Is it still possible to extend TCP?

Michio Honda[*], Yoshifumi Nishida[*], Costin Raiciu[†], Adam Greenhalgh[‡], Mark Handley[‡], Hideyuki Tokuda[*]
Keio University[*], Universitatea Politehnica Bucuresti[†],University College London[‡]
{micchie,nishida}@sfc.wide.ad.jp, costin.raiciu@cs.pub.ro
{a.greenhalgh,m.handley}@cs.ucl.ac.uk, hxt@ht.sfc.keio.ac.jp

ABSTRACT

We've known for a while that the Internet has ossified as a result of the race to optimise existing applications or enhance security. NATs, performance-enhancing-proxies, firewalls and traffic normalizers are only a few of the middleboxes that are deployed in the network and look beyond the IP header to do their job. IP itself can't be extended because "IP options are not an option" [9]. Is the same true for TCP?

In this paper we develop a measurement methodology for evaluating middlebox behavior relating to TCP extensions and present the results of measurements conducted from multiple vantage points. The short answer is that we can still extend TCP, but extensions' design is very constrained as it needs to take into account prevalent middlebox behaviors. For instance, absolute sequence numbers cannot be embedded in options, as middleboxes can rewrite ISN and preserve undefined options. Sequence numbering also must be consistent for a TCP connection, because many middleboxes only allow through contiguous flows.

We used these findings to analyze three proposed extensions to TCP. We find that MPTCP is likely to work correctly in the Internet or fall-back to regular TCP. TcpCrypt seems ready to be deployed, however it is fragile if resegmentation does happen - for instance with hardware offload. Finally, TCP extended options in its current form is not safe to deploy.

1. INTRODUCTION

The Internet was designed to be extensible; routers only care about IP headers, not what the packets contain, and protocols such as IP and TCP were designed with options fields that could be used to add additional functionality. The great virtue of the Internet was always that it was *stupid*; it did no task especially well, but it was extremely flexible and general, allowing a proliferation of protocols and applications that the original designers could never have foreseen.

Unfortunately the Internet, as it is deployed, is no longer the Internet as it was designed. IP options have been unusable for twenty years[9] as they cause routers to process packets on their slow path. Above IP, the Internet has benefited (or suffered, depending on your viewpoint) from decades of optimizations and security enhancements. To improve performance [2, 7, 16, 3], reduce security exposure [13, 28], enhance control, and work around address space shortages [21], the Internet has experienced an invasion of middleboxes that *do* care about what the packets contain, and perform processing at layer 4 or higher *within* the network.

The problem now faced by designers of new protocols is that there is no longer a well defined or understood way to extend network functionality, short of implementing everything over HTTP[18]. Recently we have been working on adding both multipath support[11] and native encryption[5] to TCP. The obvious way to do this, in both cases, is to use TCP options. In the case of multipath, we would also like to stripe data across more than one path. At the end systems, the protocol design issues were mostly conventional. However, it became increasingly clear that no one, not the IETF, not the network operators, and not the OS vendors, knew what will and what will not pass through all the middleboxes as they are currently deployed and configured. Will TCP options pass unchanged? If the sequence space has holes, what happens? If a retransmission has different data than the original, which arrives? Are TCP segments coalesced or resegmented? These and many more questions are crucial to answer if protocol designers are to extend TCP in a deployable way. Or have we already lost the ability to extend TCP, just like we did two decades ago for IP?

In this paper we present the results from a measurement study conducted from 142 networks in 24 countries, including cellular, WiFi and wired networks, public and private networks, residential, commercial and academic networks. We actively probe the network to elicit middlebox responses that violate the end-to-end transparency of the original Internet architecture. We focus on TCP, not only because it is by far the most widely used transport protocol, but also because while it is known that many middleboxes modify TCP behavior [6], it is not known how prevalent such middleboxes are, nor precisely what the *emergent behavior* is with TCP extensions that were unforeseen by the middlebox designers.

We make three main contributions. The first is a snapshot of the Internet, as of early 2011, in terms of its transparency to extensions to the TCP protocol. We examine the effects of middleboxes on TCP options, sequency numbering, data acknowledgment, retransmission and resegmentation.

The second contribution is our measurement methodology and tools that allow us to infer what middleboxes are doing to traffic. Some of these tests are simple and obvious; for example, whether a TCP option arrives or is removed is easy to measure, so long as the raw packet data is monitored at both ends. However, some tests are more subtle; to test if a middlebox coalesces segments it is not sufficient to just send many segments — unless the middlebox has a reason to queue segments it will likely pass them on soon as they arrive, even if it has the capability to coalesce. We need to force it to have the opportunity to coalesce.

Finally we examine the implications of our measurement study for protocol designers that wish to extend TCP's functionality. In particular, we look at proposals for Multipath TCP[11], TcpCrypt[5], and TCP Extended Options[8], and consider what our findings mean for the design of these protocols and their deployability.

The reminder of this paper is organized as follows: Sec. 2 describes related work; in Sec. 3 we describe our methodology and introduce the TCPExposure tool, our tool to inspect middleboxes behavior; in Sec. 4 we examine middlebox behavior on each protocol component in more detail, show how to detect this behavior, then present our measurement results from running TCPExposure in 142 networks; in Sec. 5 we examine the impact on TCP extensions as case-study. We summarise our conclusions in Sec. 6.

2. RELATED WORK

There exists a large body of work related to the measurement, analysis and identification of different deployed TCP implementations, but none of it has specifically focused on analyzing TCP middlebox behaviour.

Padhye and Floyd perform a client-side analysis of numerous public web servers to test their congestion control behavior and ECN and SACK capabilities [22]. The client-only methodology leverages existing public web servers to give great coverage, allowing the authors to examine the behavior of many different TCP implementations.

The study focuses on remote TCP implementations rather than middlebox interactions; the same methodology is not applicable for this middlebox study for three reasons. First, most users access the Internet through home and cellular networks, yet few public servers exist in these networks that could be used for tests. Further, it is not possible to test qualitative middlebox behavior without co-ordination of both end systems. Finally, the Padhye and Floyd techniques cannot distinguish the effects due to middleboxes from the particularities of remote TCP implementations and remote hardware (such as segmentation offload).

Medina *et al.* measure in their 2005 study the impact of network middleboxes on path MTU discovery transparency, sequence number shifting, as well as their effect on IP and TCP options [20]. This study undertakes similar client-only measurements as in [22], and suffers from the same limitations.

Allman [1] and Hätönen *et al.* [14] both examine the quantitative application-level performance of various middleboxes in testbeds where the box being tested is known and under their control. Allman measures transaction delay, throughput and connection persistence over the middleboxes he evaluated. Hätönen *et al.* measure NAT binding timeouts, queueing delays, throughput and support of new transport protocols over their testbed which includes a large number of home-gateway devices. We adopt the end-to-end methodology of these papers and extend it further to examine the qualitative middlebox behavior that we are interested in in the real Internet.

Paxon measures end-to-end packet dynamics such as out-of-order delivery, packet corruption and retransmission on TCP bulk transfers [23]. The author operates both end systems of each end-to-end measurement by remote login; this limits the applicability of the study to networks where the authors have (or are given temporary) direct access to hosts. This poses two challenges: first, obtaining shell access to users' machines to run privileged commands is really difficult; second, even if permitted, accessing NATed boxes is not possible unless users specifically open up NAT ports. To avoid these issues we adopted the alternative approach of asking contributors to run a single, publicly-available, shell script and to upload the results.

3. METHODOLOGY AND DATASETS

We use regular end-hosts to actively measure paths in the Internet. Our aim is to test relevant properties that could impact yet-to-be-deployed TCP extensions. We have resorted to active measurement for a number of reasons:

- We need to generate traffic that mimics new TCP extensions.
- We generate artificial traffic patterns such as contiguous small segments or gaps in the sequence space. It is difficult to use passive measurements for this purpose.
- Packets need to be inspected at both sender and receiver for tests detecting TCP option removal, sequence number shifting, re-segmentation, etc.

Table 1: Default TCP Parameters

Parameter	Initiator	Responder
Initial Sequence Num (ISN)	252001	11259375
Window Size	8064	32768
MSS	512	512
Window Scale	-	6
SACKOK	-	1
Timestamp (TS_val)	-	12345678

- We need to test different destination ports including ports not normally in use, as middlebox behavior depends on the destination port.

3.1 Testing Tool

Our middlebox inspection tool is called **TCPExposure** and consists of an initiator and a responder tool. These are a 3000-line program and a 500-line program both written in Python. The initiator (acting like a TCP client) and the responder (the TCP server) run tests aiming to trigger on-path middlebox actions. The tools send and receive TCP segments in user space via a raw IP socket or using the Pcap library similarly to Sting [25].

The client tool was built to be easy to use, as most of our tests are run by contributors. To maximize reach, the client tool is cross-platform running on Mac OS, Linux and FreeBSD. It is self-contained and only requires Python and libpcap on the host; these come pre-installed on most systems. The client is straightforward to run: all users need to do is to download it, run it and post the results.

The responder tool runs on a Linux server we control. It does not maintain state for the TCP connections it is emulating; its replies depend solely on the received TCP segments. For example, the responding segment contains SYN/ACK if the responder has received SYN, acknowledges the end of the sequence number, and has the sequence number based on the received acknowledgement (ACK) number. This stateless behavior makes it relatively easy to reason about observed behavior because there is no hidden server state.

3.2 Common Procedures

Table 1 lists the fixed TCP parameters at the initiator and the responder. These values are used in all our measurements unless stated otherwise.

We use a 512 byte MSS at both ends, less than what most TCP implementations advertise. This value is smaller than the MTU of most Internet paths, and was chosen to avoid unexpected fragmentation during tests.

We expect middleboxes to behave differently depending on the application type, and so our responder emulates TCP servers on ports 80, 443, and 34343. Ports 80 and 443 are assigned by IANA for http and https traffic; port 34343 is unassigned. The client port is randomly chosen at connection setup.

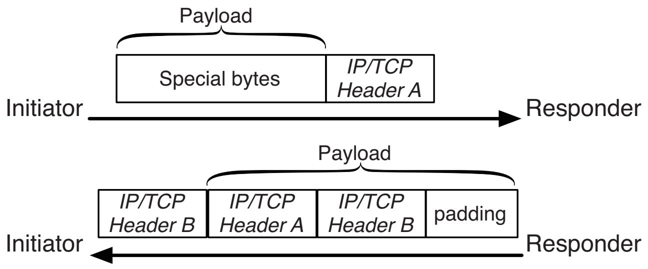

Figure 1: Echo Headers Command

Segments sent from the initiator include commands to operate the responder. The default command is "just ack", and the responder sends back a pure ACK (no data). Another command is "echo headers". Fig. 1 illustrates how this command works.

The initiator transmits a crafted segment that includes bytes indicating this command in its payload. The responder replies with a segment that contains in its payload both the received headers and the headers of the reply. The client then compares the sent and received headers for both segments to detect middlebox interference. The last command is "don't advance ack". The responder does not advance the ACK number when it receives this command; instead it sends back an ACK with the first sequence number of the receiving segment. This command is used in only the retransmission test in Sec. 4.5.

3.3 Measurement Data

Our measurements target access networks, where ISPs deploy middleboxes to optimize various applications with the goal of improving the experience of the majority their customers. The core is mostly just doing "dumb" packet forwarding. Many contributors and we ran the TCPExposure client in a variety of access networks detailed below. We ran the server tool (the responder) in *sfc.wide.ad.jp*, a middlebox-free network.

From 25th September 2010 to 30th April 2011, we measured 142 access networks in 24 countries. Table 2 shows the venues and the network types of the experiments.

Access networks are categorized in six types. Home networks consisting of a consumer ISP and a home-gateway are labeled as *Home*. Public hotspots for example in cafes, airports, hotels, and conference halls are labeled as *Hotspot*. Mobile broadband networks such as 3G and WiMAX are labeled as *Cellular*. Networks in universities are labeled as *Univ.* We count two different networks (e.g., the lecture and the residence segments)

Table 2: Experiment Venues

Country	*Home*	*Hotspot*	*Cellular*	*Univ*	*Ent*	*Hosting*	Total
Australia	0	2	0	0	0	1	3
Austria	0	0	0	0	1	0	1
Belgium	4	0	0	1	0	0	5
Canada	1	0	1	0	1	0	3
Chile	0	0	0	0	1	0	1
China	0	7	0	0	0	0	7
Czech	0	2	0	0	0	0	2
Denmark	0	2	0	0	0	0	2
Finland	1	0	0	3	2	0	6
Germany	3	1	3	4	1	0	12
Greece	2	0	1	0	0	0	3
Indonesia	0	0	0	3	0	0	3
Ireland	0	0	0	0	0	1	1
Italy	1	0	0	0	1	0	2
Japan	19	10	7	3	2	0	41
Romania	1	0	0	0	0	0	1
Russia	0	1	0	0	0	0	1
Spain	0	1	0	1	0	0	2
Sweden	1	0	0	0	0	0	1
Switzerland	2	0	0	0	0	0	2
Thailand	0	0	0	0	2	0	2
U.K.	10	4	4	2	1	1	22
U.S.	3	4	4	0	4	2	17
Vietnam	1	0	0	0	1	0	2
Total	49	34	20	17	17	5	142

in the same university as two university networks. Enterprise networks (also including small offices) are labeled as *Ent.* Networks in hosting services are labeled as *Hosting.*

4. TESTS AND RESULTS

4.1 TCP Option Tests

TCP Options are the intended mechanism by which TCP can be extended. Standardized and widely implemented options include Maximum Segment Size (MSS), defined in 1981; Window Scale, defined in 1988; Timestamp, defined in 1992; and Selective Acknowledgment (SACK), defined in 1996. IANA also lists TCP options defined since 1996, but SACK is the most recently defined option in common use, and predates almost all of today's middleboxes. The question we wish to answer is whether it is still possible to rapidly deploy new TCP functionality using TCP options by upgrades purely at the end systems.

Unknown TCP options are ignored by the receiving host. A TCP extension typically adds a new option to the SYN to request the new behavior. If the SYN/ACK carries the corresponding new option in the response, the new functionality is enabled. Middleboxes have the potential to disrupt this process in many ways, preventing or at least delaying the deployment of new functionality.

If a middlebox simply removes an unknown option from the SYN, this should be benign - the new functionality fails to negotiate, but otherwise all is well. However, removing an unknown option from the SYN/ACK may be less benign - the server may think the functionality is negotiated, whereas the client may not. Removing unknown options from data packets, but not removing them from the SYN or SYN/ACK would be extremely problematic: both endpoints would believe the negotiation to use new functionality succeeded, but it would then fail. Finally, any middlebox that crashes, fails to progress the connection, or explicitly resets it would cause significant problems.

To distinguish possibly problematic behaviors, we performed the following tests:

1. **Unknown option in SYN.** The SYN and SYN/ACK segments include an unregistered option.
2. **Unknown option in Data segment**. The test includes unknown options in data segments sent by client and server.
3. **Known option in Data Segment.** The test includes a well-known option in data segments sent by client and server.

All three tests are performed using separate connections. Test 3 is included to allow us to determine whether it is the unknown nature of the option that causes a behavior, or just any option. For test 1, we use a draft version of the MP_CAPABLE option for MPTCP [10] and for test 2 we use MPTCP's MP_DATA option; neither is currently registered with IANA, and no known middlebox yet supports them. On receipt of a SYN with MP_CAPABLE, our responder returns a SYN/ACK also containing MP_CAPABLE, mimicking an MPTCP implementation.

For test 3, we used the TIMESTAMP option [15], which is not essential to TCP's functionality, but which is commonly seen in TCP data segments. This option elicits a response from the remote endpoint; a stateful middlebox may also respond, allowing us to identify such middleboxes.

In the *unknown option in SYN test*, our code tests for the following possible middlebox behaviors:

- SYN is passed unmodified.
- SYN containing the option is dropped.
- SYN is received, but option was removed.
- Option is received, but with a modified value.
- Option is received, but with a zeroed value.
- Connection is reset by the middlebox.

In the *unknown* and the *known option in data* tests, we test for the same behaviors as in the SYN test. After a normal handshake, we transmit a full-sized TCP

Table 3: Unknown Option in Syn

Observed	TCP Port		
Behavior	34343	80	443
Passed	129 (96%)	122 (86%)	133(94%)
Removed	6 (4%)	20 (14%)	9 (6%)
Changed	0 (0%)	0 (0%)	0 (0%)
Error	0 (0%)	0 (0%)	0 (0%)
Total	135 (100%)	142 (100%)	142 (100%)

Table 4: Known Option in Data

Observed	TCP Port		
Behavior	34343	80	443
Passed	129 (96%)	122 (86%)	133 (94%)
Removed	6 (4%)	9 (6%)	6 (4%)
Changed	0 (0%)	4 (3%)	3 (2%)
Error	0 (0%)	7 (5%)	0 (0%)
Total	135 (100%)	142 (100%)	142 (100%)

Table 5: Unknown Option in Data

Observed	TCP Port		
Behavior	34343	80	443
Passed	129 (96%)	122 (86%)	133(94%)
Removed	6 (4%)	13 (9%)	9 (6%)
Changed	0 (0%)	0 (0%)	0 (0%)
Error	0 (0%)	7 (5%)	0 (0%)
Total	135 (100%)	142 (100%)	142 (100%)

segment including MP_DATA or TIMESTAMP, using the method described in Sec. 3.2 to identify what the responder received. We also look for middleboxes that split the connection, processing the Timestamp at the middlebox on either the inbound or outbound leg.

Middlebox Behavior on TCP Options

Tables 3-5 summarize the results of the options tests. 142 paths were tested in total; for ports 80 (http) and 443 (https), we obtained results from all paths for all tests. However seven paths did not pass the unregistered port 34343, even with regular TCP SYN segments. These paths appear to run strict firewall rules allowing only very basic services.

Most of the paths we tested passed both known and unknown TCPs options without interference, both on SYN and Data packets. The results are port-specific though; 96% of paths passed options on port 34343, whereas only 80% of paths passed options on port 80. This agrees with anecdotal evidence that http-specific middleboxes are relatively common.

All the paths which passed unknown options in the SYN also passed both known and unknown options in data segments. In the tables, the "*Removed*" rows indicate that packets on that path arrive with the option removed from the packet. For the unknown options in the SYN packet, this was the only anomaly we found; no path modified the option or failed to deliver the packet due to its presence. In addition, all the paths which passed the unknown option in the SYN also passed unknown options in data segments. This bodes well for deployability of new TCP options - testing in the SYN and SYN/ACK is sufficient to determine that new options are safe to use throughout the connection.

Our test did not distinguish between middleboxes that stripped options from SYNs and those that stripped options from SYN/ACKs. With hindsight, this was an unfortunate limitation of our methodology that uses a stateless responder. However it is clear that any extension using TCP options to negotiate functionality should be robust to stripped unknown options in SYN/ACK packets, even if they are passed in SYNs. If it is crucial that the server knows whether or not the client received the option in the SYN/ACK, the protocol must take this into account. For example, TcpCrypt requires that the first non-SYN packet from the client contains the INIT1 option - if this is missing, TcpCrypt moves to the disabled state and falls back to regular TCP behavior.

For port 34343, the only behaviors seen were passing or removing options. The story is more complicated for port 80 (http) and 443 (https). First, there were seven paths that did not permit our testing methodology on port 80. In data packets our stateless server relies on instructions embedded in the data to determine its response. These seven paths appear to be application-level HTTP proxies, and we were foiled by the lack of a proper HTTP request in our data packets. They are labeled *Error* in the tables. We were able to go back and manually verify two of these paths were in fact HTTP proxies; we did not get a second chance to verify the other five. All seven were in the set that removed options from SYN packets, which is to be expected if they are full proxies.

There were no other unexpected results with unknown options, but we did observe some interesting results with the TIMESTAMP "known option in data" test. Four paths passed on a TIMESTAMP option to the responder, but it was not the one sent by the initiator. In these cases, although the responder sent TIMESTAMP in response, this was not returned to the initiator. This implies that the middlebox is independently negotiating and using timestamp with the server. These paths are labeled "*Changed*" in the tables.

Returning to the middleboxes that remove unknown options from the SYN, we can use the results of additional tests to classify these into two distinct categories. In the first category, the SYN/ACK received is essentially that sent by the responder, whereas in the second the SYN/ACK appears to have been generated by the middlebox. In Sec. 4.4 we explain how fingerprints in the SYN/ACK let us distinguish the two. Paths in the

Table 6: Types of removal behavior (SYN)

Path Type	Other Observed Effects	TCP Port 34343	80	443
Elim.	None	5	4	5
Proxy	Proxy SYN-ACK	1	16	4
Total		6	20	9

Table 7: Types of removal behavior (Data)

Path Type	Other observed effects	TCP Port 34343	80	443
Elim.	None	5	4	5
Proxy	Proxy Data ACK, Segment Caching, Re-segmentation	1	9	4
Total		6	13	9

Table 8: Option removal by Network Type

	Remove option (Proxy conn)		
Network Type	port 34343	port 80	port 443
Cellular (out of 20)	4 (1)	8 (6)	4 (1)
Hotspot (out of 34)	1 (0)	6 (5)	4 (3)
Univ (out of 17)	0 (0)	3 (3)	0 (0)
Ent (out of 17)	1 (0)	3 (2)	1 (0)
Total	6	20	9

Table 9: Sequence Number Modification Test

	TCP Port		
Behavior	34343	80	443
Unchanged	126 (93%)	116 (82%)	128 (90%)
Mod. outgoing	5 (4%)	5 (4%)	6 (4%)
Mod. incoming	0 (0%)	1 (1%)	1 (1%)
Mod. both	4 (3%)	13 (9%)	7 (5%)
Proxy (probably mod. both)	0 (0%)	7 (5%)	0 (0%)
Total	135 (100%)	142 (100%)	142 (100%)

first category appear to actively eliminate options (we label them "*Elim*" in Table 6), whereas a middlebox in the second category is acting as a proxy, and unknown options are removed as a side effect of this proxy behavior (these are labeled "*Proxy*").

These two categories (*Elim* and *Proxy*) also hold when we look at data segments (see Table 7). Paths that eliminate SYN options also eliminate data options, whereas paths that show proxy behavior on SYNs also exhibit proxy behavior for data. In particular, the proxy symptoms we see are Proxy Data ACKs (ACK by the middlebox, see Sec. 4.4), segment caching (the middlebox caches and retransmit segments, see Sec. 4.5), and resegmentation (splitting and coalescing of segments, see Sec. 4.6). These proxy middleboxes show symptoms of implementing most of the functionality of a full TCP stack, rather than just being a packet-level relay.

Before we ran this study, anecdotal evidence had suggested that cellular networks would be much more restrictive than other types of network. The results partially support this, as shown in Table 8. For port 80, eight out of 20 cellular networks that we tested remove options; six of the eight proxy the connection. WiFi hotspots are also relatively likely to remove options or proxy connections, especially for http. Overall though, the majority of paths do still pass new TCP options.

We conclude that it is still possible to extend TCP using TCP options, so long as the use of new options is negotiated in the SYN exchange, and so long as fallback to regular TCP behavior is acceptable. However, if we want ubiquitous deployment of a new feature, the story is more complicated. Especially for http, there are a significant number of middleboxes that proxy TCP sessions. For middleboxes that eliminate options, it seems likely that very simple updates or reconfiguration would allow a new standardized option to pass, assuming it were not considered a security risk. But for transparent proxies, the middlebox would not only need to pass the option, but also understand its semantics. Such paths are likely to be more difficult to upgrade.

4.2 Sequence Number Modification

TCP Selective Acknowledgement (SACK) [19] is an example of a TCP extension that uses TCP options that quote sequence numbers, in this case to indicate precisely which segments arrived at the receiver. How might middleboxes affect such extensions?

In our sequence number modification test, we examine both the outgoing and incoming initial sequence number (ISN) to see whether middleboxes modify the sequence numbers sent by the end-systems.

As table 9 shows, sequence numbers on at least 80% of paths arrive unchanged. However 7% of paths modify sequence numbers in at least one direction for port 34343 and 18% modify at least one direction for TCP port 80. For port 80, the same seven paths identified earlier as having application-level HTTP proxies cannot be tested outbound, but do modify inbound sequence numbers and almost certainly modify both directions.

One might reasonably expect that middleboxes that proxy a connection would split a TCP connection into two sections, each with its own sequence space, but that other packet-level middleboxes would have no reason to modify TCP sequence numbers. If this were the case, then TCP extensions could refer to TCP sequence numbers in TCP options, safe in the knowledge that either the option would be removed in the SYN at a proxy, or sequence numbers would arrive unmodified. Unfortunately the story is not so simple.

At a TCP receiver, one use of sequence numbers is to verify the validity of a received segment. If an adversary can predict the TCP ports a connection will use, only the randomness of the initial sequence number prevents a spoofed packet from being injected into the connection. Unfortunately TCP stacks have a long history of generating predictable initial sequence numbers, so a number of firewall products try to help out by choosing a new more random ISN, and then rewriting all subsequent packets and acknowledgments to maintain consistency [13, 28].

We compared those paths that pass unknown options in the SYN with those that modify sequence numbers in at least one direction. On port 34343, 5 out of 9 allow unknown options and still modify the sequence numbers. For port 80, 7 out of 26 pass unknown options, and for port 443 it is 7 out of 14. The numbers are the same for unknown options in data packets.

We conclude that it is *unsafe* for TCP extensions to embed sequence numbers in TCP options (or anywhere else), even if the extension negotiates use via a new option in the SYN exchange.*

4.3 Sequence Space Holes

TCP is a reliable protocol; its cumulative Ack does not move forwards unless all preceding segments have been received. What would happen if from the vantage point of a middlebox, a TCP implementation violated these rules? Perhaps it wished to implement partial reliability analogous to PR-SCTP [27], or perhaps it simply stripes segments across more that one path in a similar manner to Multipath TCP?

We can distinguish two ways a middlebox might observe such a hole:

- **Data-First:** it sees segments before and after a hole, but does not see the segment from the hole. If the middlebox passes the segment after the hole, it sees it cumulatively acked by the recipient, despite the middlebox never seeing the data from the hole.

- **Ack-First:** It sees a segment of data, then an ack indicates the receiver has seen data not yet seen by the middlebox. If the middlebox passes the Ack, the next segment seen continues from the point acked, leaving a hole in the data seen by the middlebox.

These form the basis of our tests shown in Fig. 2.

Table 10 shows the result of the data-first sequence hole test. Paths where the second Ack was correctly

*SACK does embed sequence numbers in options, but it predates the existence of almost all middleboxes. We hope that these middleboxes are aware of SACK and either rewrite the options or explicitly remove SACK negotiation from the SYN exchange.

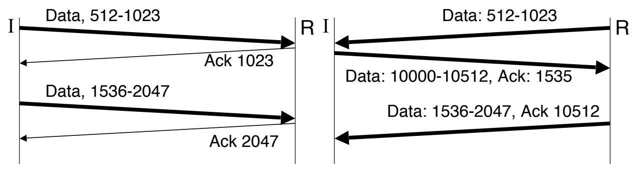

Figure 2: Sequence Hole Tests: data first (left) and ack first (right)

Table 10: Data-First Sequence Hole Test

	TCP Port		
Behavior	34343	80	443
Passed	131 (97%)	120 (85%)	135 (95%)
No response	2 (1%)	6 (4%)	2 (1%)
Dup Ack	1 (1%)	9 (6%)	5 (4%)
Test Error	1 (1%)	7 (5%)	0 (0%)
Total	135 (100%)	142 (100%)	142 (100%)

received are labeled *Passed*, and clearly have no middlebox that requires TCP flow reassembly. As before, on port 80 there are seven paths with http proxies we cannot fully test; these are labeled *Test Error*. The one path using port 34343 labeled *Test Error* was due to high packet loss during the experiment rather than middlebox interference.

The remaining cases are the most interesting. We observed two distinct middlebox behaviors:

- *No response* was received to the second data packet.

- A *duplicate Ack* was received, indicating receipt of the first data packet and by implication, signaling loss of the packet in the hole.

A middlebox implementing a full TCP stack would be expected to break the path into two sections, separately acking packets from the initiator before sending the data on to the responder. This would give the *Duplicate Ack* behavior. As expected, we see more such middleboxes on port 80.

A middlebox that does not respond to the second packet is clearly maintaining TCP state (or it would pass the second Ack), but it is not independently acking data. Its reasons for doing so are unclear - perhaps it is attempting to analyse the stream contents and is unwilling to pass an ack for data it has not seen? Whatever the reason, we still see more such middleboxes on port 80.

In the ack-first sequence hole test (Fig. 2, right), the initiator acks a segment beyond that which is received (i.e., proactive ack). The responder skips the data acked and sends additional data following on from the point that was acked. To allow us to send commands to the responder, the segments from the initiator to the responder also contain data, but what we are interested

Table 11: *Ack-first Sequence Hole Test*

Behavior	TCP Port 34343	80	443
Passed	102 (76%)	95 (67%)	105 (74%)
No response	28 (21%)	28 (20%)	29 (20%)
Ack fixed	5 (4%)	11 (8%)	7 (5%)
Retransmitted	0 (0%)	1 (1%)	1 (1%)
Test Error	0 (0%)	7 (5%)	0 (0%)
Total	135 (100%)	142 (100%)	142 (100%)

in is whether the proactive ack is received, and subsequently whether the packet following the hole is received. Table 11 shows the results.

The results of this test were a surprise - even on port 34343, middleboxes interfered with end-to-end behavior 24% of the time. As before, seven paths on port 80 could not be tested. Of those that could be tested, we saw three distinct behaviors:

- On around 20% of paths we saw *no response* to the proactive ack. Either the proactive ack was dropped or the packet above the hole was dropped, but the lack of a response does not allow us to distinguish.
- On quite a few paths (labeled *Ack fixed*), the proactive ack was re-written by the outgoing middlebox to indicate the highest data cumulatively seen by the middlebox.
- On one path on ports 80 and 443, the middlebox itself actually *retransmitted* the last packet sent by the responder from before the hole.

It is clear from these results that TCP extensions relying on sequence number holes are *unsafe*. Although some of the results can be explained by proxy behavior at middleboxes, some paths that did not exhibit clear proxy behavior (by performing separate acknowledgment) do affect both sequence holes and pro-active acking. Perhaps some firewalls attempt to protect the initiator from potentially malicious proactive acks? [26].

One interesting observation is that around 10% of home networks give *no response* in the ack-first sequence hole test. This is striking because none of the home networks strip unknown options.

4.4 Proxy Acknowledgments

In Tables 6 and 7 we observed that a subset of the paths that remove TCP options appear to show TCP proxy behavior. We now elaborate on the tests we used to elicit this information.

A hypothetical TCP proxy[2] would likely split the TCP connection into two sections; one from the client to the proxy and one from the proxy to the server. Each section would effectively run its own TCP session, with only payload data passed between the two sections. Are the proxies we observed of this form, which is fairly easy to reason about, or is their behavior more complex?

One symptom of a TCP proxy would be that acknowledgments for data are locally generated by the middlebox. We performed two tests examining this behavior:

- **Proxy SYN-ACK:** Is the SYN/ACK locally generated by the proxy? In its SYN/ACKs, our responder generates quite characteristic values for the initial sequence number, advertised receive window, maximum segment size, and Window Scale options. It is improbable that a proxy would generate these values. We simply check the value of these fields in the SYN/ACK received by the initiator - if they differ then this is symptomatic of a proxy that crafts its own SYN/ACKs.
- **Proxy Data Ack:** Is data acknowledged by the proxy before delivering it to the destination? Our initiator sends a data packet to the responder, requesting the ack is sent on a packet that includes data. If the ack received does not include data, it is extremely likely it was generated by the proxy rather than the responder.

Neither test is conclusive by itself, but taken together they give a good picture of proxy behavior. As before, there are seven paths which have HTTP-level proxies; on port 80, all seven sent proxy SYN/ACKs, but could not be tested for proxy data acks. Tables 6 and 7 show the number of proxies identified. The set of paths showing Proxy SYN/ACK behaviour is precisely the same as those showing either Proxy Data Ack or HTTP proxy behavior. Taken together, these tests provide good evidence for proxies of the form described above.

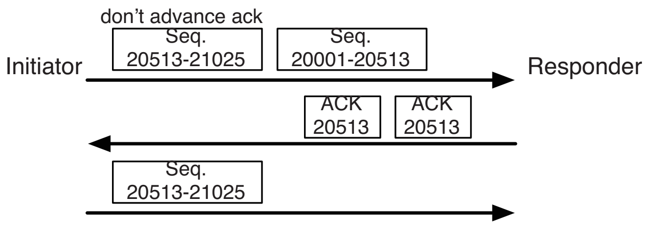

Figure 3: *Retransmission Test*

4.5 Inconsistent Retransmission

If a TCP sender retransmits a packet, but includes different data than the original in the retransmission, what happens? This might seem like a strange thing to do, but it might be advantageous for extensions that do not need stale data (such as VoIP over TCP). Given that we know sequence holes are a bad idea (Section

Table 12: Results of Retransmission Test

Observed Behavior	TCP Port / Retransmitting size 34343 same	34343 smaller	34343 larger	80 same	80 smaller	80 larger	443 same	443 smaller	443 larger
Passed	134 (99%)	134 (99%)	132 (98%)	124 (87%)	124 (87%)	123 (87%)	138 (97%)	138 (97%)	136 (96%)
No response	0 (0%)	0 (0%)	1 (1%)	0 (0%)	0 (0%)	1 (1%)	0 (0%)	0 (0%)	1 (1%)
Ack adv'ced	1 (1%)	1 (1%)	1 (1%)	10 (7%)	10 (7%)	10 (7%)	4 (3%)	4 (3%)	3 (2%)
Reset conn	0 (%)	0 (0%)	0 (0%)	1 (1%)	1 (1%)	1 (1%)	0 (0%)	0 (0%)	0 (0%)
Error	0 (0%)	0 (0%)	1 (1%)	7 (5%)	7 (5%)	7 (5%)	0 (0%)	0 (0%)	1 (1%)
Total	135 (100%)			142 (100%)			142 (100%)		

4.3), it might make sense to fill the sequence hole with previously unsent data.

Such inconsistent retransmissions would be explicitly "corrected" by a traffic normalizer[13], as its role is to ensure that any downstream intrusion detection system sees a consistent picture. Equally, depending on their implementation, TCP proxies might reassert the original data. We set out to test what happens in reality.

Fig. 3 shows our retransmission test. The initiator sends two consecutive segments, but we request that the responder sends a cumulative ack only for the first segment, then a duplicate Ack. Any stateful middlebox will infer that the second segment has not been received by the responder, and depending on its implementation, it may retain the unacked segment. We then send a "retransmission" of the second packet, but with a different payload (one that requests the responder echo the packet headers so we can see what is received).

We also repeat the test, but with the "retransmitted" packet being either 16 bytes smaller or 16 bytes longer than the original packet.

From the responses, we can distinguish four distinct middlebox behaviors, as listed in Table:.

- Most paths *passed* the inconsistent retransmission to the responder unmodified. In the case of port 34343, only one path did not do this.

- On some paths the initiator observes that the cumulative *Ack advanced*, but the headers were not echoed. This implies that the middlebox cached the original segment and resent it. Most of these paths were ones that we had previously identified as TCP proxies, but one on port 80 was not — it caches segments but does not separately ack data. We cannot know for sure, but this would be symptomatic of a traffic normalizer.

- One path returned *no response* at all when the inconsistent retransmit was larger than the original, and did so for all ports. There is no obvious reason for such behavior, so we speculate it might be a minor bug in a middlebox implementation.

- One path on port 80 *reset the connection.* This seems to be a fairly draconian response.

The usual seven paths with HTTP proxies could not be tested. One other path also failed to complete the test due to high packet loss.

Overall, any extension that wished to use inconsistent retransmissions would encounter few problems, so long as it did not matter greatly whether the original or the retransmission actually arrives. The one path that resets connections might however give the designers of extensions cause for concern.

We note that the proposal for TCP extended options might result in retransmissions that appear inconsistent to legacy middleboxes, even if the payload is consistent. This might occur if the value of an extended option such as a selective acknowledgment changes between the original and the retransmission.

4.6 Re-segmentation

TCP provides a reliable bytestream abstraction to applications, and makes no promises that message boundaries are preserved. Some TCP extensions such as TcpCrypt wish to associate a new option with a particular data segment — in the case of TcpCrypt to carry a MAC for the data. How will such extensions be affected by middleboxes?

We expect that TCP proxies will coalesce small segments if a queue builds in the proxy, and might split segments if the proxy negotiates a larger MSS with the client than that negotiated by the server. However, our results show such proxies remove unknown options from the SYN exchange, so any adverse interaction (beyond falling back to regular TCP) is unlikely. Our concern therefore is whether there are middleboxes that are not proxies that re-segment packets. In particular, any middlebox that passes new options *and* also re-segments data might be problematic.

To test resegmentation we simply advertise a relatively small 512 byte MSS. Any middlebox advertising a more normal (larger) MSS will be forced to resegment larger data packets into smaller ones. In fact, MSS advertised by 16 SYN proxies we observed at port

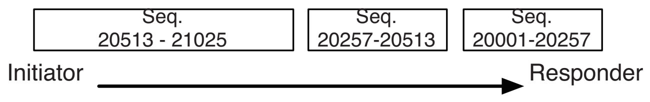

Figure 4: Segment Coalescing Test

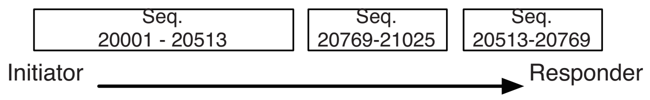

Figure 5: Queued Segment Coalescing Test

Table 13: Results of Segment Coalescing Test

Observed Behavior	TCP Port 34343	80	443
Passed	132 (98%)	123 (87%)	138 (97%)
Coal. ordered	1 (1%)	1 (1%)	0 (0%)
Coal. queued	1 (1%)	3 (2%)	1 (1%)
Coal. both	0 (0%)	6 (4%)	3 (2%)
Error	1 (0%)	9 (6%)	0 (0%)
Total	135 (100%)	142 (100%)	142 (100%)

80 varied between 1372 - 1460 bytes. We also test to see if unknown options (MP_DATA) or known options (TIMESTAMP) are copied to the split segments.

We found that 1 path on port 34343, 9 paths on port 80 and 4 paths on port 443 split segments in this way. These are the same paths identified as proxies in Table 6. None passed options to the split segments.

The opposite of segment splitting is segment coalescing, where a middlebox combines two or more segments into a larger segment. To test for this, we must send two consecutive small segments and observe whether a single larger segment arrives. However, a middlebox that has the ability to coalesce might still not do so unless it is forced to queue the segments. We therefore perform two versions of the test, as shown in figures 4 and 5.

- We test if segments are coalesced if the two small segments arrive in order (Fig. 4).
- We reorder the segments so that the small segments arrive after a gap in the sequence space, creating an opportunity for middleboxes to queue them (Fig. 5). We then send the segment which fills the sequence hole. If a middlebox queued the small segments, this will release them, potentially allowing coalescing to occur.

As before, we repeat the tests without options and with both known and unknown options.

Table 13 shows the results. We expected few middleboxes to coalesce in the in-order test (labeled *Coal. ordered*), but one path only coalesces in this case, and does not in the queued case. Other than this, the middleboxes running TCP proxies coalesce as expected. No middlebox copies either known or unknown options to the coalesced segments.

As before, on port 80 seven HTTP proxy paths could not be tested. Three other cases gave unexpected results. One path on port 34343 failed to complete all the tests, but appears not to coalesce when options are present. One path on port 80 returned no payload in the acks; other tests show this path does not show proxy behavior but does cache segments, does pass TCP options, but gives no reply to discontiguous sequence numbers. Likely it is also ignoring out of order segments in this test too. We do not know what form of middlebox this is, but its behavior seems fragile.

Finally, one non-proxy path did coalesce segments on ports 80 and 34343, but passed all the other tests. Interestingly, it only coalesced when options were not present.

Among those paths that coalesced, we saw quite a variety of behavior. The two small segments we sent were of 244 bytes. When coalescing occurred, depending on the path, the first coalesced segment received could be of 256, 488, 500 or 512 bytes in the first test and 256, 476 or 488 bytes in the second test. We have no idea what motivates these particular segment sizes.

Overall, the story is quite good for TCP extensions. Although middleboxes do split and coalesce segments, none did so while passing unknown options (indeed one changed its behavior when options were present). Thus is seems relatively safe to assume that if an option is passed, it arrives with the segment on which it was sent.

4.7 Intelligent NICs

Most of the experiments in this paper probe the network behavior, but with the rise of "intelligent" Network Interface Cards, even the NIC can have embedded TCP knowledge. Thus the NIC itself might fight with new TCP extensions.

We are concerned in particular with TCP Segmentation Offload (TSO), where the host OS sends large segments and relies on the NIC to resegment to match the MTU or the receiver's MSS. In Linux, TSO is accessed through a Generic Segmentation Offload (GSO) API. With GSO, the split segment size is chosen to allow all the IP options to be copied to all the split segments while still fitting within the MTU. But what do NICs actually do — do they really copy the options to all the split segments?

We tested eleven TSO NICs from four different vendors; Intel (82546, 82541GI, 82566MM, 82577LM, 82567V), Nvidia (MCP55), Broadcom (BCM95723, BCM5755) and Marvell (88E8053, 88E8056, 88E8059). For this, our initiator tool consists of a user application and a custom Linux kernel, and we reused the responder tool from the earlier middlebox tests. The key points about the experiment are:

- Our application calls write() to send five MSS of data to the socket layer at one time.
- The OS TCP stack composes one TCP segment that includes all the data and passes it to the GSO stack. This large segment also includes the Timestamp or Mp_Data TCP options.
- The NIC performs TSO, splitting the large segment into multiple segments and transmits them.
- Our responder receives these segments and responds with a segment echoing the headers in its payload so we can see what was received.

All the NICs we tested correctly copied the options to all the split segments. TSO is now sufficiently commonplace that designers of extensions to TCP should assume it may be encountered. The implication is that TCP options must be designed so that when they are duplicated on consecutive segments, this does not adversely affect correctness or performance.

5. PROTOCOL DESIGN IMPLICATIONS

5.1 Multipath TCP

As more and more mobile devices come equipped with multiple network interfaces such as 3G and WiFi, single path transport is fundamentally unable to utilize the aggregate capacity and robustness of the separate links. Multipath TCP (MPTCP) [24, 29] enables each TCP connection to be striped across multiple paths, while offering the same reliable, in-order, byte-oriented transport service to unmodified applications.

At first sight, MPTCP seems straightforward to implement, but the design has been evolving for a couple of years now, with most changes aimed at accommodating the middleboxes deployed today in the Internet. The experimental results described in this paper have guided the design, now undergoing standardization at the IETF.

To negotiate MPTCP, the endpoints use the Mp_Capable TCP option on SYN packets; they fall back to regular TCP if either endpoint does not support MPTCP or middleboxes along the path remove the new option. Our results indicate that if the option handshake goes through, MPTCP options will also be allowed on data segments. To be on the safe side though, MPTCP reverts to regular TCP if its options do not get through on any of the data segments sent during the first RTT of the connection.

Sequence numbers are fundamental to the MPTCP design. It would be easiest to reuse the TCP sequence numbers by striping segments coming from the TCP stack across different paths (e.g., by selecting different addresses for the same endpoint). A shortcoming of this approach is that, on each path, MPTCP subflows will look like TCP flows with holes in their sequence space. Our results show that quite a few middleboxes will not allow these flows to pass, and so MPTCP *had to* use one sequence space per subflow to pass through middleboxes. This in turn implies the need to add an additional data-level sequence number to allow the receiver to put segments back in order before passing them to the application.

How should the sender signal the data sequence numbers to the receiver? There are two possibilities: use TCP options or embed them in the TCP payload. Sending control information in the payload implies some form of payload chunking, similar to TLS-style TLV encoding. This would make it difficult for future middleboxes to work with MPTCP, as they would be forced to parse the payload. Architecturally, it is cleaner to encode data sequence numbers as TCP options.

The simplest solution is use a TCP option to add a data sequence number (DSN) to each segment. Although we observed no middlebox that both passed options and resegmented data, NICs performing TCP Segmentation Offload (TSO) would replicate the data sequence number onto multiple segments. Multiple segments would then have the same DSN — not what is desired.

Such a failure is a consequence of an *implicit* mapping of subflow sequence numbers (in the TCP headers) to data sequence numbers (in the options). The solution adopted by MPTCP is to make this mapping explicit: a data sequence mapping option carries the starting data sequence number, the starting subflow sequence number and the length of the mapping. To complicate things more, we have seen than subflow sequence numbers may be rewritten by firewalls. To avoid this problem, MPTCP signals subflow sequence numbers relative to the initial subflow sequence number. This solution makes MPTCP compatible with all the paths we observed.

Finally there is one form of application level gateway we did not test for - a NAT with knowledge of FTP or SIP that rewrites IP addresses in the TCP payload. Such rewriting can change the payload length and would be really bad for MPTCP: Reordering at the receiver might result in arbitrary-ordered data being passed to the application. MPTCP includes a checksum in the DSN mapping option to guard against such payload changes, and falls back to single path TCP if required.

There are many more design decisions in MPTCP that were dictated by verified, anecdotal or just possible middlebox behaviours. We quickly list two here:

- Retransmitting data: to avoid the problems we observed with sequence holes, MPTCP always sends the original data on retransmission, even though that same data may already have been received by the receiver via a different subflow.

- Proactive ACKing middleboxes might fail before sending data to the receiver; this would halt MPTCP if data-level ACKs were inferred from subflow ACKs. Although we observed no pro-actively acking middlebox that would pass MPTCP options, MPTCP includes a data-level acknowledgement, sent as a TCP option, to guard against such failures.

MPTCP was designed from ground up to co-exist with current middleboxes and to play nicely with future ones. The middlebox behaviour tests we have conducted in this paper have provided a solid basis for MPTCP's design choices.

5.2 TcpCrypt

TcpCrypt is a proposed extension to TCP that opportunistically encrypts all TCP traffic [4]. TcpCrypt endpoints share a public key on the wire and use that to derive a session key. After the initial handshake TcpCrypt connections are secure against eavesdropping, segment insertion or modification and replay attacks. During the initial handshake, connections are susceptible to man-in-the-middle or downgrade attacks, but TcpCrypt also provides hooks to allow application-level authentication of the encrypted connection.

TcpCrypt was motivated by the observation that server computing power is the performance bottleneck. To make ubiquitous encryption possible, highly asymmetric public key operations are arranged so that the expensive work is performed by the client which does not need to handle high connection setup rates. This is in contrast to SSL/TLS where the server does more work. This reversal of roles together with ever increasing computing power makes it feasible to have "always on" protection [5].

Use of TcpCrypt is negotiated with new CRYPT options in SYN segments, and keying material is included in INIT messages that are sent in both directions in the TCP payload before application data is sent. The INIT exchange also probes the path support for new options on data segments, thus coping with any middleboxes that allow new options on SYNs but not on data. After the initial negotiation, TcpCrypt can be either in the *encrypting* or *disabled* states. In the *disabled* state TcpCrypt behaves exactly like regular TCP. No further transitions are allowed once the connection reaches one of these two states [4]. This is because applications can query the TcpCrypt connection state and use it to make authentication decisions.

In the *encrypting* phase TcpCrypt encrypts the TCP payload with the shared session key and also adds a TCP MAC option to each segment that is validated at the receiver. The keyed MAC covers the encrypted payload as well as parts of the TCP header: the sequence numbers, the TCP options, and the length, as well the acknowledgement sequence number. The MAC covers neither the TCP ports nor the IP header to allow network address translation.

TcpCrypt only accepts segments whose MAC is correct; when the TCP MAC option is missing or incorrect the segment is silently dropped. Hence, each segment will have a unique MAC.

Middleboxes that resegment TCP packets would cause TcpCrypt's MAC to fail validation, causing the connection to stall. Unlike MPTCP, fallback to vanilla TCP behavior after entering the *encrypting* state is not viable. Fortunately we have not observed any paths that both pass new TCP options and resegment data. TCP Segmentation Offload would also cause TcpCrypt to fail, but fortunately the OS can disable this — the performance penalty is negligible compared to the cost of encryption.

To guard against segment injection and replay attacks the MAC needs to cover the TCP sequence numbers. This would fail when firewalls rewrite the ISN, so TcpCrypt includes the number of bytes since the start of the connection in the pseudo-header covered by the MAC rather than the absolute sequence number.

The MAC also covers acknowledgement sequence numbers. Any proactive ACKs sent by middleboxes will just be dropped. If no ACKs are passed end-to-end the connection will fail. Fortunately, this problem is unlikely as such boxes are proxies (See Sec. 4.4), and so would prevent TcpCrypt negotiation in the initial handshake by removing the SYN options. Finally, HTTP-level proxies require a valid HTTP header, which TcpCrypt would hide. However, such proxies also prevent the initial handshake.

A key difference between TcpCrypt and MPTCP is the distinction between disabled and enabled; when TcpCrypt is enabled it gives extra security to applications, which then rely on the protection provided. Once enabled it is unacceptable from a security point of view to revert to TCP. MPTCP, on the other hand, provides the same reliable, in-order, byte-stream service to applications, and can detect problems and revert to TCP at almost any time during a connection's lifetime.

5.3 Extending TCP Options

Extending TCP option space has been a discussion topic on IETF mailing lists on many occasions, starting as early as 2004. The main reason no solution was standardized is because people felt there was no pressing need for more option space. MPTCP uses quite a bit of option space, as does TcpCrypt; this usage, combined with existing options in use, leaves very little TCP option space remaining. With MPTCP approaching standardization, extending the TCP option space has now gained enough support to happen in practice.

Option space is scarce on both SYN and regular data packets. Extending the option space on the first SYN (active open) is difficult because of the need to be backward compatible: if one adds more options to the SYN, a legacy host might treat the extra options as application data, corrupting the connection [17].

Extending the option space in regular segments seems straightforward at first sight; the sending host simply needs to "extend" the data offset field in the TCP header. This is what the Long Option (LO) draft [8] suggests: add a new LO option that a 16 bit-wide value of the data offset. As with the other extensions we have discussed, resegmentation would be problematic here, but we did not observe any middlebox that passes options and resegments. Still, it would be good if the use of long options did not preclude TCP Segmentation Offload, and this solution would — every split segment would appear to carry a long option when in fact only the first would.

To allow TSO, the sender must be explicit about the placement of extended options, and solutions will resemble MPTCP's data sequence mapping. The receiver will be told the start of extended options and their length[†].

The same constraints apply as in the case of MPTCP signaling: the initial sequence number may be rewritten, thus the sequence number must be relative to the beginning of the flow. If middleboxes change payload length (for instance by rewriting IP addresses for FTP/SIP), the extended option sequence numbers will be inaccurate; a checksum covering the extra options is needed to cover such cases.

Another problem with extending TCP option space is the interaction between middleboxes that understand deployed TCP options, such as SACK. A middlebox might modify sequence numbers in both the header and SACK blocks, but not understand the LO option. However, if the sender places a SACK block in the extended option space, such middleboxes will not see it, and so cannot correct the selective acknowledgment numbers. We observed a significant number of middleboxes that modify sequence numbers and pass the unknown TCP options, so this problem does not seem hypothetical.

Segment caching middleboxes can also affect the LO option. If the options in the payload differ between the original and the retransmitted segments, the middlebox will consider them as different application data. We observed such segments could induce connection failures.

Work arounds are possible - SACK blocks would have to be placed in the regular options space, and no option in the extended option space would be allowed to change on a retransmission. But such workarounds rather limit the usefulness of extended options and increase both the complexity of implementations and the potential for subtle bugs.

[†]This is very much the functionality provided by the urgent pointer, but this is known not to go well through middleboxes[12]

6. CONCLUSION

Our goal in this paper has been to determine whether it is still possible to extend TCP. In particular, what limitations are imposed on TCP extensions by middleboxes and by "intelligent" NIC hardware? To answer these questions necessitated building novel measurement tools and recruiting volunteers from all over the world to run them on a wide range of networks.

From our results we conclude the middleboxes implementing layer 4 functionality are very common — at least 25% of paths interfered with TCP in some way beyond basic firewalling. We also conclude that it is still possible to extend TCP using its intended extension mechanism — TCP options — but that there are some caveats. Here are some guidelines:

- Negotiate new features on the SYN exchange before use.
- Be robust if an option is removed from the SYN/ACK - just because the server agrees to use a feature does not mean the client sees that agreement.
- Assume segments will be split (by TSO) and options duplicated on those segments.

There are also some warning stories, regarding behavior that is not safe to assume:

- Do not assume sequence numbers arrive unmodified - if you have to quote them, quote bytes from the start of the connection rather than absolute sequence numbers.
- Do not leave gaps in the sequence space - middleboxes need to see all the packets.
- Retransmitting inconsistent information is risky.
- Proxies are common, especially on port 80, and will strip TCP options.
- If options are removed, don't assume message boundaries will be preserved.
- Some middleboxes are surprisingly fragile to out of order packets.

Based on this information, we looked at whether three extensions to TCP had made sensible choices. We found that for the most part they had; in fact they were rather tightly constrained by middlebox behaviors to the solutions they had chosen. Of the three extensions we considered, TCP Long Option presents the greatest cause for concern. In particular, it becomes quite easy with long options to produce behaviour that looks to a middlebox like inconsistent retransmission due to the contents of extended options changing. Such inconsistent retransmission is demonstrably unsafe. If TCP Long Option were to be deployed, it would require additional constraints to avoid this problem.

We plan to continue this work, examining more networks and adding more tests. In addition, long-term continuous measurements are necessary to study the evolution of middleboxes, whereas this paper only presents a snapshot of the current Internet.

7. ACKNOWLEDGMENTS

We especially want to thank the volunteers from all over the world who ran our test code; without their help, we would have been unable to gather these results.

8. REFERENCES

[1] M. Allman. On the Performance of Middleboxes. *ACM IMC*, 35(2):307–312, 2003.

[2] A. Bakre and B. Badrinath. I-TCP: indirect TCP for mobile hosts. In *Proc. IEEE ICDCS*, 1995.

[3] H. Balakrishnan, S. Seshan, E. Amir, and R. Katz. Improving TCP/IP Performance over Wireless Networks. In *Proc. ACM MOBICOM*, pages 2–11, 1995.

[4] A. Bittau, D. Boneh, M. Hamburg, M. Handley, D. Mazieres, and Q. Slack. Cryptographic protection of TCP Streams (tcpcrypt). *draft-bittau-tcp-crypt-00.txt*, July 2010.

[5] A. Bittau, M. Hamburg, M. Handley, D. Mazieres, and D. Boneh. The case for ubiquitous transport-level encryption. In *Proc. USENIX Security Symposium*, Aug 2010.

[6] B. Carpenter and S. Brim. Middleboxes: Taxonomy and Issues. *RFC 3234*, Feb. 2002.

[7] R. Chakravorty, S. Katti, J. Crowcroft, and I. Pratt. Flow Aggregation for Enhanced TCP over Wide-Area Wireless. In *Proc. IEEE INFOCOM*, pages 1754–1764, 2003.

[8] W. Eddy and A. Langley. Extending the Space Available for TCP Options. *Internet Draft*, Jul. 2008.

[9] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. IP options are not an option. *Tech. Rep. UCB/EECS- 2005-24*, 2005.

[10] A. Ford, C. Raiciu, and M. Handley. TCP Extensions for Multipath Operation with Multiple Addresses. *Internet Draft*, July. 2010.

[11] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural guidelines for multipath TCP development. *RFC 6182*, Mar. 2011.

[12] F. Gont and A. Yourtchenko. On the Implementation of the TCP Urgent Mechanism. *RFC 6093*, Jan. 2011.

[13] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium*, pages 9–9, 2001.

[14] S. Hätönen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti, and M. Kojo. An Experimental Study of Home Gateway. *ACM IMC*, 2010.

[15] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. *RFC 1323*, May. 1992.

[16] J.Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. *RFC 3135*, Jun. 2001.

[17] Re: [tcpm] Extending the TCP option space - yet another approach. http://www.ietf.org/mail-archive/web/tcpm/current/msg06481.html.

[18] L. Popa and A. Ghodsi and I. Stoica. HTTP as the Narrow Waist of the Future Internet. In *Proc. ACM Hotnets '10*, 2010.

[19] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. *RFC 2018*, Oct. 1996.

[20] A. Medina, M. Allman, and S. Floyd. Measuring the Evolution of Transport Protocols in the Internet. *ACM CCR*, 35(2):37–52, 2005.

[21] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. *RFC 2663*, Aug. 1999.

[22] J. Padhye and S. Floyd. On Inferring TCP Behavior. In *ACM SIGCOMM*, pages 287–298, Oct. 2001.

[23] V. Paxon. End-to-End Internet Packet Dynamics. In *Proc. ACM SIGCOMM*, pages 139–152, 1997.

[24] C. Raiciu, D. Niculescu, M. Handley, and M. Braun. Opportunistic Mobility with Multipath TCP. In *Proc. ACM MobiArch*, 2011.

[25] S. Savage. Sting: a TCP-based Network Measurement Tool. In *USENIX USITS*, 1999.

[26] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP Congestion Control with a Misbehaving Receiver. *ACM CCR*, 29(5):71–78, 1999.

[27] R. Stewart, M. Ramalho, and et al. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension. *RFC 3758*, May. 2004.

[28] D. Watson, M. Smart, G. R. Malan, and F. Jahanian. Protocol Scrubbing: Network Security Through Transparent Flow Modification. *IEEE/ACM ToN*, 12(2):261–273, 2004.

[29] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *Proc. USENIX NSDI*, 2011.