcpcrypt. In TLS, we simply ask for more bytes from the TLS Keying material exporter and use the first part for the keys and the second part for the session ID. The same mechanism is used with TLS in order to produce the MPTCP connection tokens and the Initial Data Sequence Numbers (IDSN). For tcpcrypt, we apply HKDF [24] on the session ID to derive a uniform sequence of bits.

C. Securing data and control

This section gives an overview of how MPTCPsec protects data and options.

1) Protecting data: All data are protected through the use of Authenticated Encryption with Associated Data (AEAD) algorithms: following [22], an encryption algorithm takes data and associated data as inputs (which will be options in our case), and guarantees the integrity of both types of data and the confidentiality of the data only. These guarantees are provided only if a shared secret key is used and if a fresh nonce is used for every invocation of the encryption algorithm.

In order to guarantee the uniqueness of the nonce used for data protection, we define it as a constant of 2 bits (00 for the data) followed by the DSN and padded with zeros to the right in order to have a 128-bit nonce.

MPTCPsec does not use records like TLS, tcpcrypt and SSH do, but it 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. It enables to fragment a ciphertext into several segments by covering all these segments by only one `DSS`. Since these fragments are mapped by a single `DSS` option, they must be sent over the same subflow.

2) Protecting TCP Options: Multipath TCP uses a variety of TCP options. In order to protect them, we have chosen to authenticate the TCP options but we do not encrypt them. This is a different design than the one chosen by the designers of QUIC [25]. In QUIC, all control information is both encrypted and authenticated to prevent interference with middleboxes. Experience with Multipath TCP has shown that it can cope well with a variety of middleboxes. Some of these middleboxes have valid reasons to analyse the control information exchanged in the Multipath TCP options. For this reason, we have opted for authenticating but not encrypting the TCP options.

Data are encrypted independently of their headers (which need to be adapted depending on the subflow that is used) but, in order to link the data with these headers, MPTCPsec computes the option authentication tag based on the concatenation of the options and the data authentication tag. It is then appended to the ciphertext in the payload (i.e., after the encrypted data and its authentication tag). The format of the nonce used for the option protection (with the exception of pure acknowledgments) is a constant of 2 bits (01) followed by a subflow id (because options change in a re-injection and the nonce must be different) and the DSN. We define the subflow id as the concatenation of the first 31 bits of both HMAC exchanged during the `MP_JOIN` messages, and set the id of the first subflow to 0. Figure 4 represents this segment format (assuming that the used AEAD algorithm puts the tag at the end of the ciphertext).

The two calls to the AEAD encryption mode may seem sub-optimal, since we just need to encrypt the data and then authenticate the ciphertext and headers – something that seems to map to a single AEAD encryption. However, the AEAD API does not allow defining associated data *after* the encryption process. Furthermore, this is not even possible for the AEAD modes listed in tcpcrypt and TLS (i.e., GCM, CCM and CHACHA20_POLY1305) without requiring a full pass on the ciphertext every time a header modification happens (which

Table I
PROTECTION OF THE MPTCP OPTIONS AND HANDLING OF REINJECTIONS.

Options	Protection	Re-injection
MP_CAPABLE	MPTCPesn transcript	Cannot be re-injected
MP_JOIN	HMAC exchanged in the options	Cannot be re-injected
DSS	Included in the option tag	SSN must be modified. Data ACK and DSN can be truncated or not (i.e., flags a and m can change).
ADD_ADDR	Included in the option tag	Nothing can be changed
REMOVE_ADDR	Included in the option tag	Nothing can be changed
MP_FASTCLOSE	HMAC in the option	Nothing can be changed

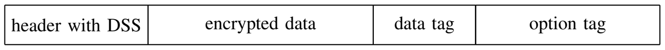


Figure 4. Segment format (assuming that the AEAD algorithm puts the tag at the end of the ciphertext), with *tag* the authentication tag produced by the AEAD.

would happen if a ciphertext needs to be retransmitted on a different subflow). Other modes (e.g., OCB) could lead to more efficient solutions, but with a non blackbox use of the AEAD.

There is an important issue for the protection of the acknowledgments. With the current protocol, as ACK segments contain a `DSS` option, they also contain an authentication tag, in the payload. Thus, they always contain data (there are no pure ACK segments in MPTCPsec), and this data must be acknowledged at the MPTCP level. In other words, each ACK segment must be acknowledged itself by another ACK segment. This could become an endless loop. To solve this issue, in case of a pure ACK segment, MPTCPsec still puts the option tag in the payload but does not map its bytes onto DSNs, so that no further MPTCP ACK will be triggered. The format of the nonce used for the acknowledgment protection is a constant of 2 bits (11) followed by the value of the "a" flag of the `DSS` option (because the Data-ACK in the segment can be either fully written or truncated) and the Data-ACK value (of 64 bits).

Table I summarises how the MPTCP options are protected and explains how they can or must be modified in case of re-injection (in order to keep the same semantics). As shown in this figure, `DSS`, `ADD_ADDR` and `REMOVE_ADDR` options are protected by using the option tag. The `MP_CAPABLE` option is protected by the MPTCPesn transcript (described in Section III-A). It cannot be protected by the option tag since the keys are not yet defined at that time.

Since `MP_JOIN` already uses authentication in its handshake, MPTCPsec does not authenticate the MP_JOIN option. But the `B` bit (indicating if this subflow will be a backup subflow or not) 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 key used by the `MP_FASTCLOSE` option in [2] is no more available with MPTCPsec. MPTCPsec sends the HMAC of the remote key, using the local key as secret.

To summarize (as shown in Table II), these mechanisms guarantee that:

- data is confidential and authentic, thanks to the AEAD;
- `DSS`, `ADD_ADDR` and `REMOVE_ADDR` options are authentic, and linked to the data, thanks to the use of the associated data feature of the AEAD;
- the only effect of an attack on one subflow will be to close this subflow and retransmit the data on another subflow, without the need of renegotiating fresh keying material.

IV. IMPLEMENTATION

To demonstrate the feasibility of adding authentication and encryption to Multipath TCP, we extend the reference implementation of Multipath TCP in the Linux kernel with the mechanisms described in this paper. Our prototype implementation lies entirely in the kernel. It significantly extends the Multipath TCP stack since our patch[2] contains 5130 lines.

Our implementation is a proof-of-concept that includes the different mechanisms described in this paper except the secure handshake. As explained in Section III-B, MPTCPsec has been designed to support different secure handshake mechanisms, including TLS 1.3 and `tcpcrypt`. The secure handshake is only used to derive the shared secret on the communicating hosts. In our experiments, we use a socket option to pass the same shared secret on the two hosts once the MPTCPsec connection has been established.

We first describe the operation of our implementation at a high level and then explain how the AEAD operations interact with the transmission and the reception of Multipath TCP segments.

When sending or receiving segments, our implementation follows a precise order for the operations in order to ensure the security properties.

- **First transmission of a segment**: firstly, our implementation executes the AEAD on the data to be sent. Then the Multipath TCP scheduler chooses the subflow on which the segment will be sent. It is only at this point that we can create the TCP header, including the different options and compute the authentication tag. The entire segment is then ready to be transmitted on the chosen subflow.

[2]The starting point for our implementation is the `mptcp_trunk` branch, kernel version 4.1 and MPTCP version 0.90. The code is available at https://bitbucket.org/mptcpsecteam/mptcpsec.

Table II
ATTACK VECTORS AND THEIR MITIGATIONS.

	Eavesdropping	Data modification	Option modification	Option replay
Secure handshake	v	v	v	
AEAD on data	v	v		
AEAD on options			v	v
HMAC in MP_FASTCLOSE			v	
HMAC in MP_JOIN			v	v

- **Retransmission of a segment**: we store the segments that have not yet been acknowledged. When such a segment needs to be retransmitted, it is sent with the same header and the same tags, on the same subflow.
- **Segment reinjection**: Multipath TCP can decide to retransmit an unacknowledged segment over a different subflow for different reasons [17]. When a segment is reinjected, our implementation modifies the `DSS` and possibly the other options included in the segment. Then it recomputes the authentication tag of the options, but does not modify the ciphertext. We also fragment the segment if the MTU of the new subflow is smaller than the length of the reijnected segment.
- **Reception of a segment**: our implementation checks whether the segment is in the MPTCP receive window and has not already been received (otherwise it is simply dropped). Then it decrypts the ciphertext and checks the option tag. If the tag is valid, it processes the options contained in the segment. Otherwise, the subflow where the invalid segment was ready is closed. This does not terminate the Multipath TCP connection since such a connection can gather different subflows. The decrypted data is passed to the application in sequence.

It is important to note that when receiving a segment, we not only check the authentication of the options but we also decrypt and authenticate the data. Otherwise, an attacker could replay the options, by using the following method:

- Host A sends a segment with a `ADD_ADDR` (for instance).
- The attacker corrupts one bit of the encrypted data (i.e., not the data tag nor the option tag).
- Host B sees that options are authenticated correctly and applies directly the `ADD_ADDR` option. But after attempting to process the data, it realizes that it is corrupted and therefore, drops the segment and closes the subflow.
- Host A re-injects the segment on another subflow with the same `ADD_ADDR` option.
- Host B sees again that options are authenticated correctly and applies a second time the `ADD_ADDR` option. This time, the data are accepted and thus, host B will not apply a third time this option.

A more precise order of the interpretation of the option is shown in Figure 5. Pure ACK segments and `MP_FASTCLOSE` options do not contain MPTCP data 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. The `DATA_FIN` is only interpreted once the previous data have been received.

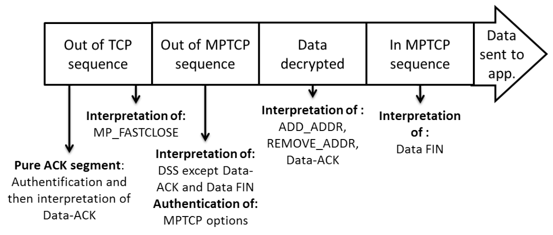

Out of TCP sequence
Out of MPTCP sequence
Data decrypted
In MPTCP sequence
Data sent to app.
Interpretation of: MP_FASTCLOSE
Pure ACK segment: Authentification and then interpretation of Data-ACK
Interpretation of: DSS except Data-ACK and Data FIN
Authentication of: MPTCP options
Interpretation of : ADD_ADDR, REMOVE_ADDR, Data-ACK
Interpretation of : Data FIN

Figure 5. Timeline of the interpretation of the MPTCP options.

A. AEAD integration in the input flow

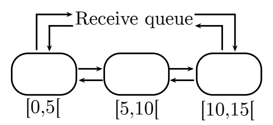
We use the GNU/Linux CryptoAPI [26] of the kernel to implement the cryptographic operations. This will enable our implementation to support other AEAD algorithms in the future with minor modifications to the code.

The GNU/Linux CryptoAPI requires the AEAD operations to be asynchronous. This is the most efficient choice given that cryptographic operations are costly from a CPU viewpoint. However, this API was designed to support the encryption/-decryption of independant packets. Using it for a stream protocol such as Multipath TCP poses special implementation challenges. One of them is that we have no guarantee that the first execution of the AEAD algorithm will be finished before the next ones. Moreover, any received segment must be dropped if its verification fails (and its subflow must be closed). For these reasons, our implementation uses two new queues: 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 that are not in sequence.

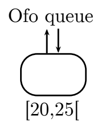
Figure 6 shows an example of a possible state of the queues of the input flow. These queues belong to different `socks`. In TCP, a `sock` is the kernel equivalent to the socket in Linux. In MPTCP, there is not only one sock like in TCP :

- 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.
- 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.

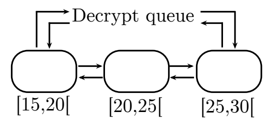
When new segments arrive, they are queued in the subflow's `Receive queue` if they are received in se-



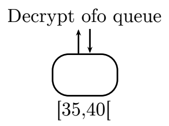
(a) Receive queue of the meta sock



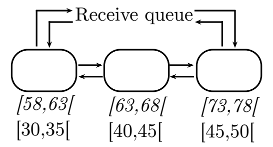
(b) Out-of-order queue of the meta sock



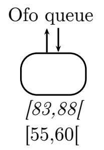
(c) Decrypt queue



(d) Decrypt out-of-order queue

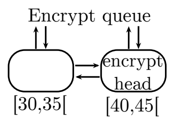


(e) Receive queue of a subflow sock

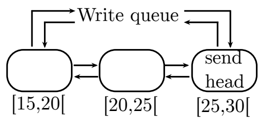


(f) Out-of-order queue of a subflow sock

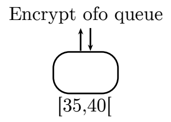
Figure 6. Example of state of the modified MPTCP input flow's structures. In Figure 6e and Figure 6f, the intervals in italic contain SSN while the other ones contain DSN.



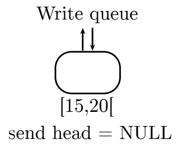
(a) Encrypt queue



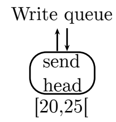
(b) Write queue of the meta sock



(c) Encrypt out-of-order queue



(d) Write queue of the master sock



(e) Write queue of a slave sock

Figure 7. Example of state of the modified MPTCP output flow where the intervals are the Data Sequence Numbers covered by the segments

quence. 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 in sequence (w.r.t. DSN) or to the `Decrypt out-of-order queue` otherwise. In both cases, the AEAD decryption and authentication checks are performed. Nevertheless, these segments cannot already be in the `Decrypt queue`, in the `Decrypt out-of-order queue` or in the meta sock's Out-of-order queue.

If an authentication check fails, 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` since they are not in sequence anymore.

B. AEAD integration in the output flow

As for the AEAD decryption and verification, the AEAD encryption and authentication must be performed asynchronously. Our implementation uses two queues to organise this processing:

- An `Encrypt queue` (Figure 7a) 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 acknowledged), either because the AEAD algorithm has still not been launched (the first of these segments is pointed by `encrypt_head`) or because AEAD is still running.
- An `Encrypt out-of-order queue` (Figure 7c) stores segments that are encrypted and authenticated while the previous ones are not.

When data arrives from the application's socket, segments are created and placed 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 to the next segment.

As soon as the authentication and encryption of a segment finish, the segment is removed from the `Encrypt queue`. If it is not the first of the `Encrypt queue`, it is placed in the `Encrypt out-of-order queue`. Otherwise it is appended at the end of the meta sock's Write queue (Figure 7b) 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 7b, Figure 7d and Figure 7e) are handled exactly like regular MPTCP.

MPTCP options are authenticated before sending the segments on the subflow and, thus, do not require additional queues.

V. Evaluation

In this section, we compare the performance of our implementation with plain MPTCP and TLS 1.2 (using OpenSSL) over MPTCP. The main objective of this evaluation is to demonstrate that MPTCPsec works correctly. Its code has not been as optimised as the OpenSSL and the MPTCP codes. A lower performance of MPTCPSec is thus expected, even if encryption and authentication are computed inside the kernel while OpenSSL operates entirely in userspace.

We perform our evaluation with two low-end x86 servers running Linux 4.1 (with MPTCP version 0.90) and connected back-to-back via two Gbps Ethernet interfaces. We use AES128-GCM AEAD for both MPTCPsec and OpenSSL and disabled the hardware offload on the Ethernet interfaces. The two x86 servers used for the tests had AMD CPUs and did not include the instructions to implement AES in hardware.

We first perform bulk data transfers. The client application sends a given amount of data (from 100MB to 1GB) from memory to the server application by blocks of 4096 bytes.

There is no interruption of the data transfer. Our test application collects a timestamp for each block. Each transfer was repeated 15 times and we plot the mean of the throughput measurements in Figure 8. In our testbed, MPTCP can reach roughly 1.6 Gbps. When we enable TLS, using OpenSSL version 1.0.2g, the throughput drops to about 700 Mbps. This reflects the cost of encrypting and authenticating the data. With MPTCPsec instead of MPTCP, the throughput drops to 350 Mbps. This lower performance is not surprising since our code has not been optimised in the same manner as the OpenSSL code that is used in production by many servers.

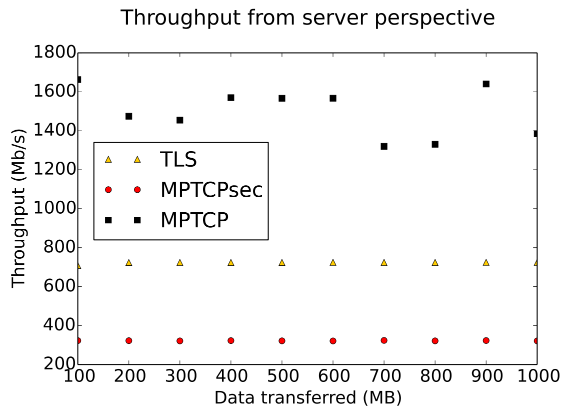


Figure 8. Throughputs achieved by MPTCP, TLS/MPTCP and MPTCPSec for long transfers

The main benefit of MPTCPsec compared to TLS over MPTCP is its reaction to different types of attacks. As explained earlier, MPTCPsec also authenticates the MPTCP options while TLS only authenticates and encrypts the data. Thanks to this, MPTCPsec can efficiently react to a range of attacks. To demonstrate the capabilities of MPTCPsec, we consider the following scenario. The client application sends 1MB to the server by blocks of 4096 bytes. A Linux PC, acting as a router is placed between the client and the server to perform a man-in-the-middle attack. More precisely, we use scapy [27] to capture the packets sent by the client. Once 500 KB have been transferred, the attacker modifies one byte of the payload of one segment and updates its checksum. This models a very simple packet injection attack. Researchers have identified various middleboxes that inject data in established TCP connections in mobile networks [28].

We perform experiments with both TLS 1.2 over MPTCP and MPTCPsec. In both cases, two subflows are established between the client and the server. Figure 9 shows the evolution of the DSN with both TLS/MPTCP and MPTCPsec. When TLS detects the authentication error, shortly after 500 KB, it immediately closes the connection and stops the transfer. MPTCPsec on the other hand detects the problem and only closes the affected subflow. As shown in Figure 9, the secure data transfer continues after the attack.

VI. Related work

Encryption and authentication have traditionally been implemented either in the application [5] or in the network layer.

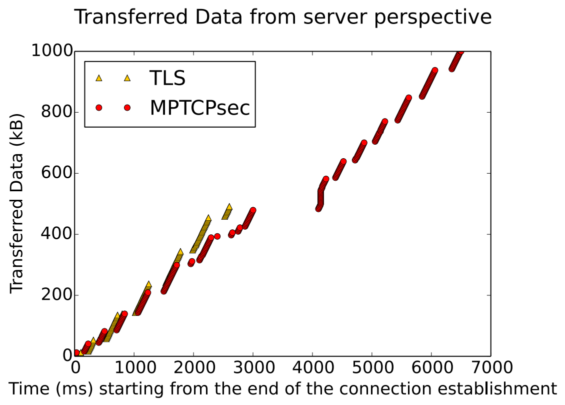


Figure 9. When a packet injection attack occurs, TLS/MPTCP closes the connection and stops the transfer while MPTCPSec closes the affected subflow and continues the transfer.

Application layer solutions have been quite successful and a growing fraction of the Internet traffic is encrypted and authenticated. Compared to application layer solutions, MPTCPsec provides several advantages. First, application layer protocols close the underlying connection as soon as some data has been injected in the underlying connection. This is a strong risk of denial of service. MPTCPsec reacts to such attacks by either ignoring the injected segments that are not authenticated or by closing the subflow, but not the entire connection.

Within the IETF, two drafts are related to MPTCPsec : MPTLS [29] and Secure MPTCP [30]. These two drafts propose initial ideas that have not yet been implemented. MPTLS proposes to couple Multipath TCP and TLS together. It uses Encrypt-then-MAC and places the HMAC computed by TLS for each record inside the `DSS` option. MPTCPsec leverages AEAD to secure both data and options. Secure MPTCP [30] discusses how `tcpcrypt` could be included inside Multipath TCP, but only at a high level. Other researchers have proposed to use a key derived by a secure handshake to authenticate the establishment of the Multipath TCP subflows [31]. This solution does not provide encryption or authentication for the data and the other Multipath TCP options. An earlier work, [32] proposed to use hash chains to secure the control part of Multipath TCP instead of the HMAC solution chosen by the IETF. To our knowledge, this solution has not been implemented.

In 2012, Andrea Bittau and his colleagues proposed to extend TCP in order to support opportunistic encryption [10]. With their `tcpcrypt` proposal, hosts negotiate keys securely at the beginning of the connection and encrypt and authenticate all the data exchanged. The IETF has formed the `tcpinc` working group to standardise a solution that is heavily inspired from this work. MPTCPsec has three main differences with `tcpcrypt`. First, `tcpcrypt` can only work with TCP and thus, cannot use multiple paths. Second, `tcpcrypt` cannot protect control data since it is included in the TCP header. Indeed, TCP is an old protocol and many middleboxes may modify control information included in the TCP header. Finally, MPTCPsec can use either the TLS or the `tcpcrypt` handshake (Section III-B).

VII. Conclusion

The traditional TCP/IP protocol stack is not sufficient for today's Internet users. On one hand, they expect better security and require the utilisation of cryptographic encryption and authentication techniques to secure their data transfers. On the other hand, they use mobile devices that include multiple interfaces. In this paper, we have proposed and implemented MPTCPsec, an extension to Multipath TCP that relies on authentication and encryption to counter various security attacks. Our design is composed of three parts. Firstly, MPTCPesn adapts TCP-ENO to Multipath TCP and allows to negotiate different security protocols during the three way handshake. Secondly, MPTCPsec authenticates and encrypts the data transported over a connection while still supporting their reinjection over other subflows. Thirdly, MPTCPsec authenticates the MPTCP options to prevent several types of denial of service attacks. To demonstrate the feasibility of our approach, we implement MPTCPsec entirely in the Linux kernel. Our further work will be to port the tcpcrypt and TLS 1.3 secure handshakes inside MPTCPsec.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work is partially supported by the ARC grant 13/18-054 (ARC-SDN) from Communauté française de Belgique and by the Innoviris project SeCloud.

References

[1] J. Postel, "Transmission control protocol," RFC 793, September 1981.

[2] A. Ford *et al.*, "TCP extensions for Multipath operation with multiple addresses," RFC 6824, January 2013.

[3] O. Bonaventure and S. Seo, "Multipath TCP deployments," *IETF Journal*, Nov. 2016.

[4] S. Farrell and H. Tschofenig, "Pervasive monitoring is an attack," RFC 7258, May 2014.

[5] E. Rescorla, *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2000.

[6] F. Gont, "ICMP attacks against TCP," RFC 5927, July 2010.

[7] A. Ramaiah *et al.*, "Improving TCP's robustness to blind in-window attacks," RFC 5961, August 2010.

[8] A. Heffernan, "Protection of BGP sessions via the TCP MD5 signature option," RFC 2385, August 1998.

[9] J. Touch, A. Mankin, and R. Bonica, "The TCP authentication option," RFC 5925, June 2010.

[10] A. Bittau *et al.*, "The case for ubiquitous transport-level encryption," in *USENIX Security'10*. Berkeley, CA, USA: USENIX Association, 2010, pp. 26–26.

[11] M. Bagnulo, "Threat analysis for TCP extensions for Multipath operation with multiple addresses," RFC 6181, March 2011.

[12] M. Bagnulo *et al.*, "Analysis of Residual Threats and Possible Fixes for Multipath TCP (MPTCP)," RFC 7430, Jul. 2015.

[13] F. Gont, "Security assessment of the transmission control protocol (TCP)," Internet-Draft draft-gont-tcp-security-00, February 2009.

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

[15] C. Paasch *et al.*, "Exploring mobile/wifi handover with Multipath TCP," in *Cellnet'12*. ACM, 2012, pp. 31–36.

[16] D. Wischik, M. Handley, and M. B. Braun, "The resource pooling principle," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 5, pp. 47–52, 2008.

[17] C. Raiciu *et al.*, "How hard can it be? Designing and implementing a deployable Multipath TCP," in *NSDI'12*. USENIX, 2012, pp. 29–29.

[18] A. Ford, C. Raiciu, M. J. Handley, O. Bonaventure, and C. Paasch, "TCP extensions for Multipath operation with multiple addresses," Internet-Draft draft-ietf-mptcp-rfc6824bis-07, October 2016.

[19] A. Bittau *et al.*, "TCP-ENO: Encryption negotiation option," Internet-Draft draft-ietf-tcpinc-tcpeno-06, October 2016.

[20] ——, "Cryptographic protection of TCP streams (tcpcrypt)," Internet-Draft draft-ietf-tcpinc-tcpcrypt-03, October 2016.

[21] E. Rescorla, "Using TLS to protect TCP streams," May 2016, Internet draft, draft-ietf-tcpinc-use-tls-01.

[22] D. McGrew, "An interface and algorithms for authenticated encryption," RFC 5116, January 2008.

[23] E. Rescorla, "Keying material exporters for transport layer security (TLS)," RFC 5705, March 2010.

[24] H. Krawczyk, "Cryptographic extraction and key derivation: The HKDF scheme," in *Annual Cryptology Conference*. Springer, 2010, pp. 631–648.

[25] R. Hamilton, J. Iengar, I. Sweet, and A. Wilk, "QUIC: A UDP-Based Multiplexed and Secure Transport," January 2017, internet draft draft-ietf-quic-transport-01.

[26] S. Mueller and M. Vasut, "The GNU/Linux CryptoAPI," http://www.chronox.de/crypto-API/.

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

[28] N. Weaver *et al.*, "Here be web proxies," in *PAM'14*. Springer, 2014, pp. 183–192.

[29] O. Bonaventure, "MPTLS : Making TLS and Multipath TCP stronger together," Oct. 2014, internet draft, draft-bonaventure-mptcp-tls-00, work in progress.

[30] M. Bagnulo, "Secure MPTCP," Feb 2014, internet draft, draft-bagnulo-mptcp-secure-00, work in progress.

[31] C. Paasch and A. Ford, "TLS Authentication for MPTCP," May 2016, internet draft, draft-paasch-mptcp-tls-authentication-00, work in progress.

[32] J. Diez *et al.*, "Security for Multipath TCP: a constructive approach," *International Journal of Internet Protocol Technology*, vol. 6, no. 3, pp. 146–155, 2011.