Minion—an All-Terrain Packet Packhorse to Jump-Start Stalled Internet Transports

Janardhan Iyengar[*], Bryan Ford[†]
Dishant Ailawadi[†], Syed Obaid Amin[*], Michael F. Nowlan[†], Nabin Tiwari[*], Jeffrey Wise[*]

ABSTRACT

Transport layer evolution is stuck. A proliferation of middleboxes in the Internet has shifted the waist of the hourglass upward from IP to include legacy transports [10, 13, 23]. While popular for many different reasons, middleboxes deviate from the Internet's end-to-end design, creating large deployment "black-holes"—singularities where legacy transports get through, but any new transport technology or protocol fails—severely limiting protocol evolution.

To restore the Internet's openness to innovation at the ends, we propose the *minion* suite: a protocols suite that uses legacy transports—UDP, TCP, and SSL—to provide a generic unordered datagram service between communicating endpoints, as a substrate atop which more sophisticated transports, such as those supporting partial-ordering, can be built *and* deployed. These *minions* are modified forms of the legacy transports where the protocols appear unmodified on the wire, thus making deployability through middleboxes possible. Our minions provide the basis for a much richer set of services that can be offered to the ends, thus recognizing the shifting waist of the Internet hourglass, and creating a powerful new substrate at the new waist.

1. INTRODUCTION

As Internet applications have changed over the past three decades, need for transport services outside the strait-jacket of TCP's ordered and reliable byte-stream has led to several alternatives [9, 17, 27]. Since TCP, however, no significant new transport has seen large-scale deployment.

The waist of the Internet hourglass has arguably moved upwards to include TCP and UDP [10, 23, 26], as is evident in the design of new transports, which use UDP encapsulation to traverse middleboxes. While encapsulating specific transports in UDP solves part of the deployment problem, this approach is inadequate and possibly detrimental to long-term deployment efforts, as we discuss in Section 2.

Applications and application developers care most about services that the networking infrastructure offers to them, and not how packets look on the wire; that is, they care about new transport *services*, not new transport *protocols*. On the other hand, middleboxes care most about how packets look on the wire, and generally do not care about what services are offered to the applications; that is, changing the transport protocol's bits on the wire will require changing middleboxes to respond to these changes as well. There is clearly a gap between traditional approaches to offering new transport services and what middleboxes expect.

The minion suite bridges precisely this gap—enabling rapid deployment of new transport services while appearing as UDP, TCP or SSL on the wire. The minion suite includes TCP-minion and SSL-minion, variants of TCP and SSL that look exactly like TCP and SSL on the wire, respectively, while offering an *unordered* datagram service, with optional congestion control, to the user above[1]. The minion suite can be used by new transports or applications to get more powerful, responsive services at the ends, while managing to get through existing middleboxes.

2. WHY NOT UDP ENCAPSULATION?

Given that the demand for new transport services exists, new transport services are being built and depoyed on top of UDP. Applications such as Internet telephony (Skype), video streaming (Adobe's RTMFP for Flash), and bulk background transport (Bittorrent's uTP), continue to build their own narrowly-focussed transport services over UDP, and deploy them as part of the application. To encourage deployment, transports designed at the IETF are also employing UDP encapsulation to navigate NATs in the network [22,29]. While the idea of UDP encapsulation seems seductively simple, this section discusses why *simple* UDP encapsulation of a new transport is an inadequate and fragile solution that may lead to more network encumbrances in the long term.

2.1 A Taxonomy of Transport Functions

To understand how UDP encapsulation fits in the larger picture of middlebox traversal, we examine the interactions between middleboxes and the traditionally-end-to-end transport layer. We first factor out functional components within the traditional transport layer into four sub-layers, as previously proposed in T*ng* [10]. Starting from the sub-layer immediately above the network layer, these are:

The Endpoint Layer factors out communication endpoint information—port numbers in the current Internet architecture—that policy-enforcing middleboxes such as firewalls, NATs, and traffic shapers require for operation;

The Flow Regulation Layer factors out congestion control and other functions that may require network interaction for purposes of adapting to network heterogeneity;

[*]Franklin and Marshall College, Email: jiyengar@fandm.edu
[†]Yale University, Email: bryan.ford@yale.edu

[1]While the minions, as discussed in this paper, are as reliable or unreliable as the underlying protocol, we are investigating methods for providing partial reliability to the application, and do not discuss them in this paper.

The Semantic Layer factors out application-oriented services such as end-to-end reliability and ordering; and

The Isolation Layer is an optional layer, traditionally above the Semantic Layer, which provides end-to-end cryptographic security via protocols such as TLS [6] and DTLS [25].

Figure 1 shows how middleboxes interact with and interpose on different transport layer components. NATs and Firewalls care about port numbers and application endpoint information, Performance Enhancing Proxies (PEPs) [4] care about congestion control, Traffic Normalizers [14] use detailed information about the TCP state machine to thwart network attacks, and Corporate Firewalls are increasingly starting to employ SSL proxies as trusted intermediaries to prevent attacks through encrypted channels [19].

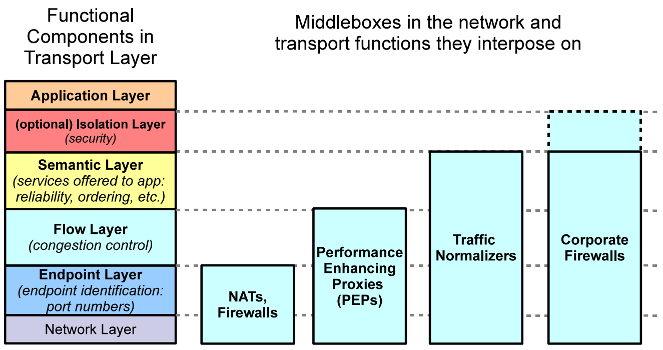

Functional Components in Transport Layer
Middleboxes in the network and transport functions they interpose on
Application Layer
(optional) Isolation Layer
(security)
Semantic Layer
(services offered to app: reliability, ordering, etc.)
Flow Layer
(congestion control)
Endpoint Layer
(endpoint identification: port numbers)
Network Layer
NATs, Firewalls
Performance Enhancing Proxies (PEPs)
Traffic Normalizers
Corporate Firewalls

Figure 1: A taxonomy of Transport functions showing how middleboxes interact with the Transport

2.2 Network Considerations

Figure 2 shows how transports are designed and deployed on the Internet, and a comparison with Figure 1 shows how new transports conflict with legacy middleboxes.

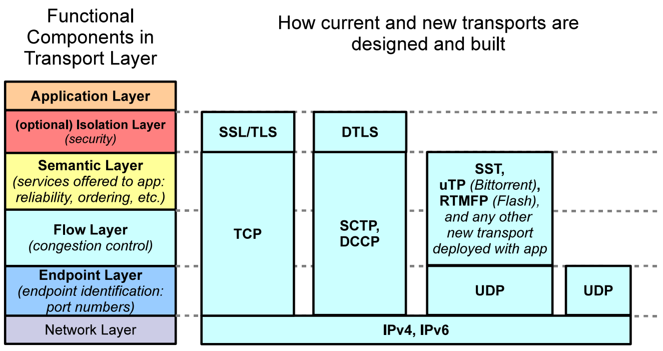

Functional Components in Transport Layer
How current and new transports are designed and built
Application Layer
(optional) Isolation Layer
(security)
Semantic Layer
(services offered to app: reliability, ordering, etc.)
Flow Layer
(congestion control)
Endpoint Layer
(endpoint identification: port numbers)
Network Layer
SSL/TLS
DTLS
TCP
SCTP, DCCP
SST, uTP (Bittorrent), RTMFP (Flash), and any other new transport deployed with app
UDP
UDP
IPv4, IPv6

Figure 2: Transport protocols in the Tng taxonomy

While NATs and firewalls often support both TCP and UDP, they support TCP flows more efficiently due to the connection state information it provides. Since NATs cannot precisely determine a UDP session's lifetime, an application wishing to keep a UDP connection alive across a NAT or firewall must send keepalive packets at least every two minutes [3], whereas idle TCP connections require keepalives only once every two hours [11]. TCP thus has practical advantages in both bandwidth and power consumption, especially important for mobile devices, which are frequently located behind NATs and firewalls and must keep connections open to listen for incoming notifications.

While simple home NATs and Firewalls care mainly about port numbers (application endpoint information), significant parts of the Internet are behind more sophisticated middleboxes [4] which use TCP connection state information and assume TCP mechanisms, such as ack clocking and flow control to achieve their ends.

Stateful Firewalls maintain and use connection state for filtering traffic and to avoid attacks. TCP's connection-oriented state machine makes this information easily available. Since UDP is connectionless, Stateful Firewalls use non-trivial methods such as Deep Packet Inspection (DPI) and rely on arbitrary timeouts and ICMP messages to determine the state of a session that uses UDP [21], thus creating disincentives for operators to allow UDP traffic into their networks.

Traffic Normalizers [14] use detailed information about the TCP state machine to thwart network attacks, and Corporate Firewalls increasingly employ SSL proxies as trusted intermediaries to prevent attacks through encrypted channels [19]. Using UDP leads to heavy use of DPI, since both Firewalls and Traffic Normalizers have less information to work with than they would have with TCP, and they are thus heavily incentivized against allowing UDP traffic.

A new transport that runs on UDP appears as one among the plethora of extant applications that use UDP, and thus will not be able to navigate broad sledgehammer-like firewall rules often found in corporate (and often hotel) firewalls that disallow any incoming UDP traffic.

PEPs [4] improve TCP flow performance over unconventional network paths, such as lossy wireless or satellite links, often by intervening in TCP mechanisms such as flow control and ack clocking. PEPs ignore UDP traffic, and consequently transports such as SCTP, SST, and DCCP, that use TCP-like congestion control mechanisms, appear to perform poorly on these paths when run atop UDP.

All these middleboxes need to be made aware of a new transport for the transport to be deployable and performant in the network on top of UDP.

2.3 Yet Another Layer of Multiplexing

Current proposals [22, 29] use UDP port numbers merely to demultiplex the new transport protocol on top; the identity of the communicating endpoints is hidden away in a *new* set of port number fields (separate from the UDP port numbers) embedded in the new transport's protocol headers. There are thus two sets of port numbers at the ends: the UDP port numbers are used to identify the transport protocol—e.g., UDP port 9899 indicates SCTP atop UDP—plus the inner transport's SCTP port numbers, used to identify the application process. The UDP port numbers are thus not usable as a basis for identifying end-applications, as is commonly done at firewalls. As a result, while this solution works for NATs, Firewalls will need to understand the new protocol's interior headers to enforce network policy based on port numbers, and until they are able to, administrators will be tempted simply to block the new transport's UDP port. Thus, where an HTTP session over TCP works, an HTTP session over SCTP/UDP will not, even if the firewall's policy is intended to allow web traffic in general.

Future generations of firewalls may eventually understand the inner transport and enforce policy based on the inner transport's port numbers, but this simply brings us back to where we started, with devices in the network "understanding" and allowing traffic from a relatively static and difficult-to-evolve set of transports atop IP.

Proposals for UDP encapsulation of new transports [22, 29] miss the point that middleboxes need endpoint information, and encapsulating that information does more harm by triggering an arms race with firewall providers.

2.4 End Host Considerations

Getting user-space transport implementations to perform competitively with kernel-space implementations is a non-trivial issue, and requires much more low-level access than is available to most applications for close to kernel-level performance [7]. Ack-clocked transports, such as SCTP and SST, rely on tight timing in transmission and delivery of acks. Coalescence of ack- and data-delivery, which results from poor interaction between the ack-clocking of data and process scheduling on the endhosts, leads to increased burstiness in the transport [24], breaking TCP's self-clocking mechanism [15], and resulting in poor performance of TCP-like congestion control [30]. Deploying highly performant transports on top of UDP is difficult, and bandwidth intensive applications will tend to opt for TCP, despite its strait-jacketed service options, for performance reasons.

Further, new transports that use delay-based congestion control mechanisms will simply not work in user-space due to the increased noise in delay signals that comes with data copying across the kernel-user boundary and with context-switching latency.

2.5 Legacy Comes With Benefits Too

TCP implementations have been maturing and optimized over the past 2-3 decades, and detailed instrumentation is available for learning from and tweaking TCP stacks [2, 12] Offloading parts of the TCP engine is becoming increasingly important and relevant [20], improving TCP performance and efficiency in high-speed networks.

While other parts have been difficult to get deployed, variants to the original TCP congestion control [15] are commonplace in general-purpose OS kernels—likely in large part due to the fact that congestion control modifications can usually be deployed without wire-format modifications to TCP. The major general-purpose Operating Systems—MacOSX, Windows7, and Linux—all use very different congestion control algorithms: TCP NewReno, Compound TCP, and CUBIC TCP, respectively. As of kernel v2.6.13, Linux has made it easy for a user to choose from a number of congestion control alternatives in the kernel at run-time[2], and Mac OSX allows enabling and disabling TCP variants such as NewReno and SACK through *sysctl*s. Kernel mechanisms for plugging different congestion control variants for TCP already exist in popular Operating Systems, and where they do not exist yet, they represent a worthy goal for kernels that are already dealing with diverse network conditions ranging from low-bandwidth and lossy cellular connectivity on mobile and handheld devices to high-bandwidth connectivity on desktops within enterprise networks.

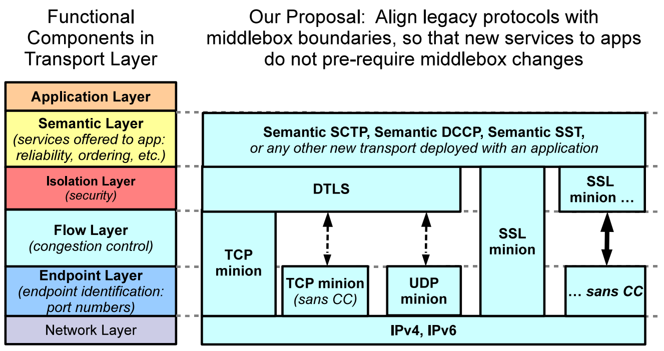
Legacy TCP thus comes with a performant and functionality-rich body of software artifacts and tools, as well as widespread human experience in the industry, which generally outweighs the costs of deploying and performance-tuning a new transport atop UDP.

3. OVERVIEW OF THE MINION SUITE

The goals of the minion suite are as follows:

- *Overcome network inertia:* Provide robust, low-latency (out-of-order) forwarding of any transport or application PDU through middleboxes supporting only TCP or SSL.
- *End-to-end enforcement of transport- and application-neutrality:* Provide ability to deploy TCP, SCTP, SST, and new transport services atop secure end-to-end channels.
- *Substrate-agnostic transport and application operation:* A transport or application should not generally *need to know* which specific minion it is using on a given path.

The minion suite employs modified forms of UDP, TCP, SSL, and DTLS to provide a low-latency datagram service, as shown in Figure 3.



Functional Components in Transport Layer
Our Proposal: Align legacy protocols with middlebox boundaries, so that new services to apps do not pre-require middlebox changes
Application Layer
Semantic Layer
(services offered to app: reliability, ordering, etc.)
Isolation Layer
(security)
Flow Layer
(congestion control)
Endpoint Layer
(endpoint identification: port numbers)
Network Layer
Semantic SCTP, Semantic DCCP, Semantic SST,
or any other new transport deployed with an application
DTLS
SSL minion
SSL minion ...
TCP minion
TCP minion (sans CC)
UDP minion
... sans CC
IPv4, IPv6

Figure 3: The minion protocols and their place in the Tng taxonomy

TCP-minion provides a datagram service while appearing as TCP on the wire, with the option of turning congestion control OFF, and with the option of providing transport layer security through DTLS. UDP-minion provides a datagram service with no congestion control but exposes endpoint (port number) information. SSL-minion provides a secure datagram service while appearing as SSL on the wire,

[2] *ls /lib/modules/'uname -r'/kernel/net/ipv4/* on a Linux-2.6.34 system shows an impressive 11 distinct TCP congestion control mechanisms available as kernel modules.

with the option of turning congestion control OFF. Transport protocols, such as TCP, SCTP, SST, and DCCP, trimmed to run on the minion suite, continue to provide their services to applications in the way they were designed to.

In the following three sections, we now describe each of the three minions in more detail.

4. A UDP MINION

Perhaps the simplest of the minions, UDP-minion is a UDP encapsulation of the transport or application protocol, but with one key difference from current proposals [22,29]—UDP-minion does not add an additional point of demultiplexing. While the recent proposal for Generic UDP Tunneling (GUT) [18] suggests such a possibility, our UDP-minion deliberately avoids a UDP demultiplexer followed by an upper layer demultiplexer, in keeping with Feldmeier [8] and Tennenhouse [28].

5. A TCP MINION

The goal of TCP-minion is to provide a datagram service to the application above it, while appearing as TCP on the wire. Our design of TCP-minion does not change the TCP protocol per se, but achieves a datagram service by transforming application messages into encoded messages which are delimited by marker-bytes in the underlying bytestream. At a TCP-minion receiver, the encoded messages are extracted from between marker-bytes, and decoded before delivery to the application. Thus, in TCP-minion, application data is transformed before transmission, and the API at the ends is modified, but the TCP protocol is not modified—TCP-minion sits close to the transport-application boundary, and *uses* the underlying TCP bytestream for transmitting and extracting application messages.

Providing a datagram service *within* a TCP bytestream requires delimiting application messages in the bytestream. While record delimiting is commonly done by application protocols such as HTTP, SIP, and many others, a key property that we require to provide a true datagram service is that a receiver must be able to extract a given message independently of other messages. That is, as soon as a complete message is received at a TCP-minion receiver, the message delimiting mechanism must allow for extraction of the message from the bytestream, irrespective of what earlier messages have or have not been received.

When an application sender needs to send a message to the receiver, the TCP-minion sender employs the use of marker-bytes to delimit the message in the TCP bytestream and follows a three-step process:

1. The application message is encoded using the minion-encoding described below, so that all occurrences of the marker byte are eliminated from the message,

2. The marker byte is inserted at the beginning and at the end of the message,

3. The encoded message, along with the marker bytes, is sent into the TCP bytestream.

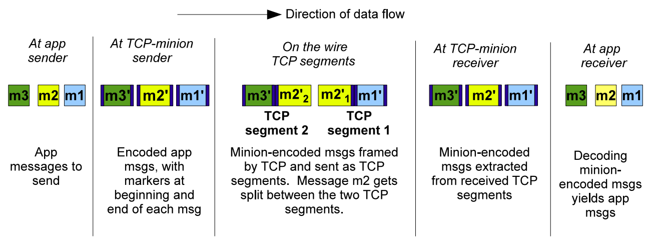

Direction of data flow
At app sender
At TCP-minion sender
On the wire TCP segments
At TCP-minion receiver
At app receiver
m3
m2
m1
m3'
m2'
m1'
m2'2
m2'1
TCP segment 2
TCP segment 1
App messages to send
Encoded app msgs, with markers at beginning and end of each msg
Minion-encoded msgs framed by TCP and sent as TCP segments. Message m2 gets split between the two TCP segments.
Minion-encoded msgs extracted from received TCP segments
Decoding minion-encoded msgs yields app msgs

Figure 4: An illustrative example showing the steps in a TCP minion transfer

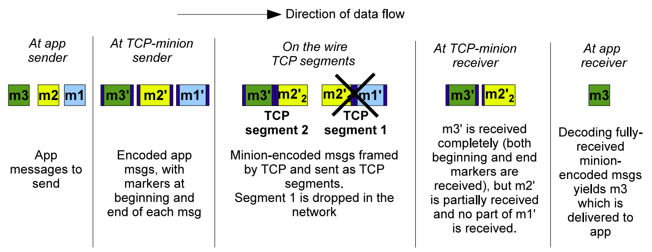

Direction of data flow
At app sender
At TCP-minion sender
On the wire TCP segments
At TCP-minion receiver
At app receiver
m3
m2
m1
m3'
m2'
m1'
m2'2
m2'1
TCP segment 2
TCP segment 1
App messages to send
Encoded app msgs, with markers at beginning and end of each msg
Minion-encoded msgs framed by TCP and sent as TCP segments.
Segment 1 is dropped in the network
m3' is received completely (both beginning and end markers are received), but m2' is partially received and no part of m1' is received.
Decoding fully-received minion-encoded msgs yields m3 which is delivered to app

Figure 5: An illustrative example showing TCP minion behavior when loss occurs.

The TCP-minion receiver uses the following 3-step process for extracting and delivering application messages to the receiving application:

1. The received TCP segments, both the ones received in-order and the ones received mis-ordered, are scanned for marker bytes.

2. If two marker bytes are found, and if all bytes between the two markers have been received, the minion-encoded message between the two markers is extracted.

3. The extracted message is decoded and delivered to the application.

Figure 4 shows an illustrative example, where three application messages are transferred using TCP-minion. Note that TCP-minion does not assume anything about TCP segment boundaries; minion-encoded messages may be segmented arbitrarily by TCP, as shown in the figure. TCP-minion is designed to work with arbitrary segmentation, to be resilient to middleboxes that transparently re-assemble and re-segment the transmitted TCP segments. Figure 5 shows the same transfer, but this time with a loss of one segment during the transfer. TCP-minion delivers the only fully received message to the application, even though the message is not received in order. Legacy TCP would not have delivered any part of the bytestream in this scenario.

5.1 The TCP-Minion Encoding

TCP minion uses *Consistent Overhead Byte Stuffing (COBS)* for eliminating all occurrences of the marker byte in the application message. This mechanism was originally proposed by Cheshire and we direct the reader to [5] for a complete description including detailed evaluations; we provide a brief summary of the COBS mechanism below.

Assuming zero as the marker byte, COBS first fragments every message at the existing zeros in the original app message, and then eliminates each zero while inserting a "pointer" to the deleted zero at the beginning of the corresponding fragment. Thus, each zero in the message is effectively replaced by a non-zero number at a different location, which results in a size-preserving transformation that eliminates all zeros in the message. When zeros are sparse in the original message, however, special "pointers" are inserted in the message without any deletions, leading to some overhead. This overhead is limited to a maximum of 1 additional byte in every 254 bytes, or 0.4%.

TCP minion first uses COBS to encode an incoming application message, so that all zeros are eliminated, and then both prepends and appends a zero byte to the message to delimit the message at both ends. While inserting a zero byte only at the beginning *or* end of each message (but not both) would be sufficient to delimit messages in a stream delivered in-order, but would prevent TCP-minion from passing received messages to the application as early as possible in the common case in which network middleboxes do *not* resegment the TCP stream. For example, if TCP-minion inserted a zero byte only at the end of each message, then a complete message arriving out-of-order immediately after a dropped packet in TCP sequence number space would appear indistinguishable to the receiver from the incomplete tail-end of a longer message, requiring the receiver to wait until the immediately preceding "hole" is filled before delivering the (already-complete) message to the receiving application. By placing zeros at both the beginning and end of each message, the receiver can determine with certainty that it has received a complete message and pass it on to the application as soon as it arrives.

6. AN SSL MINION

While we expect TCP-Minion to provide a relatively low-overhead but effective delivery substrate for deploying new transports atop existing middleboxes, it has an important disadvantage: middleboxes often attempt to—and *expect* to be able to—"see inside" of TCP streams to perform Deep Packet Inspection and even manipulation of application-level protocols. It has become a *de facto* rule that anything in a TCP or UDP stream that is *not* encrypted is "fair game" for middleboxes to inspect and manipulate. As a result, the only way for an application to communicate "end-to-end" and be certain that its session will *not* be inspected or manipulated by middleboxes is by encrypting and authenticating it cryptographically. Network-layer encryption mechanisms such as IPsec [16] have suffered deployment challenges similar to those new transports have faced and are still uncommon outside the niche of corporate VPNs, and *new* encrypted transports are likely to be blocked by many firewalls. The only way to form an end-to-end encrypted path on today's Internet that is nearly universally supported, therefore, is for that encrypted path to appear to the network as an SSL stream (atop TCP), because that is the mechanism the Web uses for now-crucial E-commerce traffic. While a network administrator or ISP might disable nearly any other port while still claiming to provide "Internet access," he would be hard-pressed to disable SSL connections to port 443 while making such a claim.

While we could simply layer SSL atop TCP-Minion, doing so would have two severe shortcomings. First, although TCP-Minion supports out-of-order message delivery, SSL supports only in-order delivery and thus, unmodified, would simply place streams back in-order at the receiver and defeat the primary performance benefits TCP-Minion provides. Second, since TCP-Minion uses COBS to demark message boundaries, layering SSL atop this COBS encoding would make the SSL stream appear fundamentally different than it does when layered directly atop TCP, e.g., for HTTP traffic, and thus would make it impossible for SSL-Minion streams to "masquerade" as HTTPS connections to port 443 for example. Thus, to achieve out-of-order delivery while maximizing traversal capability, we must modify SSL to provide out-of-order delivery while remaining indistinguishable on the wire from conventional SSL over TCP, *without* adding a new network-visible encoding layer such as COBS.

To accomplish this, we observe that the SSL protocol already breaks its communication into *records*, separately encrypts and authenticates each record, and prepends a version/type/length header to each record it transmits on the underlying TCP stream. To convert SSL into an out-of-order delivery protocol, therefore, we only need to modify SSL so that the receiving host can recognize, authenticate, and deliver to the application the contents of any complete record that it receives, even if not all the contents of the underlying TCP stream before that record has yet been received. Accomplishing this goal involves two specific challenges: first, enabling the receiver to recognize a record out-of-order in a received TCP stream; and second, ensuring that the receiver is able to decrypt and authenticate such a record without necessarily having received all preceding records.

To recognize records out-of-order, we first use SSL's basic record structure as a "weak" recognizer, to infer where a record *might* begin in a partial TCP stream. Each SSL record starts with a type/version/length header, which we scan for in any TCP segment SSL-Minion receives. Encrypted SSL data could contain a byte sequence that looks like a type/version/length header, however, which our preliminary scan might falsely recognize as the beginning of a record. Fortunately, SSL already provides a means be certain whether the apparent record header is indeed valid: simply attempt to decrypt and authenticate the message. If we have attempted to decode a "false" record header, then SSL's cryptographic record authentication mechanism will fail. When decoding SSL records in-order, such a failure would be fatal (terminating the underlying TCP connection). In SSL-Minion where false positives are possible, however, we simply take an out-of-order record decryption failure as an indication of a false record header, and ignore it.

SSL-Minion's second technical challenge is to ensure that the receiver's ciphersuite *can* decrypt and authenticate a record without all prior records already having been received. Stream ciphers cannot be used with SSL-Minion, for example, because they fundamentally require decryption to be done in-order. Fortunately, most SSL ciphersuites today use Cyclic Block Chaining (CBC) with a fresh Initialization Vector (IV) transmitted as part of each record, so that each record's starting encryption state is independent of all others, thus allowing for out-of-order decryption.

SSL's authentication mechanism poses a slight problem, however, because the implicit "pseudo-header" that it authenticates along with the content of every packet includes a *sequence number* that is incremented once for each record transmitted. When SSL-Minion receives a TCP segment out-of-order containing an SSL record, however, the receiver knows the byte-oriented TCP sequence number of that record in the TCP stream, but does not know the record-oriented SSL sequence number. Since records are variable-length, the SSL record to be authenticated out-of-order (following at least one "hole" in TCP sequence number space) might be preceded by a small number of large records or a larger number of small records. Two solutions to this problem present themselves, which we intend to explore. First, we could modify the SSL ciphersuite slightly, to use byte-oriented TCP sequence numbers in place of the record counter; this would serve the same cryptographic purposes (e.g., to prevent replay attacks) while using only information the receiver is guaranteed to have even when a record arrives out-of-order. The second approach is to leave the SSL ciphersuite unmodified, but attempt to *predict* the correct record number to use in out-of-order packet decryption and authentication, perhaps trying several possible record numbers if necessary. The first approach is likely to be cleaner and more efficient, but the latter has the advantage of requiring no modification to SSL ciphersuites or the SSL negotiation process.

7. IS MINION DEPLOYABLE?

To enable deployment of new transport services atop the minion suite, the minion protocols needs to be deployed first. The changes we need to make to UDP, TCP, and SSL in endhosts are described below:

- UDP-minion requires no modifications to UDP; the transport and/or application atop UDP-minion need to use port numbers that identify the application endpoints. Any additional headers required for new services are encapsulated within the UDP-minion payload.

- TCP-minion requires COBS encoding, which can be done in user-space, and requires no modifications at the sender-side. At the receiver, the TCP implementation will need to be modified to allow delivery of out-of-order data up to the COBS decoder. We have implemented a prototype of such an API extension to TCP in Linux, by adding an *SO_UNORDERED* socket option to the TCP socket.

- SSL-minion similarly needs the same kernel API extension to TCP at the receive side; all other changes required can be implemented in the user-space SSL library.

- In addition to changes to the TCP receiver as discussed above, we are exploring sender-side modifications that are required to disable TCP's congestion control.

Arguments can be made for the deployability of the end-host changes to legacy TCP and SSL for TCP-minion and SSL-minion, such as the momentum behind low-latency web transports such as Google's SPDY [1]. Perhaps the strongest arguments for the deployability all minion protocols are that none of them requires modifying any middlebox, and that deploying the minion protocols enables deployment of hitherto undeployable transport services.

Acknowledgments: This research is sponsored by the NSF under grants CNS-0916413 and CNS-0916678.

8. REFERENCES

[1] SPDY: An Experimental Protocol For a Faster Web. http://www.chromium.org/spdy/spdy-whitepaper.
[2] The Web100 Project. http://www.web100.org.
[3] F. Audet, ed. and C. Jennings. Network address translation (NAT) behavioral requirements for unicast UDP, Jan. 2007. RFC 4787.
[4] B. Carpenter and S. Brim. Middleboxes: Taxonomy and Issues, Feb. 2002. RFC 3234.
[5] S. Cheshire and M. Baker. Consistent Overhead Byte Stuffing. In *ACM SIGCOMM*, Sept. 1997.
[6] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2, Aug. 2008. RFC 5246.
[7] A. Edwards and S. Muir. Experiences implementing a high performance TCP in user-space. *Computer Communications Review*, 25(4):196–205, Oct. 1995.
[8] D. C. Feldmeier. Multiplexing issues in communication system design. In *SIGCOMM*, Sept. 1990.
[9] B. Ford. Structured streams: a new transport abstraction. In *SIGCOMM*, Aug. 2007.
[10] B. Ford and J. Iyengar. Breaking up the transport logjam. In *HotNets-VII*, Oct. 2008.
[11] S. Guha, Ed., K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. NAT behavioral requirements for TCP, Oct. 2008. RFC 5382.
[12] H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deploying Safe User-Level Network Services With icTCP. In *OSDI 04*, pages 22–22. 2004.
[13] M. Handley. Why the Internet only just works. *BT Technology Journal*, 24(3):119–129, 2006.
[14] M. Handley, V. Paxson, and C. Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-end Protocol Semantics. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 9–9. 2001.
[15] V. Jacobson. Congestion avoidance and control. pages 314–329, Aug. 1988.
[16] S. Kent and K. Seo. Security architecture for the Internet protocol, Dec. 2005. RFC 4301.
[17] E. Kohler, M. Handley, and S. Floyd. Datagram congestion control protocol (DCCP), Mar. 2006. RFC 4340.
[18] J. Manner, N. Varis, and B. Briscoe. Generic UDP Tunnelling (GUT), July 2010. Internet-Draft draft-manner-tsvwg-gut-02 (Work in Progress).
[19] K. Marko. Using SSL Proxies To Block Unauthorized SSL VPNs. *Processor Magazine, www.processor.com*, 32(16):23, July 2010.
[20] J. Mogul. TCP Offload is a Dumb Idea Whose Time Has Come. In *HotOS IX*, May 2003.
[21] S. Northcutt, L. Zeltser, S. Winters, K. Kent, and R. Ritchey. *Inside Network Perimeter Security*. SAMS Publishing, 2005.
[22] T. Phelan. DCCP Encapsulation in UDP for NAT Traversal (DCCP-UDP), Aug. 2010. Internet-Draft draft-ietf-dccp-udpencap-02 (Work in Progress).
[23] L. Popa, A. Ghodsi, and I. Stoica. HTTP as the narrow waist of the future Internet. In *HotNets-IX*, Oct. 2010.
[24] R. Prasad, M. Jain, and C. Dovrolis. Effects of interrupt coalescence on network measurements. In *Workshop on Passive and Active Measurements (PAM)*, 2004.
[25] E. Rescorla and N. Modadugu. Datagram transport layer security, Apr. 2006. RFC 4347.
[26] J. Rosenberg. UDP and TCP as the new waist of the Internet hourglass, Feb. 2008. Internet-Draft (Work in Progress).
[27] R. Stewart, ed. Stream control transmission protocol, Sept. 2007. RFC 4960.
[28] D. L. Tennenhouse. Layered multiplexing considered harmful. In *1st International Workshop on Protocols for High-Speed Networks*, May 1989.
[29] M. Tuexen and R. Stewart. UDP Encapsulation of SCTP Packets, Jan. 2010. Internet-Draft draft-tuexen-sctp-udp-encaps-05 (Work in Progress).
[30] M. Zec, M. Mikuc, and M. Zagar. Estimating the Impact of Interrupt Coalescing Delays on Steady State TCP Throughput. In *SoftCOM*, 2002.