NoTCP: steganographic tricks to bypass middleboxes

[Paper: #137]

ABSTRACT

TCP is the de facto transport protocol in the Internet. However it has many limitations that make it ill-suited to the modern world of high mobility and multi-homing. Protocol optimizations and filtering performed by middleboxes make extension deployment challenging. This has led researchers to explore new ways of exchanging control information between endpoints. We present NoTCP – a mechanism which uses the TCP header fields in combinations that have no defined semantics in order to hide a small amount of control information from middleboxes and hence bootstrap a control channel. We also present a preliminary study showing how the mechanism works across cellular and WiFi networks.

1. INTRODUCTION

Over the last decade a number of new protocols have been proposed to replace TCP. However many middleboxes will actively block or interfere with traffic that does not conform to their view of what TCP should look like [19, 22, 36]. Consequently, few of these protocols have seen real deployment.

One of the main requirements to extend and modify TCP is the ability to exchange control information. Traditionally this is what TCP option space was used for. However option space is limited, heavily contended [21, 26] and network middleboxes make it difficult to use because of inconsistent interpretation of the specifications [19, 23, 37]. On the other hand, modern networks demand new protocols for mobile and multi-homed devices [26], ubiquitous traffic encryption [21, 32], energy efficiency [2], connection parameters tuning for specific workloads and environments [9, 13] that were unknown 30 years ago.

One solution is to work round the middleboxes by using steganographic techniques [16, 27, 33, 40] to conceal control information within the TCP header beyond the sight of middleboxes. This could be as simple as using one of the currently undefined flags to toggle the meaning of existing fields, through to using the urgent pointer without setting the urgent flag or even hashing information into the initial sequence number in much the way that TCP Fast Open [7] does for application data delivery.

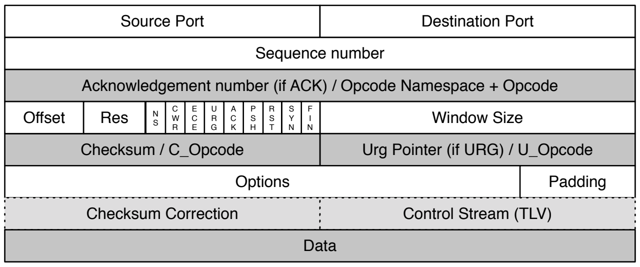

Figure 1: TCP Header with overloaded fields and extra control information (Checksum Correction / Control Stream) in payload. Fields in gray used for control information. Light-gray fields are optional and are within payload.

Steganographic communication channels are typically low bandwidth and statistical analysis of a sufficient amount of data usually exposes the existence of such channels [27]. Nevertheless, it is the perfect choice for exchanging a small amount of end-to-end control data in real-time and it does not compromise network security since the channel can be detected using existing techniques. In our work we discuss the possible ways to conceal information in the TCP header and present the results of a preliminary study on the suitability of various header fields for the purpose.

We use our initial study to design a way of exchanging a small amount of control information successfully across all of our tested networks and show a potential mechanism to use it for bootstrapping a full control stream currently missing from TCP.

Our approach is not a short-term hack. Many recent proposals have demonstrated that control over end-to-end protocol behavior should remain at the endpoints. These include TCPCrypt which offers improved communication confidentiality and security and TCP Fast Open for better communication performance. We hope that our successful results will motivate the research community into exploring the full range of possibilities in this space.

2. CANDIDATE PROTOCOL CHANGES

Design of a control channel is constrained by protocol specification and its interpretation by middleboxes. In the past, TCP sequence numbers [27], timestamps [17] and IP ID [1] have been used for encoding hidden information. In our work we focus on a different set of TCP header fields (Fig. 1, in gray, defined in §5). Overall, the header leaves few options for hiding control information:

- Setting fields without the corresponding flags so they have no meaning to legacy implementations.
- Assigning one or more of the reserved field bits.
- Overloading the semantics for valid header fields.

Flag-dependent semantics

Only two header fields have their semantics set by corresponding flags. These are the *Acknowledgment Number* (which must always be set after the initial SYN packet) and the *Urgent Pointer*.

The Acknowledgment Number field is only available in the initial SYN packet, but is usable across our tested networks (§4). It must therefore be combined with another mechanism to acknowledge control information unless TCP Simultaneous Open is used. Simultaneous open is still viable in most cellular networks [37], but we also explore other options.

The general recommendation is to not use the urgent mechanism [18]. The *Urgent Pointer* could therefore be reused for other purposes as long as both endpoints understand the new semantics and middleboxes permit it. On the other hand, Network Intrusion Detection Systems (NIDS) often reset the field as it makes it difficult to track application-layer data [39]. Importantly, the field becomes *Non-Urgent* [25] when URG is not set – it has no meaning for legacy implementations and so can be redefined if middleboxes permit it as they often do (§4).

Reserved bits

Three TCP reserved bits remain unallocated. One further bit is only allocated to the experimental ECN nonce [12]. Despite their limited size, there are proposals that make use of new reserved bits due to implementation simplicity and non-interference with legacy implementations. One suggestion for reducing Web latency [14] by mitigating the effects of a small number of lost packets for short connections is to send multiple copies of packets. The proposal deliberately sends copies, but uses a reserved flag to explicitly mark solution avoids triggering fast retransmit by such duplicate acknowledgments by marking them as duplicates in a reserved flag.

Value-specific semantics

The remaining choice is to encode information within otherwise valid header fields. It involves generating packets with specific values for *Initial Sequence Numbers*, *Window Size* or *Checksums*. Arguably it is an acceptable choice if deployable in today's networks. Indeed SYN Cookies already do this [10]

Initial Sequence Numbers must be hard to predict as they provide a measure of security against TCP injection attacks. Adding information to the field to be easily decoded by the receiver makes it more predictable and weakens already problematic security [3, 30]. Also some stateful middleboxes are known to interfere with sequence numbers in an attempt to improve security [29]. Stronger security measures such as TCPCrypt [26] are better suited for connection protection.

Window Size is potentially a better candidate, especially if used during connection setup phase since data is not typically transmitted during the handshake. Using lower-order bits for control information would prevent flow control from operating properly, but higher-order bits could instead be given up when the *window scaling* option is used to compensate for the decrease of advertised window size. Using the field risks interfering both with TCP flow control mechanism as well as window size optimizations by the network [5, 24] performed by *e.g.,* Citrix ByteMobile Proxy [20].

There are multiple benefits to using the checksum field. Firstly, it is only used for error detection. It may be unnecessary when reliable error detection is done at the layer below, however link-level checks have been shown to be unreliable [35]. Instead, it may be replaced with potentially stronger mechanisms such as MPTCP DSS checksum and TCPCrypt Message Authentication Code or left for the upper layer. Secondly, checksum operations are efficient as they only involve one's complement addition. We show later in the paper (§5) how checksums can be generated in a way that the intended value is recovered by the receiver after traversing multiple layers of Network Address Translation (NAT).

3. MEASUREMENT METHODOLOGY

To measure current network behavior we have developed an Android application distributed through the *Google Play Store*[1] to collect data from volunteers. It contains a simplified implementation of TCP which generates packet sequences that appear valid to network middleboxes: three-way handshake, data exchange and connection teardown. It allows us to modify various header fields and check the values received by the other endpoint. We control both endpoints of each connection, therefore we can detect changes from sent values received by the other endpoint.

The test setup is:

- Android application with a simplified TCP stack using Raw Unix sockets for full control over the TCP header (*SOCK_RAW*). It manages the tests and sends results back to our servers.
- Server instrumented to generate specific responses to incoming packets based on header and payload

[1]Link omitted to anonymize submission

values. We currently use a dedicated test server in a single location and no firewalls or NATs.

Non-transparent proxies are a major problem for extension deployment, therefore we used another well know and widely deployed tool to detect proxies on our tested networks. We implemented previously developed techniques [38] and run the test separately. The primary one was non-responsive server test: the server is instrumented not to respond to SYN, therefore if the connection succeeds it is to a proxy on the path. We repeated these tests on different destination ports and found that proxying can be specific to certain port numbers.

Our methodology avoids packet splitting by only using small packets. Coalescing is prevented by only exchanging a single data message each way. For the same reason we do not take congestion control into account.

Raw sockets operate in parallel to the kernel network stack. We had to suppress reset packets generated by the kernel in response to TCP packets from sessions it is not aware of. We also had to choose local port numbers unused by the kernel. We chose them randomly from the unprivileged port range and relied on other error detection mechanisms preventing invalid packet delivery to any applications.

4. PRELIMINARY RESULTS

Our preliminary study focused on mobile networks including public, residential, university and enterprise WiFi networks as well as six cellular networks in four different countries. We present our results in Table 1.

The results include networks that did not filter any modifications (only one WiFi and two cellular nets), and others that interfered with our traffic. We looked for dropped packets and reset or altered fields: (*Acknowledgment Number* and *Urgent Pointer*, potentially invalid checksums and reserved flags. Only one network provided clients with a global IP (WiFi edu 2) so we also tested whether NATs rewrite sequence numbers in addition to addresses and port numbers.

As discussed in §2, we can squeeze in a few extra bits of information by using certain header fields when the corresponding flags are not set. None of our tested networks filtered the Acknowledgment Number field for the initial SYN packet on non-proxied paths and only two networks discarded packets with the Urgent Pointer set without the corresponding URG flag.

We also tested middlebox interaction with reserved header bits (§2). We separated our tests to handshake and data exchange parts as well as set each bit individually, however we did not observe any differences. All networks appear to pass reserved bits on non-proxied connections.

Traditional NATs [11] perform a simple checksum recalculation: subtract the old header fields from the checksum and add the new ones. The networks that deliver packets with invalid checksum only do that. Otherwise they would detect and drop an invalid packet.

A bigger issue we have discovered is that middleboxes do not always respect the standard [28] and drop SYN-ACK packets with payload. We did not see it on cellular networks, but most WiFi (4/7) behaved in such way even though the standard allows handshake packets to contain payload [7, 28].

4.1 Port-specific middlebox behavior

To differentiate between general behavior of particular network and application-specific optimizations (*e.g.,* HTTP acceleration) we repeat our tests on a few application ports as well as a random one:

- 80, 443 - HTTP and HTTPS
- 993 - Secure IMAP port
- 5228 - Google cloud messaging port
- 6969 - random port number
- 8000 - common HTTP proxy port number

We only observed port-specific behavior on two networks: *Cellular 2* and *Cellular 4*. In the first case, traffic travels over SSL ports (443, 993) unmodified except for packets with invalid checksum being dropped. Our traffic was not encrypted. An increasing proportion of traffic is served over SSL, making the value of such proxies questionable: a commercially deployed HTTP accelerator reports the proportion of HTTPS requests to be close to that of HTTP[2].

We confirmed the presence of a ByteMobile proxy on Cellular 2. It replaces the original connections with new ones, adding options not present in original packets, changing window size and resetting other fields to different values to optimize TCP for cellular networks [20]. Such behavior makes deployment of protocol extensions difficult – we verified that TCPCrypt fails on this network. We do not currently have an estimate of how widespread ByteMobile proxies are as we only observed them on one network.

In the case of *Cellular 4*, traffic was only modified when going to ports 80 and 443. The network likely optimizes HTTP traffic and drops non-HTTP traffic. Generally port-specific behavior appears related to proxies.

4.2 Non-transparent proxies

Proxies potentially change every aspect of the original connection, rewrite or filter header values they do not recognize so hiding any information within the packet becomes difficult.

We tested for them on a number of popular port numbers, from those use for HTTP and mail to typically encrypted ones: 21, 22, 25, 80, 110, 135, 139, 143, 161, 443, 445, 465, 585, 587, 993, 995, 1194, 1723, 5060, 6881, 9001.

[2] `http://db.awazza.com/users/global/`

Net ID	**Port-specific**	**Validate checksum**	**Drop SYN-ACK data**	**Normalize SYN Ack number**	**Normalize Urgent Pointer**	**Normalize Reserved**	**Remap Sequence**
WiFi edu 1		✓					
WiFi edu 2			✓		✓		✓
WiFi pub 1							
WiFi pub 2		✓	✓				
WiFi res 1		✓					
WiFi res 2		✓	✓				
WiFi ent 1		✓	✓				
Cellular 1							
Cellular 2	443, 993	*	*	*	*	*	*
Cellular 3					✓		✓
Cellular 4	80, 443	*	*	*	*	*	*
Cellular 5		✓					
Cellular 6		✓					

Table 1: Network behavior observed through tests generating custom TCP packets. * means different cases observed based on port numbers.

Net ID	**Radio Technology**	**Proxied Ports**
Cellular 2	HSPA	21, 22, 25, 80, 110, 135, 143, 161, 465, 585, 587, 995, 1194, 1723, 5060, 6881, 9001
Cellular 4	GPRS	80, 110, 143
Cellular 4	HSPA	80, 110, 143
Cellular 4	LTE	80, 110, 143, 443, 993, 995
Cellular 4*	HSPA	25, 80, 110, 143
Cellular 4**	HSPA	25, 80, 110, 143

Table 2: Non-transparent proxies and their proxied port numbers by radio technology. */** - see text for details.

5. CONTROL INFORMATION EXCHANGE

What is common to all candidate protocol changes described above is that they are limited in size and therefore only sufficient to exchange an opcode. Other protocol details must be exchanged either out of band, implicitly to the opcode, or explicitly by redefining the meaning of part of the data payload or previously exchanged information. Here we propose one way of exchanging control information as an opcode to bootstrap a full control stream.

5.1 Opcode embedding

A surprising result in our tests is that the *Acknowledgment Number* field without the ACK flag set is passed on all non-proxied connections. As it is a 32-bit value we can use part of it to signal that the rest of the field is significant. The exact allocation is subject to further discussion, but we suggest using the first 16 bits as *Extension Space* and the others as *Opcode* (Figure 1). For *simultaneous open* we can use the same mechanism in both directions.

For acknowledging extensions in the SYN-ACK packet we use the Urgent Pointer and Checksum fields. Our tests indicate that neither choice is guaranteed to succeed, therefore we combine the two by redefining their values when SYN=1. We define the Urgent Pointer to be $Opcode_U$ when URG=0. If Urgent Pointer is 0, Checksum is the opcode to get around the cases where one is blocked but the other one is not. Across all networks tested in our preliminary study, at least one of the choices succeeds. In the cases when both are usable data can be encoded across them as a single field once the handshake is completed.

Checksum value can not be used directly as an opcode since it changes at every level of NAT. The traditional NAT does a simple recalculation of the checksum [11]: subtracts the original source address and port number (in some cases also the sequence number) and adds the new values. The Checksum after n NAT hops is:

$$\begin{aligned} Checksum_2 &= Checksum_1 - header_1 + header_2 \\ Checksum_i &= Checksum_{i-1} - header_{i-1} + header_i \\ &\dots \\ Checksum_n &= Checksum_1 - header_1 + header_n \end{aligned}$$

Here *header* refers to the fields of the header that change, namely the destination port number, address and sequence number. A previous study [37] found no NATs mangling sequence numbers, but we witnessed a few cases of such behavior. The Opcode can then be encoded as a target checksum so that it is recoverable by a simple binary subtraction of the changing fields (3 to 7 16-bit arithmetic operations in total):

$$\begin{aligned} Checksum_{target} &= Opcode + header_1 \\ Checksum_n = Opcode_C &= Opcode + header_n \end{aligned}$$

There are two choices for setting checksum to a specific value: invalid checksum or adding padding payload. The latter can only be done when the receiving end is already expecting it, since it should not pass such payload up to the layer above, but this approach keeps a packet valid from network's perspective.

For our prototype we assign the first 16 bits of the payload as *Checksum Correction* to force a specific checksum to maintain validity of a packet. The field value is not considered as part of application data. The choice is in line with other extensions which reuse part of data payload [4,26]. Sending data with the SYN-ACK packet works on all tested cellular networks, but not on most WiFi.

For the above header modifications the information either reaches the destination as intended, or the packet is dropped. Finally, on all of our tested networks at least one case succeeds as long as there is no non-transparent proxy for the chosen destination port.

5.2 Bootstrapping the control stream

Covert channel is only needed to ensure that both endpoints understand the same protocol semantics. Once the handshake with hidden control information is complete, any aspect of a protocol can be redefined. For example, a higher-bandwidth *Control Channel* can use TLV (Type-Length-Value) encoding to delimit control information within packet payload [4]. The channel is bootstrapped using an extended three-way handshake with fields shown in Figure 1:

1. **SYN**: the active opener (client) sends SYN packet with $OpcodeNamespace$ + $Opcode$ and ACK unset.
2. **SYN-ACK**: if passive opener (server) recognizes the $Namespace$ and the $Opcode$, it replies with SYN-ACK with $Opcode_U = Opcode$ (URG unset) to acknowledge the Opcode.
3. If the server does not receive an ACK, retransmit with $Opcode_C$ and *Checksum Correction*. If that fails as well, retransmit the SYN-ACK with an invalid checksum. Finally, fall back to vanilla TCP and retransmit a standard SYN-ACK.
4. **ACK**: when the client receives a packet, it checks for $Opcode_U$ (if URG is unset). Otherwise, it checks if Checksum = $Opcode_C$ as described above. If no Opcode is recoverable, fall back to vanilla TCP and transmit a standard ACK.
5. Otherwise both openers recognize the *Opcode*. Active opener sends an ACK packet with control information in the field it received it on and potentially TLV-encoded control information in packet payload. The control stream has been successfully bootstrapped.

Figure 2 shows an example handshake. In this case a client sends a SYN packet as described above. The packet traverses a middlebox that does not remove the opcode and reaches the server. The server replies with SYN-ACK with $Opcode_U$, but the middlebox discards it. The server then retransmits SYN-ACK with $Opcode_C$

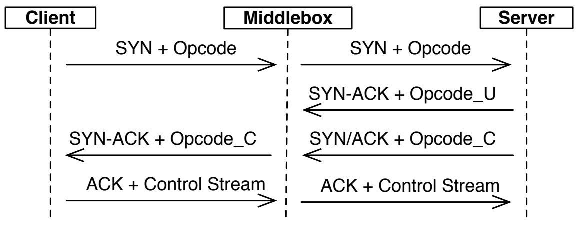

Figure 2: Example handshake with extension negotiation

and Checksum Correction bits, delivered to the client. The client recovers and verifies the received opcode. Hence both endpoints understand the meaning of the opcode, therefore the client replies with an ACK with control information acknowledging *Opcode* reception.

We witnessed one case of packet splitting where an ACK segment is split to one only acknowledging the received data and another other one with the actual payload. We did not observe a case where a middlebox both passes payload in SYN-ACK packet and splits ACK segments. If the handshake succeeds we need to take a little extra care with the fallback mechanism – we need to wait for an ACK segment with payload before falling back to vanilla TCP. If the passive opener receives data it can not decode as valid TLV-encoded control information, it falls back to vanilla TCP.

Putting control stream information into packet payload suffers from much the same issues as other proposals to extend TCP option space [31]. The premise of our work is that middleboxes should not be aware of the new control streams in the first place and therefore not interfere with them. Currently we focus on exchanging control information within the packet header and leave refinement of the full control channel for future work.

5.3 Control channel bitrate

Our control channel does not need to protection from steganalysis as it is only used to avoid middlebox interference with end-to-end control information exchange. The bitrate of the channel without redefining part of packet payload for control information is bounded by the size of the TCP header (20 bytes without options), the round trip time (RTT) and the number of packets in flight per RTT. Depending on fields usable for a particular connection as inferred during the handshake, we can use between 16 (either $Opcode_C$ or $Opcode_U$) and 35 bits when using both fields and reserved flags.

For an operating window *cwnd* (in bytes) a TCP flow has $cwnd/MSS$ data packets per RTT, hence that many headers can carry control data. If *window scaling* option is not used, the maximum *receive window* is 65536 bytes. Using the typical HSPA network RTT of about 100ms the maximum data rate is therefore 5 Mbit/s, reasonable for current networks [6]. In this case covert channel bitrate is 7.0 to 15.4 kbit/s, depending on the fields used.

We can estimate channel bitrate from previously studies of 3G and 4G network characteristics [6]. The average bandwidth measured for a large TCP transmission ranges from 1.2 ± 0.2 Mbit/s for a 3G network and 348 ± 141 ms RTT to 15.6 ± 1.0 Mbit/s and 125 ± 41 ms RTT for LTE. Covert channel bitrates for these cases range from 3.7 ± 0.6 kbit/s for 3G with one 16-bit field usable to 49.0 ± 3.1 kbit/s for LTE and both fields as well as reserved bits usable. Similarly, maximum bitrate of full-length option field is up to 34.5 ± 5.7 kbit/s and 448.2 ± 28.7 kbit/s for the two cases respectively.

Finally, the actual amount of information exchanged will depend on the coding scheme and the required reliability guarantees. For reliability in particular, a scheme described in Nusha [34] can be used.

For comparison, bitrates achievable by detection resistant schemes using initial sequence numbers are limited by only using the first packet in each connection to fit up to 15 bits of data [27]. Using the same assumptions as before and establishing connections sequentially results in 150 bit/s when only SYN and SYN-ACK are exchanged, to 75 bit/s with complete handshake and teardown.

6. RELATED WORK

Steganography has been used in the past to hide information within network protocols. The main focus has been on covert channels that violate system's security policy and initial sequence numbers [33], TCP Timestamps [17] and combinations with IP flags [27]. There are detection techniques for these methods, however our main goal is to exchange a small amount of control information rather than a genuine covert channel, calling for a different design.

TCP implements an *urgent mechanism* that allows the sender to make the receiver accept some *urgent data*. The mechanism is often quoted as providing "out-of-bound" data delivery and may be compelling to use for control information even though it is explicitly not designed as such. There are ambiguities regarding the semantics of the urgent pointer and Network Intrusion Detection Systems (NIDS) tend to clear the URG flag and pointer. Hence the general recommendation is not to use of the mechanism [18].

AccECN [25] uses an additional reserved bit, overloads the meaning of already assigned ECN [15] and NS [12] bits and redefines Urgent Pointer as Non-Urgent when URG is not set. The idea is related to ours, but reserving the field solely for ECN use prevents other extensions from using it. Also, we saw that some networks block such packets, making it especially difficult to use for ECN.

Generic control stream for MPTCP [4] is another alternative. It proposes mapping the control stream into a separate sequence number space and exchanging control data over established subflows, only modifying the MPTCP DSS option to differentiate control and data streams. The underlying assumption, however, is that MPTCP is already deployed and that existing deployments will be compatible with the new specification.

7. DISCUSSION AND FUTURE WORK

The main points of discussion for *NoTCP* are the same as with all TCP extensions: is it necessary, deployable and forwards-compatible? We have discussed the first two throughout the paper, but the biggest challenge for forwards-compatibility is NoTCP's overloading of header fields instead of using options. The exact protocol semantics are subject to further testing and discussions. The main goal of this work is to identify potential methods for exchanging control information by hiding it from existing middleboxes.

We have left open the question of finding a method to hide such information from non-transparent proxies. The easiest, albeit limited solution is to identify and cache port numbers that allow unmodified traffic. Instead we plan to study commonly found proxy implementations in detail in order to quickly identify them and encode information within their own operating patterns.

In the long term each protocol-related standard should explicitly define how it affects header fields. Examples include implicitly extending sequence number space [26], allowing encrypted TCP session to omit the timestamp option, using stronger message integrity checks making checksum field redundant, changing semantics of duplicate ACKs [14, 21] or even simple window size modifications [8].

We hope to map such header field modifications by existing extensions as well as network middlebox behavior patterns into an algebraic form as we have done in the simple case of checksum. Such definitions would allow us to go beyond the simple opcode exchange and signal detailed protocol information between endpoints using a low-bandwidth channel. Thus extending a protocol would become a matter of defining a new set of such header algebra rules hence simplifying the process of design and testing.

We have demonstrated the feasibility of using steganography to exchange end-to-end control information across current networks. Importantly, this is not a short term solution: TCP Fast Open is already being deployed and offers even more opportunities to hide information in the initial packets within higher-level protocol information, such as HTTP headers and cipher suites [40]. Even if TCP was completely replaced by another protocol, opportunities for control information exchange remain.

8. REFERENCES

[1] K. Ahsan and D. Kundur. Practical data hiding in TCP/IP. 2002.

[2] A. Aucinas, N. Vallina-Rodriguez, Y. Grunenberger, V. Erramilli, K. Papagiannaki, J. Crowcroft, and D. Wetherall. Staying online while mobile: the hidden costs. In *ACM CoNEXT '13*, 2013.

[3] S. Bellovin and F. Gont. Defending against Sequence Number Attacks, 2012. RFC 6528.

[4] O. Bonaventure and C. Paasch. A generic control stream for Multipath TCP, 2014. draft-paasch-mptcp-control-stream-00 (work in progress).

[5] R. Chakravorty, S. Katti, J. Crowcroft, and I. Pratt. Flow aggregation for enhanced TCP over wide-area wireless. In *IEEE INFOCOM '03*, 2003.

[6] Y. C. Chen, D. Towsley, and E. M. Nahum. Characterizing 4G and 3G networks: Supporting mobility with multipath TCP. Technical Report UM-CS-2012-022, UMass Amherst, 2012.

[7] J. Chu, A. Jain, Y. Cheng, and S. Radhakrishnan. TCP Fast Open, 2014. draft-ietf-tcpm-fastopen (work in progress).

[8] J. Crowcroft and P. Oechslin. Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP. *ACM SIGCOMM CCR*, 1998.

[9] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An argument for increasing TCP's initial congestion window. *ACM SIGCOMM CCR*, 2010.

[10] W. M. Eddy. TCP SYN flooding attacks and common mitigations, 2007. RFC 4987.

[11] K. B. Egevang and P. Srisuresh. Traditional IP Network Address Translator (Traditional NAT), 2001. RFC 3022.

[12] D. Ely, N. Spring, and D. Wetherall. Robust Explicit Congestion Notification (ECN) Signaling with Nonces, 2003. RFC 3540 (experimental).

[13] J. Erman, V. Gopalakrishnan, and R. Jana. Towards a SPDY'ier mobile web? In *CoNEXT '13*, 2013.

[14] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing web latency: the virtue of gentle aggression. *ACM SIGCOMM CCR*, 2013.

[15] S. Floyd, K. K. Ramakrishnan, and D. L. Black. The Addition of Explicit Congestion Notification (ECN) to IP, 2001. RFC 3168.

[16] W. Frczek, W. Mazurczyk, and K. Szczypiorski. Hiding information in a Stream Control Transmission Protocol. *Computer Communications*, 2012.

[17] J. Giffin, R. Greenstadt, P. Litwack, and R. Tibbetts. Covert Messaging Through TCP Timestamps. In *PET Symposium '02*, 2002.

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

[19] S. Guha and P. Francis. Characterization and measurement of TCP traversal through NATs and firewalls. In *ACM IMC '05*, 2005.

[20] S. Ha, S.-w. Han, T.-e. Kim, V. Bharghavan, U. Madhow, K. Ramchandran, and Bytemobile, Inc. Data transport acceleration and management within a network communication system, 2006. Patent. US7099273 B2.

[21] M. Handley, O. Bonaventure, C. Raiciu, and A. Ford. TCP Extensions for Multipath Operation with Multiple Addresses, 2013. RFC 6824 (experimental).

[22] M. Handley, V. Paxson, and C. Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *USENIX SSYM '01*, 2001.

[23] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *ACM SIGCOMM '11*, 2011.

[24] S. Kopparty, S. V. Krishnamurthy, M. Faloutsos, and S. K. Tripathi. Split TCP for mobile ad hoc networks. In *IEEE GLOBECOM '02*, 2002.

[25] M. Kühlewind, R. Scheffenegger, and B. Briscoe. More Accurate ECN Feedback in TCP, 2014. draft-kuehlewind-tcpm-accurate-ecn-03 (work in progress).

[26] D. Mazieres, D. Boneh, Q. Slack, M. Hamburg, A. Bittau, and M. Handley. Cryptographic protection of TCP Streams (tcpcrypt), 2014. draft-bittau-tcp-crypt-04 (work in progress).

[27] S. J. Murdoch and S. Lewis. Embedding Covert Channels into TCP/IP. In *PAM '05*, 2005.

[28] J. Postel. Transmission Control Protocol, 1981. RFC 793.

[29] Z. Qian and Z. M. Mao. Off-path TCP Sequence Number Inference Attack - How Firewall Middleboxes Reduce Security. In *IEEE SP '12*, 2012.

[30] Z. Qian, Z. M. Mao, and Y. Xie. Collaborative TCP sequence number inference attack: How to crack sequence number under a second. In *ACM CCS '12*, 2012.

[31] A. Ramaiah. TCP option space extension, 2012. draft-ananth-tcpm-tcpoptext-00 (expired).

[32] E. Rescorla. TCP Use TLS Option, 2014. draft-rescorla-tcpinc-tls-option-00 (work in progress).

[33] C. H. Rowland. Covert channels in the TCP/IP protocol suite. *First Monday*, 1997.

[34] J. Rutkowska. The implementation of passive covert channels in the Linux kernel. *Chaos Communication Congress*, 2004.

[35] J. Stone and C. Partridge. When the CRC and TCP checksum disagree. In *ACM SIGCOMM '00*, 2000.

[36] M. Vutukuru, H. Balakrishnan, and V. Paxson. Efficient and Robust TCP Stream Normalization. In *IEEE SP '08*, 2008.

[37] Z. Wang, Z. Qian, Q. Xu, Z. M. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *ACM SIGCOMM '11*, 2011.

[38] N. Weaver, C. Kreibich, M. Dam, and V. Paxson. Here Be Web Proxies. In *PAM '14*, 2014.

[39] Y. Yoon, J. Y. Oh, and Y. M. Yoon. NIDS Evasion Method named "SeolMa". Phrack Magazine, Volume 0x0b, Issue 0x39, Phile #0x03 of 0x12.

[40] E. Zielińska, W. Mazurczyk, and K. Szczypiorski. Trends in steganography. *Communications of the ACM*, 2014.