Are TCP Extensions Middlebox-proof?

Benjamin Hesmans, Fabien Duchene, Christoph Paasch,
Gregory Detal and Olivier Bonaventure
ICTEAM, Université Catholique de Louvain
Louvain-La-Neuve – Belgium
firstname.name@uclouvain.be

ABSTRACT

Besides the traditional routers and switches, middleboxes such as NATs, firewalls, IDS or proxies have a growing importance in many networks, notably in entreprise and wireless access networks. Many of these middleboxes modify the packets that they process. For this, they to implement (a subset of) protocols like TCP. Despite the deployment of these middleboxes, TCP continues to evolve on the endhosts and little is known about the interactions between TCP extensions and the middleboxes.

In this paper, we experimentally evaluate the interference between middleboxes and the Linux TCP stack. For this, we first propose *MBtest*, a set of `Click` elements that model middlebox behavior. We use it to experimentally evaluate how three TCP extensions interact with middleboxes. We also analyzes measurements of the interference between Multipath TCP and middleboxes in fifty different networks.

Categories and Subject Descriptors

C2.5 [**Local and Wide-Area Networks**]: Internet; C2.6 [**Internetworking**]: Standards

Keywords

Protocol; TCP; Multipath TCP; Middlebox

1. INTRODUCTION

The TCP/IP protocol suite was designed with the end-to-end principle in mind [13]. In particular, the design of the Transmission Control Protocol (TCP) assumed that routers never modify any field in the TCP headers or payloads. As the Internet grew, the need for more sophisticated nodes rose. To protect against malicious nodes, firewalls were introduced. Network Address Translation (NAT) was added to overcome the problems that were encountered with the insufficient IPv4 address space. More and more middleboxes have been introduced over time, effectively disrupting the end-to-end principle.

HotMiddlebox'13, December 9, 2013, Santa Barbara, CA, USA.

http://dx.doi.org/10.1145/2535828.2535830.

During the last decade, a wide variety of middleboxes have been proposed, implemented and deployed [14]. These middleboxes are frequently found in enterprise and cellular networks [15]. A first analysis was carried out by Medina et al [9], using TBIT [10]. This study showed that middleboxes interfere with TCP extensions, effectively decreasing TCP's performance. Honda *et al.* [6] took another approach, trying to detect the behavior of the Internet's middleboxes by sending and recording specially crafted TCP segments. They conclude that any extension to TCP must be designed with middleboxes in mind. Despite the widespread utilization of these middleboxes, there are few detailed documents or tools that model their behavior. Such tools could be used by designers of new protocols or TCP extensions to verify if their proposal is deployable.

TCP is an evolving protocol. Researchers regularly propose extensions to improve its performance [11]. Several of these extensions have been standardized by the IETF and are widely deployed [3]. Still, TCP extensions are usually not designed by taking middleboxes into account. As of this writing, only one TCP extension has been designed with middleboxes in mind : Multipath TCP [4]. Multipath TCP (MPTCP), allows a single data stream to be sent along different paths. This allows to achieve a higher throughput, as the resources of the paths are pooled together, and a better resilience to failures through vertical handover along different interfaces (e.g., from WiFi to 3G) [4].

In this paper, we propose a methodology to test TCP extensions behavior through known middleboxes and *MBtest* to support it. *MBtest* is a set of `Click` elements that model the main types of interference that can exist between TCP extensions and middleboxes. We use these elements to experimentally evaluate how TCP reacts to various types of middleboxes.

In contrast with previous work, we use the complete TCP and Multipath TCP stacks in the Linux kernel for our evaluation instead of relying on simplified models of TCP on top of raw sockets [10, 6]. We first use the TCP Selective Acknowledgment [8] and the TCP large window extension [7] to verify experimentally whether these extensions could be deployed today. Although Multipath TCP was designed with middleboxes in mind, based notably on the measurements described in [6], little is known of Multipath TCP implementation's abilities to correctly handle the middleboxes. Our experimental evaluation shows that our implementation of Multipath TCP in the Linux kernel can cope with most middleboxes except one corner case that was unforeseen in the Multipath TCP specification [4]. We complement

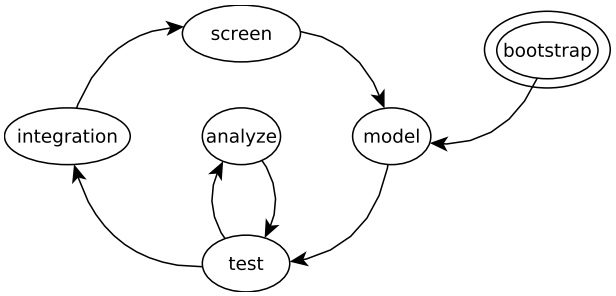

Figure 1: Proposed methodology

these lab experiments with a detailed measurement study of Multipath TCP with real applications in fifty different access networks. These measurements revealed some strange interactions between some `ftp` Application Level Gateways running on NATs and Multipath TCP.

This paper is structured as follows. Section 2 presents our testing methodology. Section 3 describes our `Click` elements that model middleboxes and how they interfere with TCP extensions. Section 4 analyzes the interactions between MultiPath TCP and real middleboxes. Section 5 presents the lessons we learned from these experiments.

2. METHODOLOGY

This section describes the iterative process we used, to verify whether TCP extensions are still middlebox-proof.

Figure 1 summarizes our methodology. The first cycle is a "pseudo" cycle that begins with a *bootstrap*. Thanks to prior knowledge about the today's middleboxes [9, 6, 2] on the Internet, we can start from a known list of middlebox-behaviors.

The *model* phase isolates the behaviors of the individual middleboxes and builds simple blocks that allow us to fully control the behavior of a given middlebox.

In the *test* phase these blocks and the composition of blocks are used to deploy simple and advanced tests for middleboxes in a fully controlled environment. The tests simulate a network and monitor the behavior of the transport protocols.

If an error occurs, we analyze the problem manually to understand the behavior of the transport protocol with respect to the middlebox. The goal is to pinpoint the problem with the protocol's implementation and (if possible) provide a solution. The test phase should not enter the integration phase before all problems are solved.

When the *integration* phase is reached, all the tests that have been made in the previous state should be integrated in a test suite that would allow one to do regression tests on the next developments.

At this point the first cycle is finished. All the future iterations will begin with the *screening* step. In this step, the protocol that we want to test is deployed on the Internet and should allow one to gather information about other middlebox behaviors. Once we find new middleboxes behavior, we can model them in the next step, etc.

3. TESTING THE DEPLOYABILITY OF TCP EXTENSIONS

The model and testing phase require the ability to run small middleboxes within a controlled environment. We chose to create *MBtest*[1], composed of a set of `click` elements that allow us to mimic middlebox behavior.

The middlebox functions identified during the bootstrap phase are implemented within the following `click` elements:

- **TCP sequence randomizer**. Some firewalls are known to randomize the TCP sequence numbers [6, 2] of passing TCP connections to prevent security problems with older TCP/IP stacks. Our randomizer is stateless. It increments the TCP sequence number of passing segments with a fixed value.
- **TCP window**. Middleboxes that perform traffic control may change the TCP window field in acknowledgements to force a given flow to slow down. Our `click` element adjusts the window field of the TCP header.
- **MSS option**. The MSS option is used in the three-way exchange to negotiate the maximum segment size. Our `click` element can change the value encoded in the MSS option to mimic some middleboxes that reduce the MSS for all TCP connections.
- **TCP option removal**. Some middleboxes remove TCP options [6]. Often, middleboxes remove a specific TCP option by replacing it with the NOP TCP option to avoid having to update the segment length. Our `click` element allows to remove a chosen TCP option from all segments or only from segments having some flags (e.g. SYN or FIN). It can also accept a given TCP option in the first n segments and discard it afterwards to model path changes.
- **Segment splitting**. Some middleboxes split large segments before transmitting them. The most widely deployed example are the high speed interfaces that support hardware offload. These network interfaces expose a large MSS value to the TCP stack and split the segments before transmitting them on the wire. Our `click` element is able to split a segment in two or at a given length. The element splits every segment if it contains enough data in the payload. Some segments contain TCP options. In this case, our `click` element can either copy the TCP options only in the first segment, only in the second or in both.
- **Segment Coalescing**. This `click` element does the opposite of the previous one. It coalesces two consecutive segments. If they contain options, the coalesced segment can either use as TCP options the ones that were in the first segment or in the second segment.

These `click` elements can be combined together to represent more complex middleboxes. *MBtest* is integrated in `netkit`[2] and contains a configuration language that allows to easily attach middleboxes on the forward or the backward path between two `netkit` hosts. *MBtest* also lets you choose the kernel that is running on the end-hosts within the simulated network. Additional details are available in *MBtest*'s documentation [5].

[1]Freely available at `https://bitbucket.org/bhesmans/mbtest`

[2]`http://www.netkit.org/`

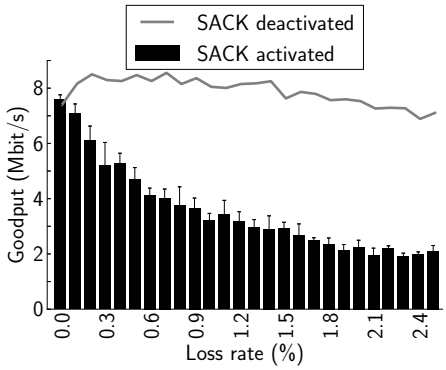

Figure 2: When SACK is enabled, the TCP sequence number randomizer introduces a large performance penalty compared to a TCP session without SACK.

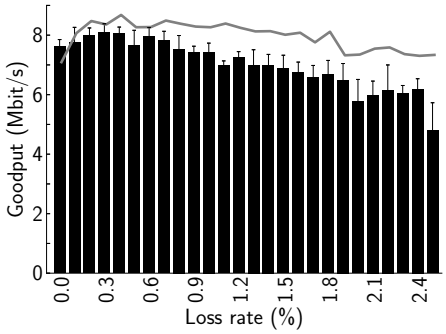

Figure 3: Accounting invalid SACK as duplicate acknowledgment brings benefits.

3.1 TCP SACK and middleboxes

The Selective Acknowledgement (SACK) option is defined in [8]. It allows a TCP host to advertise the reception of blocks of data separated by holes. This is done by placing in the SACK option a list of the first/last sequence numbers of each received block.

We tested the support of SACK by the unmodified Linux stack through *MBtest*. One test with the sequence number randomization revealed surprising results. When a TCP connection goes through a middlebox that randomizes the TCP sequence number, its performance drops. Figure 2 reports the goodput of a TCP connection between hosts connected to an emulated 10 Mbps network via *MBtest*. We compare the Linux TCP stack without SACK (gray line) with the Linux TCP stack where SACK has been enabled (black bars) on a path that includes a TCP sequence number randomizer. The horizontal axis reports the packet loss ratio and the vertical axis the TCP goodput. We varied the packet loss ratio from 0 to 2.5%. When there are no losses, the TCP sequence randomizer does not affect the goodput.

However, when the packet loss ratio increases, the performance of TCP drops when SACKs are used. This is counterintuitive since selective acknowledgements were designed to improve the reaction of TCP to packet losses. However, the sequence number randomizer updates the TCP sequence number field but does not change the SACK option. This implies that the receiver receives a SACK option that contains invalid sequence numbers. Today's Linux TCP stack considers that this invalid SACK block is an indication of an invalid segment and discards all the information included in this segment, including its acknowledgement number (which is valid). This implies that when there are losses, many TCP acknowledgements are discarded and TCP cannot perform a fast retransmit. It can only recover from the losses by relying on its retransmission timer, which slows down the goodput significantly. We proposed a patch to the Linux TCP maintainer to solve this problem[3]. We also performed similar experiments with Mac OSX (Mountain Lion) and found a similar performance decrease.

We run our experiments again after applying our patch. The results are presented on figure 3. With this patch, if an acknowledgment arrives with an invalid SACK option, it will be considered as a duplicate acknowledgment and the fast retransmit mechanism will be triggered. This change significantly improves the performance.

This case could be considered a bit academic since one would expect today's TCP sequence number randomizers to be aware of the TCP SACK option. Unfortunately, this is not true and our campus firewall suffered from this problem. Detal *et al.* [2] found in a recent measurement study that such sequence number randomizers are still widespread.

3.2 RFC1323 and middleboxes

We also used *MBtest* to study the interactions between the TCP large windows extension [7] and middleboxes. Although RFC1323 was published in the last century, there is anecdotal evidence that some deployed middleboxes still do not completely support this TCP extension. RFC1323 defines two different TCP extensions. The first one is the timestamp option. This option should not cause problems through middleboxes except that some of them are known to modify the passing timestamps [6]. The second option is used to support larger TCP windows. For this, the two end hosts store the receive window as a 32 bits integer. This enables the utilisation of large windows. However, the window field of the TCP header remains a 16 bits field. The WScale option in the SYN and SYN+ACK segments is used to negotiate a shift of the 32 bits receive window before encoding its low order 16 bits in the window field of the TCP header.

Our tests with *MBtest* revealed that this extension is robust to middleboxes that change the sequence number, split or coalesce segments. However, a middlebox that changes the TCP window field without understanding the negotiated WScale option would clearly cause problems. Our tests revealed another subtle problem when there are different middleboxes on the forward and the backward paths. Let's consider that the middlebox on the forward path accepts the WScale option while it is removed on the backward path. If the WScale option is removed from the SYN+ACK then the TCP state machines on the client and the server are desynchronized. The client assumes that the large window extension is not used for the connection (since it did not receive the WScale option) while the server still uses window scaling.

For example, consider a WScale option set to 8 by the client and the server, but removed from the SYN+ACK segment. If the server has an initial window of 14592 bytes, a typical value for the Linux TCP stack, it will encode a win-

[3] `http://marc.info/?l=linux-netdev&m=137694059706871&w=2`

#	Middlebox-function	Successfull?	Fallback?
1	NAT	Y (Yes)	N (No)
2	Remove opt. SYN	Y	Y
3	Remove opt. Data	Y	Y
4	Sequence number rand.	Y	N
	Segment Splitting		
5	Opt. both segment	Y	N
6	Opt. first segment	Y	N
7	Opt. second segment	Y	Y
8	Opt. second segment*	N	-
9	Coalesce	Y	Y

Table 1: MPTCP works across all common middleboxes, by either falling back to regular TCP, or thanks to a built-in support for this type of middlebox.

dow field of 57 in the TCP headers that it sends. Upon reception of this segment, the client considers this value as the window size in bytes ! It will only be able to send a 57 bytes segment causing a significant drop of performances.

3.3 Multipath TCP and middleboxes

Multipath TCP (MPTCP) is a major extension to TCP that allows to send a single data stream across multiple interfaces [4, 12]. MPTCP has been designed in such a way that it should work across today's Internet with all its middleboxes. To achieve the above, MPTCP opens multiple TCP subflows. Each subflow appears as a regular TCP connection to middleboxes. MPTCP multiplexes the data among these subflows and relies on TCP-options for signalling. During the design of MPTCP, its authors kept the middleboxes in mind and the specification includes several mechanisms to deal with different kinds of middleboxes. [4] These mechanisms allow MPTCP to realize that there is a middlebox on the path that modifies the segments in such a way that a safe operation of MPTCP is no more possible. In this case, MPTCP performs a fallback to regular TCP.

In order to validate the protocol mechanisms, we reproduce well known middlebox-behavior in *MBtest* and verify that the Linux Kernel implementation of MPTCP[4] v0.87 correctly handles this kind of middlebox. Table 1 shows the summary of the tests performed. The first column describes the type of middlebox used, the second column reports whether the connection successfully transmitted the data and the third column indicates whether MPTCP had to fallback to TCP. Each test uses two interfaces on the client, one interface on the server and 2 disjoint paths.

Middleboxes may remove unknown TCP options from the TCP header. As MPTCP uses a TCP option to signal the support of MPTCP to the peer, it may be affected by such a middlebox. MPTCP includes two mechanisms to deal with option removal. First, MPTCP places an MPTCP option in the SYN segments of the three-way handshake. This allows the end hosts to negotiate support for MPTCP, but also to detect middleboxes that remove unknown TCP options. In case of such a middlebox, MPTCP seamlessly falls back. We validated with *MBtest* that this case was handled correctly by the Multipath TCP implementation in the Linux kernel. However, this is not the only case where a middlebox that removes an MPTCP option may cause problems. Internet paths may be assymetric, or a middlebox may only remove a TCP option from non-SYN segments. In order to support this kind of asymmetric behavior, MPTCP is able to fallback , if a middlebox removes TCP options from non-SYN segments. For this, MPTCP requires that the first window worth of data contain the MPTCP option in each segment [4]. A host that does not receive an MPTCP option in the first data segment seamlessly falls back, thus stop using the MPTCP-options. After a fallback, the sender receives the first acknowledgment without an MPTCP option, which causes it to also fallback. We validate this behavior in *MBtest* by removing the MPTCP options only out of data segments (line 3, table 1).

[4]Available at `http://multipath-tcp.org`

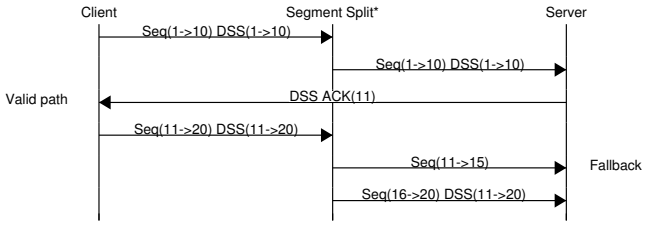

Figure 4: A very unlikely segment splitting scenario.

As MPTCP multiplexes data among different TCP subflows, the receiver must handle out-of-order segments. To reorder the data at the receiver, MPTCP uses a data-sequence number which is included in a TCP option (the DSS option). This option includes the mapping between the TCP sequence number and the data sequence number. To cope with sequence number randomizers, this mapping is relative to the beginning of the MPTCP connection. We verified with *MBtest* that the Linux implementation of MPTCP operates correctly through such middleboxes.

Middleboxes that perform segment splitting or coalescing have also been discussed during the design of MPTCP. A segment-splitting middlebox has three choices on where to set the TCP options of the original packet. It can either copy the options in both packets, in the first one or only in the second one. The first two cases work seamlessly for MPTCP. If the middlebox places the DSS option only on the second packet, MPTCP peforms a fallback, as the first data-segment does not contain a DSS option (see the explanation above).

A special case arises when only the second data segment is split (line 8 in Table 1 and depicted in Figure 4). This particular scenario has the following packet-sequence. The first data-segment is correctly acknowledged by the receiver, indicating to the sender that this path correctly supports Multipath TCP. However, the second data-segment is split by the middlebox. If the middlebox does not copy the DSS option in the first packet, the receiver will have to fallback to regular TCP. Thus, this scenario results in an inconsistent state between the sender and the receiver. it must be said that this scenario is very unlikely to happen in the real world.

Due to space limitations, we do not discuss the impact of segment coalescing. However, we can confirm that MPTCP works across such kind of middleboxes.

Finally, application-level gateways may modify the data stream by adding or removing bytes to the payload. Adding or removing bytes modifies the boundaries of the data-sequence mapping. Multipath TCP has to detect this and, fallback. This is achieved through an additional checksum over the payload which is included in every MPTCP segment. If an application-level gateway modifies the data-stream, this

checksum allows to detect this modification and trigger a fallback, allowing the data stream to proceed without any problems. The following section describes a particular case, where this checksum is vital to the correct operation of MPTCP.

4. MPTCP AND REAL MIDDLEBOXES

The screening phase of our proposed methodology consists in deploying the extension or protocol on the current Internet in order to detect yet unknown middlebox behaviors. Indeed, some of the deployed middleboxes, such as Deep Packet Inspections (DPI), firewalls or some application-level proxies might not have been considered during the bootstrapping phase. Only a real deployment can reveal these kind of middleboxes in order to integrate them into our list of middlebox behaviors.

The rest of this section is organized as follows. First, we show that by deploying MPTCP across the Internet, we detected odd and unmodeled middleboxes behavior. We then modeled those middleboxes in *MBtest* and investigate their impact on MPTCP.

4.1 Detecting new middleboxes

To evaluate the interactions between MPTCP and existing middleboxes we implemented a test suite inside a VirtualBox [1]. Our VirtualBox image contains an instrumented MPTCP kernel and several applications (`ftp` client, `http` client and `ssh` client). Although the VirtualBox is used on single-homed hosts, we configured it to use one, two or four subflows. We also modified the MPTCP kernel to use a different scheduler. The MPTCP scheduler is the algorithm that selects the TCP subflow over which each data segment is sent. The default scheduler always tries to send segments over the subflow that has both an open congestion window and the lowest round-trip time. We implemented two different schedulers. The *round-robin scheduler* sends segments over the established subflows in round-robin. This scheduler ensures that each subflow carries a part of the data stream. Thus, a DPI will only see part of the application-level protocol on each single TCP-session. The *duplicating scheduler* sends each segment over all established subflows. This scheduler allows us evaluate the possible impact of retransmissions.

Our VirtualBox includes several measurement scripts that use the various clients to interact with our MPTCP server. Our measurement script collects all packets sent and received by both the VirtualBox and our server. This enables us to detect whether middleboxes have modified TCP segments between the client and the server.

Colleagues and MPTCP users ran the VirtualBox in more than fifty different entreprises and access networks. By analyzing the collected packet traces, we discovered several unexpected interactions between MPTCP and middleboxes.

The SSH and HTTP(S) protocols were always working for both MPTCP and TCP. FTP however revealed interesting results. FTP uses two different types of connections. First, the control connection on port 21 allows to control the FTP session. Second, data connections are used to transfer files. The data connection can either be created by the client (passive mode) or by the server (active mode). In both modes, the host specifies to its peer the IP address and port number used for the data connection in the `PORT` command. If there is a NAT, it must include an ALG that modifies the data stream in order to replace the private IP address by the public one.

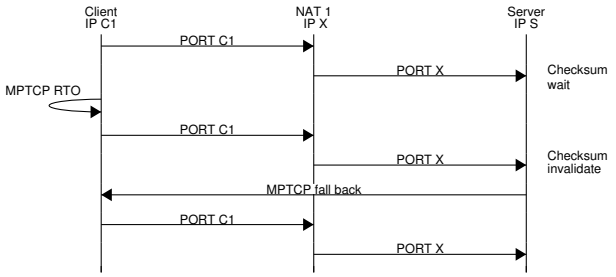

Figure 5: Active FTP using MPTCP with one NAT

We discovered that even with regular TCP, active FTP works only in 23 out of the 50 access networks. Many NAT devices are not able to correctly modify the IP address in the control connection, and thus do not allow FTP to operate correctly.

In the networks where active FTP over regular TCP worked, FTP over MPTCP succeeded in 86% of the cases. In the cases were MPTCP failed, the IP address contained in the `PORT` command was not properly modified by the NAT.

4.2 FTP in *MBtest*

To better understand the fallback mechanism in case of ALG we modeled such behavior in *MBtest* and performed experiments in a setup where a client is connected to a server through a single NAT.

The client uses FTP in active mode to transfer a file. In this configuration, we need to distinguish two cases; the "public" IP address of the NAT is such that the length (in characters) of the `PORT` command with regard to the original command emitted by the client is either smaller or larger/equal. As MPTCP includes the DSS-mapping, which allows to map the subflow sequence number to the data sequence number, changing the size of a packet affects the DSS mapping. If the translated `PORT` command is smaller, the mapping is incomplete. The MPTCP stack must wait for a subsequent data segment to fill the mapping and be able to verify the DSS checksum. If the translated `PORT` command is larger or equal, the mapping is complete and thus the DSS checksum can be verified immediately.

We conducted experiments with both cases :

smaller `PORT` *command.*

We collect the traces on both interfaces from the client and on the server side. The analysis of the traces show that the `PORT` command is sent 3 times. Figure 5 shows the exchange.

After the first `PORT` command reaches the server, MPTCP cannot validate the DSS checksum because the MPTCP mapping covers more data than received. MPTCP does not acknowledge the data at MPTCP level (but well at the subflow level) causing the client to wait for an MPTCP-level retransmission timeout before retransmitting the data. When received by the client it completes the MPTCP mapping from the first segment. As the payload changed, the checksum is invalid. Because this subflow is the only one and consequently the last one, MPTCP performs a fallback for this subflow. Finaly the client will retransmit a third `PORT`

command without MPTCP options that is correctly handled by the FTP server.

equal/larger `PORT` *command.*
In this case, as soon as the segment is received on the server, the data covers the mapping and the checksum can immediately be invalidated, triggering a fallback to regular TCP.

Instead of a single subflow, we may have n subflows. In this case each sublflow will sequentially receive an incomplete mapping for the `PORT` command separated by at least one MPTCP-level timeout. Then MPTCP will retransmit again on all subflows the same PORT command and it will complete each mapping. Each subflow, except the last one will be closed. And the last one will fallback. In this case MPTCP will need to retransmit the `PORT` command $2n + 1$ times before fallback. Moreover, if the last subflow that survives and fallback to TCP is not the initial subflow, `PORT` command carries an IP address that is not the one expected by the FTP server and it will respond with `ILLEGAL PORT`.

5. LESSONS LEARNED

During this detailed study of the interactions between TCP extensions and middleboxes, we've learned several lessons that are valid for any designer of TCP extensions. The first lesson is that a TCP extension cannot assume that one field of the TCP (or IP) header will never be modified *in-transit*. Changing the semantics of one field of the TCP header, like placing a shifted receive window instead of the entire receive window in the window field of the TCP header [7] is unsafe through middleboxes. If the large window extension had to be redesigned today, it would probably be necessary to place the entire receive window inside a TCP option. Furthermore, as shown by our SACK tests, assuming that the sequence number will not be modified is not safe. If Selective ACK had to be redesigned today, they would probably need to encode in the SACK blocks a delta from the beginning of the TCP connection. This solution, used by Multipath TCP to deal with sequence number randomization, appears to be safe. Second, a TCP extension cannot assume that a TCP option sent by a host will always be delivered to the other host. In a world with middleboxes, the negotiation performed during the three-way handshake is not anymore a two party negotiation. It becomes an *n-party* negotiation where n is not usually known. This makes the negotiation more complex. In particular, terminating the negotiation by the transmission of an option in the SYN+ACK segment is not safe as demonstrated by our tests with RFC1323. Multipath TCP solves this problem by placing an option in the third ACK. This should become the default for TCP extensions. Third, any middlebox that modifies the length of the payload may lead to some troubles. In regular TCP, if a middelbox wants to change the data length of a segment that follows an out-of-order packet, TCP sequence numbers may not be modified anymore because it would either create a hole or an overlap in sequence number. In the case of MPTCP, any change of the payload length will invalidate the MPTCP mappings present in DSS. Moreover, if the data have been shrank, MPTCP may need several MPTCP-level timeouts to detect the issue because the receiver will have to wait for each subflow to complete the mapping and compute the checksum. It is also worth to note that DPI may be harder with MPTCP because it is not always possible to reassemble the entire stream.

Our measurements with *MBtest* and across the Internet have shown that Multipath TCP is able to cope with various types of middleboxes.

6. ACKNOWLEDGMENTS

This work is partially funded by the European Commission funded CHANGE (INFSO- ICT-257422) projects and the IAP-BESTCOM.

7. REFERENCES

[1] Virtualbox. https://www.virtualbox.org.
[2] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet. Revealing Middlebox Interference with Tracebox. In *ACM/USENIX Internet measurement conference (IMC)*, 2013.
[3] M. Duke, R. Braden, W. Eddy, and E. Blanton. A Roadmap for Transmission Control Protocol (TCP). RFC4614, Sept. 2006.
[4] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC6824, Jan. 2013.
[5] B. Hesmans. Mbtest. Technical report, 2013.
[6] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *ACM/USENIX Internet measurement conference (IMC)*, pages 181–194. ACM, 2011.
[7] V. Jacobson, B. Braden, and D. Borman. TCP Extensions for High Performance. RFC1323, May 1992.
[8] M. Mathis, J. Mahdavi, S. Floyd, and R. Ally. TCP Selective Acknowledgment Options. RFC2018, Oct. 1996.
[9] A. Medina, M. Allman, and S. Floyd. Measuring Interactions between Transport Protocols and Middleboxes. In *SIGCOMM'04*, pages 336–341. ACM, 2004.
[10] J. Padhye and S. Floyd. Identifying the TCP behavior of web servers. In *ACM SIGCOMM'00*, 2000.
[11] C. Raiciu, J. Iyengar, and O. Bonaventure. Recent advances in reliable transport protocols. In *SIGCOMM ebook on Recent Advances in Networking*, 2013.
[12] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? Designing and Implementing a Deployable Multipath TCP. In *USENIX Networked Systems Design and Implementation (NSDI)*, 2012.
[13] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.
[14] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *SIGCOMM'12*, pages 13–24, 2012.
[15] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An Untold Story of Middleboxes in Cellular Networks. In *SIGCOMM'11*, pages 374–385, 2011.