This project has received funding from the European Unions Horizon 2020
research and innovation programme under grant agreement No 688421.

Measurement and Architecture for a Middleboxed Internet

H2020-ICT-688421

Middlebox Cooperation Protocol Specification and Analysis

Author(s):	ETH	M. Kühlewind (ed.), Brian Trammell
	UNIABDN	Gorry Fairhurst, Tom Jones
	ZHAW	Stephan Neuhaus, Roman Müntener
	ALCATEL	Thomas Fossati

Document Number:	D3.2
Internal Reviewer:	Diego Lopez
Due Date of Delivery:	28 Februar 2018
Actual Date of Delivery:	2 March 2018
Dissemination Level:	Public

Disclaimer

Contents

Executive Summary

This deliverable specifies the Path Layer UDP Substrate (PLUS), a UDP-based, shim-layer protocol for middlebox cooperation. Further, this deliverable explains how middlebox cooperation mechanisms that have been developed by the project for use with PLUS can be applied to other new and existing protocols at different layers. It additionally provides a security analysis for middlebox cooperation mechanisms, with PLUS as one example. Moreover, this deliverable also specifies a protocol-independent transport to application interface, called Post Sockets, that supports flexible protocol selection, without requiring changes to applications, in order to enable faster deployment of new protocols such as PLUS within a more Flexible Transport Layer (FTL).

1 Introduction to the Middlebox Cooperation Concept in the Path Layer

The deployment of encrypted transport protocols imposes new challenges for network management. Key in-network functions such as those implemented by firewalls and passive measurement devices currently rely on information exposed by the transport layer. Encryption, in addition to improving privacy, helps to address ossification of network protocols caused by middleboxes that assume certain information to be present in the clear. However, “encrypting it all” risks diminishing the utility of these middleboxes for the network management tasks for which they were designed. A middlebox cannot use what it cannot see.

With the design of an explicit middlebox cooperation protocol, we propose an architectural solution to this issue, introducing a new *Path Layer* for transport-independent, in-band signaling between Internet endpoints and network elements on the paths between them. Such a layer reinforces the boundary between the hop-by-hop network layer and the end-to-end transport layer, by separating stateful network functionality from forwarding or simple per-packet handling and enabling an opportunity to encrypt all high-layer information that is not indented for network usage respectively.

In the following subsections we will outline our design goals for such a path layer header and describe the abstract mechanisms for sender-to-path and path-to-receiver signaling, providing integrity-protected, enhanced signaling under endpoint control. Further in this chapter, we detail an additional use case, compared to the use cases described in D3.1, for use of the path layer for explicit network performance measurements. Since the start of the project it has become more and more clear that network monitoring is the most critical use case for the deployment of explicit path signal for existing network operations. As such we outline this use case and its requirements and needed mechanisms in detail in this deliverable inline with the *Principles for Measurability in Protocol Design* as developed within the project [2].

In the next chapter we specify the Path Layer UDP Substrate (PLUS), a path layer header on top of UDP to provide a common wire image for new, encrypted transports that can provide an integrity-protected replacement for signals that are not available with encrypted traffic anymore, as well as additional enhanced signaling to support our use cases as described in D3.1 and section 1.3. This specification is base on previous work in the IETF, as documented in several Internet Drafts [45, 40, 21] and published in an academic publication for further dissemination the the research community [26].

Since the start of the project, we have disseminated our proposed approach via standardization. While middlebox cooperation is still under discussion in standardization, especially with the development of a new encrypted transport protocol as well as increased deployment of encryption on all layers, the discussion about the introduction of an explicit path layer signaling protocol did not come to a conclusion that led to the creation of a working group in the IETF. Deployment of the PLUS protocol, however, must be supported by both the endpoint, in terms of OS and software vendors, as well the network or network devices that can use and provide PLUS-based information. Without a standardized protocol, deployment cannot be fostered by only one of the parties. However, based on on-going work and discussion, we illustrate in chapter 3 how the mechanisms and principles develop for and with PLUS can be applied to existing protocols or approaches that are currently under development in standardization.

Chapter 4 provides a detailed analysis of attacks on middlebox cooperation mechanisms, as provided by PLUS as well as based on existing protocols that (often unintentionally) provide cooperation facilities, and new encrypted protocols that intentionally expose information to the network.

Further, with the goal to enable more flexible protocol development, deployment, and run-time selection of protocols, as foreseen on top of PLUS, the project is working on a protocol-independent, message-based transport interface. Chapter 5.1 describes the basic components of the Post Sockets interface and discusses basic usage scenarios as we are investigating in experimentation.

1.1 Design Goals

In this section, we elaborate the design principles for our middlebox cooperation approach. The design goals are derived from the first principles and functional requirements as previously described in D3.1. They are set up to cover the functionality needed for middlebox cooperation use cases, to maximize deployability on today's Internet based on measured observations about devices currently deployed, and based on present best practices in protocol design and implementation:

1. **An endpoint should be able to explicitly expose any signals used by on-path devices.** We refer to this as *sender-to-path signaling*.

2. **An endpoint should be able to request signals from devices on the path.** We refer to this as *path-to-receiver signaling*.

3. **An on-path device should not be able to forge, change, or remove a signal sent by an endpoint.** This implies a need for end-to-end integrity protection for these signals.

4. **The endpoint should control signaling between endpoints and the path, or from one on-path device to another.** The use of signaling on a given packet or flow is therefore optional. An on-path device should not be able to force an endpoint to send a signal, or to use the mechanism to send a signal of its own volition without explicit cooperation from the endpoints. This can be achieved by integrity protection over a *scratch space* allowing devices on path to send specified signals to receivers.

5. **It should be possible for an endpoint to request and receive signals from a previously unknown on-path device.** This implies that, in the absence of a cryptographic introduction protocol or public-key infrastructure for on-path devices in the Internet – which we consider to be impractical – authentication of signals from these on-path devices is not possible.

6. **The mechanism should present no significant surface for amplification attacks.** We can achieve this by specifying that no signal can request transmission of a packet beyond forwarding the packet carrying the signal. This relies on in-band signaling, piggybacked on higher-layer traffic.

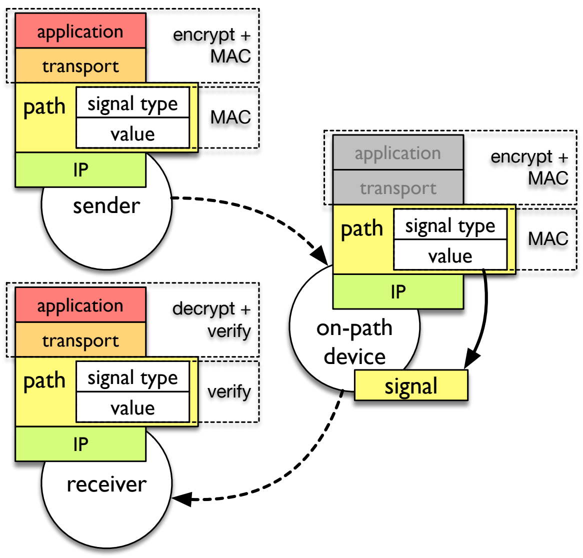

Figure 1: Sender-to-path signaling: signals are readable by on path-devices, but modification is detected by the receiver

1.2 Abstract Mechanisms and Integrity Protection

From these first principles we define a set of simple abstract mechanisms for bidirectional in-band signaling under endpoint control.

1. **Sender-to-path** signaling is unencrypted but with integrity protection, relying on the receiver to verify integrity. On-path devices can read these signals, but not verify their authenticity. Modifications to these signals will be detected by the receiver, which can respond by raising an error to the transport and/or application layer.

   To make a declaration about a packet or flow to all path elements, the sender adds a key-value pair to a packet within the flow, as shown in Figure 1. The fact that this is a sender-to-path declaration is part of the definition of the key. Only one declaration can appear in a single packet. The declarations, together with other transport and network layer information which must not be modified by the path, are protected by a message authentication code (MAC) sent along with the packet, generated with a key derived from a secret known only to the endpoints.

   The receiver then verifies the MAC on receipt. Verification failure implies an attempt to modify the header used by this mechanism. Should a violation should be communicated to the sender and may cause the transport association to reset.

2. **Path-to-receiver** signaling is also initiated by the sender. The sender places the requested signaling type in the packet and creates a “scratch space” writable by on-path devices in the packet. The length of the scratch space is fixed by the sender, to avoid problems with downstream loss and/or fragmentation due to packet size changes along the path. Path accumulation signals, such as those proposed in IPIM [2] and further specified in Section 2.3, can also use this mechanism; in this case, the scratch space is initialized by the sender to some value, and each device aware of the signal along the

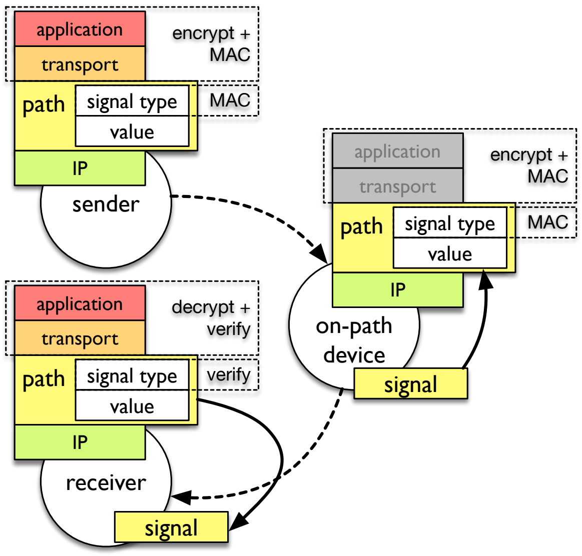

Figure 2: Path-to-receiver signaling: signal content is writable by on-path devices, but signal presence is protected end-to-end. Signals can neither be added nor removed on path.

path updates it according to an algorithm specified by the type, until the final accumulated value is received at the receiver. Since some path-to-receiver signals may be of more use to the sender than the receiver, path-to receiver signaling can be augmented by **feedback**, the return of the final value of a signal from the path by the receiver back to the sender.

As shown in Figure 2, to allow one or more path elements to make a declaration about itself with respect to a packet or a flow to the receiver, the sender adds a key-value pair to a packet within the flow. The fact that this is a path-to-receiver declaration is part of the definition of the key. Further, the value has a fixed length of N bytes (which my also be part of the definition of the key). When calculating the MAC for a path-to-receiver declaration, its value is assumed to be an N-byte array of zeroes. The MAC therefore protects the presence of the key and the length of the value, but not its content.

The initial value of a path-to-receiver declaration is up to the sender, and is generally defined by the declaration itself. The behavior of a path element in filling in a path-to-receiver declaration given which value is already present is also part of the declaration definition. Declarations may accumulate by some operation (e.g., max, min, sum for measurement declarations), be determined by the first or last path element, or be addressed to a specific path element to fill in.

These mechanisms rely on an upper layer to establish a cryptographic context in order to establish a shared secret from which MAC keys can be derived. This cryptographic state may be established with each transport session or may be resumable across multiple transport sessions, depending on the upper layer's design. If no context exists, though, the integrity of the declarations made via these mechanisms cannot be protected by MAC. We propose two possible solutions to this situation:

1. The mechanisms can be implemented such that MAC is mandatory. In this arrangement,

Horizon 2020
European Union funding
for Research & Innovation

no sender-to-path and/or path-to-receiver declarations can be made until cryptographic context is bootstrapped. The vocabulary of declarations can therefore not include declarations that must be sent on the first packet.

2. The mechanisms can be implemented such that the MAC is eventual. In this arrangement, sender-to-path declarations can be made before cryptographic context establishment, but are open to undetected modification along the path; path-to-receiver declarations are not allowed before cryptographic context establishment. A MAC for previous sender-to-path declarations must be sent after cryptographic context establishment; lack of receiving this MAC within a defined (and small) number of packets from the sender is treated by the receiver as verification failure and leads to transport association reset.

If a path element and the sender share some cryptographic context through some out-of-band means, sender to path declarations can also be integrity protected using a MAC generated by the sender and carried within the declaration itself. In this case, if the path element fails to verify the MAC, it simply ignores the declaration.

Similarly, if a path element and the receiver share some cryptographic context through some out-of-band means, path to receiver declarations can also be integrity protected using a MAC generated by the path element and carried within the declaration itself. The use of a MAC is part of the definition of the key. In this case, if the receiver fails to verify the MAC, it causes a transport association reset.

This design pattern can also be used to address sender-to-path declarations to specific path elements: a declaration with an encrypted value is inherently addressed to only those path elements that possess the private or secret key to decrypt the value.

In each of these cases, the presence of a MAC within the declaration value and/or encryption of the declaration value is part of the definition of the key. Further definition of mechanisms for building cryptographic protocols over these mechanisms is out of scope for this specification of the path layer header iteself.

1.3 Use Case: Network Performance Measurements

In addition to the use cases that have been outlined in D3.1 and that have been used to derive requirements and subsequently the design principles for PLUS, the project is also focusing on passive performance measurement as a central use case for PLUS given that presently passive performance measurements are mostly based on information available in the unencrypted TCP header. This section discussed which information are desirable for network measurement independent of the encoding on the wire. However, a path layer protocol like PLUS would of course be the most preferred place to provide these signals to network elements that preform network monitoring.

Basic metrics suffice for most performance measurement tasks: transmission rate, latency/jitter, and packet loss rate. Transmission rate can be trivially measured by counting bytes associated with a traffic flow, whether the headers are encrypted or not, then dividing this count by the observation time interval. We therefore focus on latency and loss.

Latency measurement, as round-trip time, requires the ability to match packets in one direction with packets in the opposite direction, for both directions of the flow. Current inference-based passive measurement approaches [37, 44] generally use the TCP Timestamp Option [7], when available [15, 22]. Loss measurement, on the other hand, requires inference about the loss detection and retransmission algorithms in use by the sender [4].

Loss and latency measurements are basic, key tools for network management, troubleshooting, and diagnostics. For an operator to address performance issues within its network, it needs to know the nature and magnitude of the performance issue, as well as its location – the most important information being whether the issue is caused within the operator's own network, or is outside its domain of control. Loss and latency measurements at multiple points in the network, and basic statistical functions over these measurements (loss pattern, latency variance, and so on) are inputs to this process, as well as to diagnostic processes for a range of other faults: inconsistent BGP routing, inappropriate forwarding rules, buffer management issues, and so on.

In this section we discuss mechanisms that are designed to support explicit passive measurability of latency and loss to replace inference-based approaches. These are also supported by the PLUS basic header, see 2.1. We follow the principles proposed in [2] that measurement should be explicit, in-band, visible, cooperative, and under the absolute control of the endpoint. Indeed, we find that designing signals expressly for measurement, much of the messy inference involved in turning a TCP packet stream into metric samples becomes unnecessary.

1.3.1 Latency Measurement

To measure latency, we propose a simplification of the arrival information primitive in [2]: a simple Packet Serial Number (PSN) that increases by one with every packet sent, including retransmissions and control packets (unlike TCP), and a packet serial echo (PSE) that reflects the last packet number seen in the opposite direction. This is simple to implement, and provides adequate information for the calculation of latency between each endpoint and a passive observation point on both the upstream and downstream path between them, e.g., at a network border or home router (customer premises equipment, CPE) gateway, where these measurements are usually deployed.

Given two endpoints a and b, an observation point c can measure the time interval between seeing a given PSN on a packet sent by a and an equal or greater PSE on a packet sent by b to get a latency estimate on the path $c \leftrightarrow b$. By reversing the measurement, it can likewise estimate $c \leftrightarrow a$, and $a \leftrightarrow b = c \leftrightarrow b + c \leftrightarrow a$.

1.3.2 Loss Estimation

Loss estimation on the path $a \rightarrow c$ for packets in direction $a \rightarrow b$ is supported by our PSN/PSE latency measurement facility by counting gaps in the sequence of PSNs sent by a. However, this gives no visibility into loss on the path $c \rightarrow b$.

To provide a solution that is transport protocol independent, we cannot simply expose transport protocol internals: requiring observation points and measurement analysis to infer behavior of endpoints based upon an identification of the transport protocol is precisely the current situation we want to avoid. Therefore, we must rely on the receiver to expose information about the loss and congestion experienced explicitly. We take design inspiration from the Congestion Exposure (ConEx) protocol [9], and propose a sender-to-path signal for periodically exposing counts of detected packet loss and congestion signals to provide a delayed but accurate loss metric to the path.

Each sender emits a periodic sender-to-path signal containing a two-tuple $\{l, m\}$ where l is a cumulative count of the number of detected losses l since the beginning of the flow and m is a cumulative count of the number of detected ECN [30] marks since the beginning of the flow. A measurement point can receive and analyze the series of signals to derive various loss statistics: $l_{t2} - l_{t1}$, for example, gives the number of losses detected by the sender in the time interval $(t1 - owd(a \rightarrow c), t2 - owd(a \rightarrow c))$, where a first-order estimate of the one-way delay from the sender $owd(a \rightarrow c)$ can be derived from RTT measurements[1]. Comparing this to the observed packet transmission rate during the same interval, as well as to the upstream loss information available from the PSN series, allows a measurement point to estimate upstream as well as downstream loss rates. The addition of ECN marking information, similarly, together with an analysis of observed Congestion Experienced (CE) codepoints at the IP layer, allows the estimation of upstream and downstream congestion for ECN-enabled flows.

[1]Path asymmetry, of course, reduces the accuracy of this estimate. We leave accurate OWD estimation through timestamp exposure and synchronization as a topic for future work.

European Commission | Horizon 2020 European Union funding for Research & Innovation

2 Path Layer UDP Substrate (PLUS) Specification

This section specifies PLUS. PLUS is realized as a shim header between the UDP header and an encrypted transport header. The PLUS header explicitly exposes information intended by the sender for use by devices along the path independent of the transport protocol, hiding everything else with transport encryption.

PLUS defines two headers, a Basic Header carrying information for state management, basic measurability, and simple packet treatment; and an Extended Header carrying a path communication field, described in Section 2.2. The Basic Header supports sender-to-path signaling for state maintenance and basic measurement. The Extended Header supports both sender-to-path and path-to-receiver signaling, for carrying information for which there is no room in the Basic Header, as well as for accumulated path information as in section 4.3 of [2].

2.1 Basic Header

The format of the PLUS header, together with the UDP header, is shown in Figure 3. Since PLUS is designed to be used for UDP-encapsulated, encrypted transport protocols, overlying transports are presumed to provide encryption and integrity protection for their own headers. For the sake of efficiency, it is also assumed that this integrity protection can be extended to the bits in the PLUS Basic Header.

The magic field is a 28-bit number that identifies this packet as carrying a PLUS header. This magic number can be used by PLUS-aware devices on path to distinguish PLUS packets from non-PLUS packets, since PLUS cannot be identified by UDP port number. It is chosen to avoid collision with possible values of the first four bytes of any packet in widely deployed protocols on UDP. This both minimizes the chance of accidental classification of a non-PLUS protocol as PLUS, and makes it difficult to cause spurious PLUS packets to be generated by reflection or amplification.

The next four bits provide flags: two per-packet Quality of Service (QoS) signals, a stop signal, and an extended header bit. The (L)oLa flag, when set, indicates that the packet is latency sensitive and prefers drop to delay. The (R)oI flag, when set, indicates that the packet is not sensitive to reordering and thus does not need to be given the same treatment or routing as other packets of the same flow. These two signals address many QoS challenges operators face in current networks but cannot be reliably provided end-to-end by existing mechanisms, such as the use of the Differentiated Services Codepoint (DSCP) field in the IP header [51]. The (S)top flag indicates a stop or stop confirmation signal when set. When the E(X)tended Header bit is set, the Extended Header is present; see Section 2.2.

The Connection/Association Token (CAT) is a 64-bit token identifying this association. The CAT is chosen randomly by the connection initiator, and allows both multiplexing of flows over a 5-tuple as well as fast rebinding: a PLUS packet sharing one endpoint (source address/port pair, or destination address/port pair) and the CAT with an existing flow is taken to belong to that flow, since the other endpoint identifier might change due to a mobility event or address translation change. The CAT also can be used as an identifier to maintain state for stateful in-network processing.

The Packet Serial Number (PSN) is a 32-bit serial number for this packet. The initial PSN for

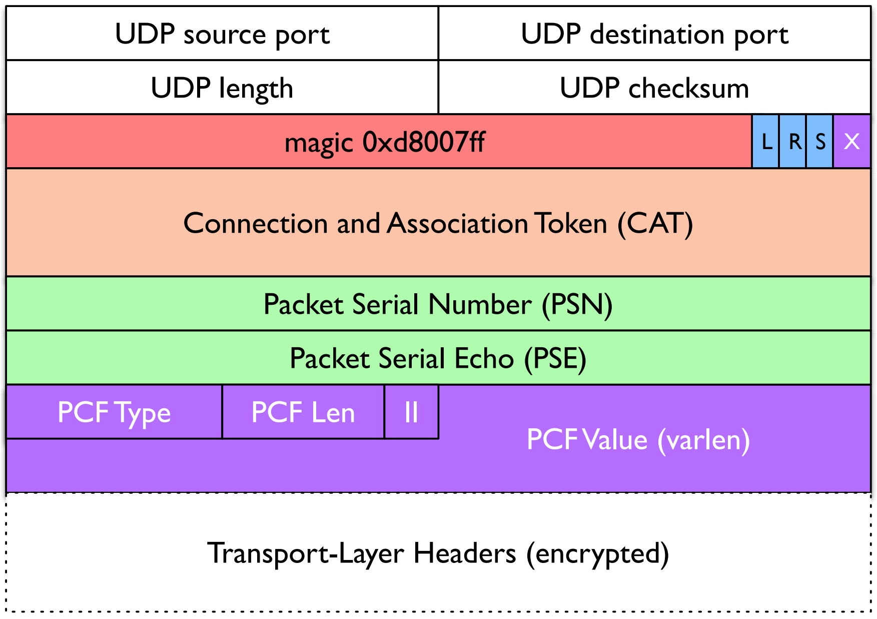

Figure 3: The PLUS header format. The Basic Header (X=0) omits the PCF fields; the Extended Header (X=1) has all fields.

each direction in a flow is chosen randomly. Each subsequently sent packet in a flow increments the PSN by one, wrapping around. Respectively, the Packet Serial Echo (PSE) is the most recent PSN seen by the sender in the opposite direction before this packet was sent. If no packet has yet been seen by the sender (i.e., the packet is the first packet in a bidirectional flow), the sender sets the PSE to zero. The CAT and PSE together serve as an indication that a packet was actually seen by the remote endpoint, and used for confirmation and stop-confirmation signals to protect against off-path attacks and as an indication for on-path state management. PSN and PSE together can also be used for in-network latency measurement as described in Section 1.3.

When a sender has a packet ready to send using PLUS, it determines the values in the Basic Header as follows:

1. The magic number is set to the constant 0xd8007ff.

2. If the sender is the flow initiator, and the packet is the first packet in the flow, the sender selects a cryptographically random 64-bit number for the CAT. When multiplexing, it must ensure the CAT is not already in use for the 5-tuple. Otherwise, the sender uses the CAT associated with the flow.

3. If the packet is the first packet in the flow in this direction, the sender selects a cryptographically random 32-bit number for the PSN. Otherwise, the sender adds one to the PSN on the last packet it sent in this flow, and uses that value for the PSN. If the last PSN is 0xffffffff, it wraps around, setting the PSN to 0x00000001. A PSN of 0x00000000 is never sent.

4. If the packet is the first packet in the flow in this direction, the sender sets the PSE to

Horizon 2020
European Union funding
for Research & Innovation

0x00000000. Otherwise it sets the PSE to the PSN of the last packet seen in the opposite direction.

5. If the overlying transport determines that this packet is loss- insensitive but latency-sensitive, the sender sets the L flag.

6. If the overlying transport determines that this packet may be freely reordered, the sender sets the R flag.

7. If the overlying transport determines that the connection is shutting down, and no further packets will be sent in this direction other than packets part of this shutdown, the sender sets the S flag; see Section 2.3.2 for details.

When a receiver receives a packet containing a PLUS Basic Header, it processes the values in the Basic Header as follows:

1. It verifies that the magic number is the constant 0xd800fff. If the receiver is expecting a PLUS packet, and it does not see this value, it drops the packet without further processing.

2. It verifies the integrity of the information in the PLUS Basic Header, using information carried in the overlying transport. Packets failing integrity checks should be dropped, but may be further analyzed by the receiver to determine the likely cause of verification failure; reaction to the failure is transport and implementation specific.

3. It stores the PSN to be sent as the PSE on the the next packet it sends in the opposite direction.

2.2 Extended Header

When the Extended Header bit is 1, the PLUS Extended Header is present. This header adds a Path Communication Field (PCF) to the Basic Header, as shown in Figure 3. The Extended Header has an 8-bit PCF type field, a 6-bit PCF length field, a 2-bit Integrity Indicator, and a variable-length PCF value field, protected by the overlying transport encryption.

The PCF Type field defines the structure and semantics of the PCF value. The PCF Length field defines the length of the PCF value field in bytes, from 0 to 63 bytes. The PCF Integrity Indicator (II) is used to implement path-to-receiver as well as sender-to-path signaling, implemented by treating a portion of the field as if all its bits are zero for purposes of integrity protection and integrity verification. This protects the type and length of path-to-receiver signals, but allows part or all of the PCF Value to be writable by devices on path.

If the Integrity Indicator is 00, the PCF value is not integrity protected; if it is 11, the PCF value is integrity protected in its entirety. The other two values provide for partial integrity protection: 01 indicates that the the first quarter of the PCF value is protected; 10 indicates that the first half of the PCF value is protected. The integrity protected range is always rounded up to the nearest byte.

European Commission | Horizon 2020 European Union funding for Research & Innovation

2.3 Extended Header Types

This section defines a set of initial Extended Header types to illustrate the flexibility provided by the Extended Header. Other PCF types are reserved for future use.

The PCF is used to expose loss and congestion markings to the path. PCF Type 1 indicates the Loss and Congestion Exposure field, a field which is 2, 4, 8, or 16 bytes long, and contains two 1, 2, 4, or 8 byte unsigned integers in network byte order. The first is a total count of packets detected as lost by the sender since the start of the connection. The second is a count of the total number of congestion markings (ECN CE codepoints) received by the sender since the start of the connection. Its integrity indicator is always 11, since the information cannot be modified by on-path devices. PCF Type 1 packets can be emitted as often as once per RTT as estimated by the sender; lower rates result in less overhead, but lower fidelity measurements by devices on path.

The PCF is also used for path accumulation; to be useful this requires deployment of future PLUS-aware middleboxes. PCF Type 2 carries a two byte value reflecting the Path Maximum Transmission Unit (PMTU), initialized to the MTU of the link on which the packet is initially sent. When a PLUS-aware middlebox forwards a packet with this PCF, it updates the value with the minimum of the value present and the MTU of the link over which it will forward the packet. The integrity indicator for PCF Type 2 is always 00, since the path can modify the entire value.

PCF Type 3 carries an 8 byte field for path tracing, similar to the Path Changes mechanism in section 4.3 of [2]. The sender chooses a random 8-byte value for each flow, and initializes the PCF value field with this value for each packet with PCF Type 3 in the flow. Each on-path PLUS-aware device forwards the packet fills the result of XORing the received value with an 8-byte device identifier. The same randomly chosen value must be used for all path trace accumulator signals. Packets traversing the same set of PLUS-aware forwarding devices are therefore received with the same accumulated value, and changes to the set of devices on path can be detected by the receiver. The integrity indicator for PCF Type 3 is always 00, since the path can modify the entire value.

PCF Type 255 is reserved for extensions to the PCF mechanism. PCF Types 253 and 254 are reserved for experimentation.

Devices on path must recognize that they are requested to participate in signaling by understanding the Extended Header and the semantics of each type. However, there is no guarantee that they will participate or reply honestly. Therefore all signals are only advisory, guiding the transport or application receiving them to take appropriate actions. In general, endpoints still need to implement mechanisms to handle incorrect or incomplete exposed information, as today when default values are assumed (e.g., the accumulated MTU should not be directly used to set the segment size, but rather as input to a packetization layer path MTU discovery process [24]).

2.4 Encrypted Feedback and the Transport layer API

Path-to-receiver signals transmit information requested from the path by the sender to the receiver; however, the sender often also needs the information that was requested. The transport protocol running atop PLUS must therefore provide a feedback channel for the full PLUS Extended Header on any packet received containing a PCF value where the integrity indicator is not 11. Returning the entire header allows the sender to associate the value fed back with the original packet sent; feedback of the entire header depending only on the II value allows receivers to feed back PCF values they do not understand. We assume the overlying transport provides end-to-end encryption, and that signals fed back do not need to be exposed to the reverse path. The feedback channel interface must be designed on a per-transport basis.

Other mechanisms supported by PLUS as presently defined does not require much application interaction: CAT, PSN/PSE, and the loss/congestion PCF all expose either information available within PLUS, or that is readily available from the internals of the overlying transport. Similarly, MTU accumulation is only useful at the transport layer, and path change detection is more suited to diagnostic tools in the spirit of traceroute than online use by an application. Therefore, the API presented by the transport does not need to change when a given transport is running over PLUS. However, there can be further benefits if an application is aware of PLUS. The decision to enable or disable PLUS on a given flow or node may be driven not only by the needs of the transport protocol but also the application, or even the user, following the principles of transparency and endpoint control. We discuss how to add PLUS-specific controls to our flexible transport layer API in section 5.1.

2.5 On-Path State Maintenance

PLUS headers are added to packets by a sender and can be inspected by on-path devices such as middleboxes. The first packet with a PLUS header forwarded by an on-path device for a given 5-tuple plus CAT the $a \rightarrow b$ direction is used to establish initial state. When a packet with a PLUS header with a reversed 5-tuple and identical CAT in the $b \rightarrow a$ direction is observed by the PLUS-aware on-path device this can be see as a confirmation to communicate to the initiation endpoint. However, before final state is established, with potentially longer time-outs or additional access rights, the on-path device should waits to see a packet with a PSE in the $a \rightarrow b$ direction equal to the PSN on the first reverse packet as a proof of return-routability.

A PLUS-aware on-path device observing a packet with the S flag set, for a flow that is has state for, can prepare to remove that state but should wait until it sees a packet in the opposite direction with the S flag set and the PSE set to exactly the PSN as in the PLUS packet with the S flag set that was previously observed in the opposite direction. Otherwise, even though the S flag is integrity-protected end-to-end, a packet with the S flag set could be forged by one on-path device to drive the flow state to be removed on all downstream devices which could lead to packet loss, additional latency, connectivity breakage, or full blocking of any further transmission between the endpoint depending on the stateful network function that is provided by the network device that is under attack. Waiting for the confirmation signal from the other end makes this attack much harder, as it would require coordination between attackers on both sides of a given on-path device in order to forge a confirmation of the stop signal. Further the device should forward packets in both directions for flows before removing state completely for about an additional RTT, or a time that is large enough to safely cover most common RTTs, as packets might still be inflight, due to e.g. reordering, after the packets with the S bit set in both directions have been observed.

One end of a PLUS association may change its address while maintaining on-path state; e.g. due to a NAT rebinding. A PLUS-aware on-path device that forwards a packet where one of the endpoint identifiers (address and port) and the CAT, but not the other endpoint identifier,

European Commission | Horizon 2020 European Union funding for Research & Innovation

match existing flow state, treats that packet as belonging to the existing flow, and updates the endpoint identifier.

However, if state is lost or removed prematurely, the information in the Basic Header on each packet provide enough information to reestablish state quickly even in the middle of a flow, enabling fast recovery even in the case of such a coordinated attack.

Horizon 2020
European Union funding
for Research & Innovation

3 Applying Middlebox Cooperation Mechanisms to New and Existing Protocols

The PLUS header, as described in the previous chapter, provides a clean architectural approach for middlebox cooperation, separating information that are intended to be consumed by the path (the wire image [19]) from end-to-end signaling. This approach is necessary for transport protocol evolution. The PLUS header thereby effectively introduces a new path layer between the transport and network layers that provides signaling for stateful network devices. While such in-network functions were not intended in the original Internet architecture, we can no longer deny or ignore that these functions exis, and have even proven highly useful for network management as well as in-network security, especially as a first wall of defense again DDoS attacks.

However, deploying such a new protocol requires an alignment of incentives for all involved parties: users, developers of endhost network stacks and applications, as well as developers and deployers of network nodes. Moreover, a standardized protocol specification is necessary to ensure interoperability.

Attempts to standardize middlebox cooperation approaches have been criticized as potentially dangerous, as any additional information that is exposed in the clear can bear an additional risk in the face of pervasive passive surveillance. The IETF has declined, for now, to begin work on a generic, protocol-independent explicit middlebox cooperation mechanism, due in large part to this criticism.

However, given the need to maintain existing in-network functions, such as firewalls, NATs, or load-balancing, while increasing deployment of encryption and thereby limiting the information available to the path, specific middlebox cooperation mechanism are still under discussion and standardization, embedded into existing protocols or new protocols currently in development. While the project has consensus that the best way to solve this problem is a protocol-independent approach for middlebox cooperation, we currently are active in standardization by applying the mechanisms developed for PLUS to new and existing protocols. This section provides more details about on-going activities and the chosen approaches.

Moreover, the project is performing further experiments based on PLUS as proposed. These are intended as a proof of concept showing that the proposed solution and approach for specific middlebox cooperation use cases are viable. However, we do not expect the deployment of PLUS as specified in this deliverable without a more radical trend for redesigning protocol stacks in the Internet protocol standardization community.

In our experiment, we show how these concepts can be applied to other protocols, especially the QUIC protocol which is a new, encrypted transport that is currently under standardization in the IETF with an expected large impact on the near-future Internet traffic.

In this chapter we first focus on the most basic signaling requirement for stateful in-network devices, flow state management. In the next section we detail a protocol-independent state machine to be implemented by network devices that aim to support state management for different protocols, and show how the input signals that drive the state machine are applicable to different protocols with PLUS, TCP, and QUIC as examples.

Further, we detail how concepts derived from the information provided by the Basic PLUS header such as the packet number and packet number echo, the connection ID, and the

(L)atency flag can be applied and adopted for latency measurements, packet flow binding, and latency support in other protocols, such as QUIC, DTLS, or DiffServ.

Finally, we discuss how mechanisms provided by the PLUS Extended Header PCF Types 1 and 2, for congestion exposure and path MTU discovery, can be utilized by a new transport protocol like QUIC or a new UDP signal mechanism that enables option space for UDP-based protocols.

3.1 Protocol-independent State Management

As most complex in-network functions are stateful, the most basic signaling requirement for on-path network functions is to support state maintenance, through on-path recognition of session establishment and disestablishment. In the current, TCP-dominated Internet, this is provided by passive observation of the TCP three-way handshake and the FIN and/or RST flags on connection shutdown. These are visible on the path between endpoints, and therefore are often used by those network devices for state management by devices such as NATs and firewalls.

The most important signals derived from these observations concern flow lifetime and association of the two directions of a bi-directional flow, associating a uniflow with its reverse counterpart. Firewalls and other stateful network elements can use this association to assume endpoint consent to communicate. A stop signal that indicates the last packet of a flow can also assist state management, and allows for a longer timeout for active flows than presently possible for UDP [14], reducing the rate at which keep-alive traffic must be sent to avoid loss of on-path state. Stateful network devices can apply stricter policies for a flow which does not provide these signals, such as shorter timeouts to remove flow state, or rules to drop unknown traffic.

Figure 4 shows a transport-independent state machine that details the basic operation of a stateful on-path device. This abstract state machine is also described an IETF Internet-Draft in [21] providing input for the QUIC standardization process on manageability considerations [20].

The state machine is designed to replace TCP state-tracking for firewalls and NAT devices. When encrypted transport protocols encapsulated in UDP adopt a set of signals and a wire format for those signals to drive this state machine, these middleboxes could continue using TCP-like logic to handle those UDP flows. Recognizing the wire format used by those signals would allow these middleboxes to distinguish “UDP with an encrypted transport” from undifferentiated UDP, and to treat the former case more like TCP, providing longer timeouts for established flows, as well as stateful defense against spoofed or reflected garbage traffic.

The state machine has the following properties:

1. A device on path that can see traffic in both directions knows each side of an association wishes that association to continue. This allows firewalls to delegate policy decisions about accepting or continuing an association to the servers they protect.

2. A device on path that can see traffic in both directions knows that each device can receive traffic at the source address it provides. This allows firewalls to protect against trivially spoofed packets, implemented by associating and associated states.

3. Both endpoints confirm the desire to end communication, providing resistance against

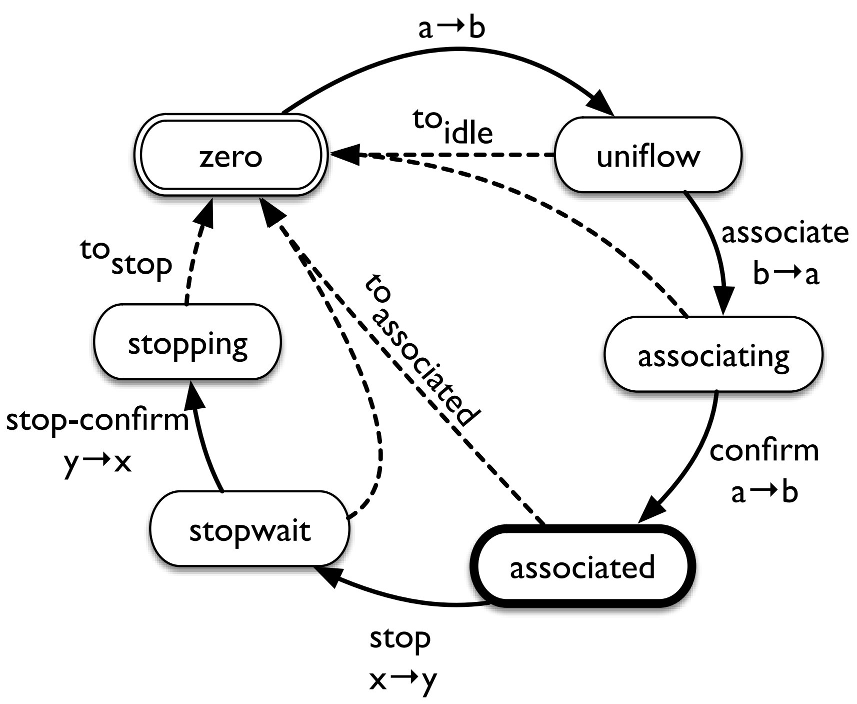

Figure 4: A transport-independent path layer state machine.

early state expiry attacks (e.g. TCP RST injection) by on-path devices, implemented by the stopwait and stopping states.

The first two properties hold with current firewalls and network address translation devices observing the flags and sequence/acknowledgment numbers exposed by TCP.

The state machine has on six states, three configurable timeouts, and a set of signals.The states are defined as follows:

1. **zero:** there is no state for a given flow at the device
2. **uniflow:** at least one packet has been seen in one direction
3. **associating:** at least one packet has been seen in one direction, and an indication that the receiving endpoint wishes to continue the association has been seen in the other direction.
4. **associated:** a flow in associating state has further demonstrated that the initial sender can receive packets at its given source address.
5. **stop-wait:** one side of a connection has sent an explicit stop signal, waiting for confirmation
6. **stopping:** stop signal confirmed, association is stopping.

The three timeouts are defined as follows:

European Commission | Horizon 2020 European Union funding for Research & Innovation

1. TO_IDLE, the unidirectional idle timeout, can be considered equivalent to the idle timeout for transport protocols where the device has no information about session start and end (e.g. most UDP protocols).

2. TO_ASSOCIATED, the bidirectional idle timeout, can be considered equivalent to the timeout for transport protocols where the device has information about session start and end (e.g. TCP).

3. TO_STOP is the teardown timeout: how long the device will account additional packets to a flow after confirming a close signal, ensuring retransmitted and/or reordered close signal don't lead to the spurious creation of new flow state.

Selection of timeouts is a configuration and implementation detail, but generally

$\text{TO\_STOP} \leq \text{TO\_IDLE} < \text{TO\_ASSOCIATED}$;

see [14] for an analysis of the magnitudes of these timeouts in presently deployed gateway devices.

The state machine is driven by four signals: a *new flow* signal, the first packet observed in a flow in the zero state; an *association* signal, allowing a device to verify that an endpoint wishes a bidirectional communication to be established or to continue; a *confirmation* signal, allowing a device to confirm that the initiator of a flow is reachable at its purported source address; and a *stop* signal, noting that an endpoint wishes to stop a bidirectional communication. There are a different ways to implement these signals, as further explained in the next subsection for PLUS, TCP, QUIC as example protocols.

For stateful in-network functions, every packet received by a device keeping per-flow state must associate that packet with a flow in order to extract common properties to identify the flow it is associated with. In general, the set of properties used for flow identification on presently deployed devices includes the source and destination IP address, the source and destination transport layer port number, the transport protocol number. The differentiated services field [RFC2474] may also be included in the set of properties defining a flow, since it may indicate different forwarding treatment. However, other protocols may use additional bits in their own headers for flow identification.

When a device receives a packet associated with a flow it has no state for, it create a flow entry, moves that flow from the zero state into the uniflow state and starts a timer TO_IDLE. It resets this timer for any additional packet it forwards in the same direction as long as the flow remains in the uniflow state. When timer TO_IDLE expires on a flow in the uniflow state, the device drops state for the flow and performs any processing associated with doing so: tearing down NAT bindings, stopping associated firewall pinholes, exporting flow information, and so on. The device may also drop state on a stop signal, if observed.

We refer to the zero and uniflow states as "uniflow states", as they are relevant both for truly unidirectional flows, as well as in situations where an on-path device can see only one side of a communication. We refer to the remaining four states as "biflow states", as they are only applicable to true bidirectional flows, where the on-path device can see both sides of the communication.

Devices that only see one side of a communication, e.g. if they are placed in a portion of a network with asymmetric routing, use only the zero and uniflow states. In addition, true uniflows - protocols which are solely unidirectional (e.g. some applications over UDP) - will also use only the uniflow-only states. In either case, current devices generally do not associate much state with observed uniflows, and a short idle timeout is generally sufficient to expire this state.

A uniflow transitions to the associating state when the device observes an association signal, and further to the associated state when the device observes a subsequent confirmation signal from the other communication endpoint. If the flow has not transitioned from the associating to the associated state after TO_IDLE, the device drops state for the flow.

After transitioning to the associated state, the device starts a timer TO_ASSOCIATED. It resets this timer for any packet it forwards in either direction. The associated state represents a fully established bidirectional communication. When timer TO_ASSOCIATED expires, the device assumes that the flow has shut down without signaling as such, and drops state for the flow, performing any associated processing.

The transport-independent state machine uses bidirectional stop signaling to tear down state. This requires a stop signal to be observed in one direction, and a stop confirmation signal to be observed in the other, to complete tearing down an association. A stop signal indicates that a flow is ending, whether normally or abnormally, and that state associated with the flow should be torn down. A device on path moves the flow from uniflow state to zero, or from associated state to stopwait state, to wait for a confirmation signal in the other direction. While in stopwait state, state will be maintained until a timer set to TO_ASSOCIATED expires, with any packet forwarded in either direction reseting the timer.

The stop confirmation indicate that the endpoint receiving the stop signal confirms that the stop signal is valid. The stop confirmation signal contains some assurance that the far endpoint has seen the stop signal. When a stop confirmation signal is observed in the opposite direction from the stop signal, the on-path devices moves the flow from stopwait state to stopping state. When a flow enters the stopping state, it starts a timer TO_STOP. The flow will then remain in stopping state until the timer set to TO_STOP has expired, after which state for the flow will be dropped. The stopping timeout TO_STOP is intended to ensure that any packets reordered in delivery are accounted to the flow before state for it is dropped.

Stop signals, as association signals, could be forged by a single on-path device. However, unless a stop confirmation signal that can be associated with the stop signal is observed in the other direction, the flow remains in stop-wait state, during which state is maintained and packets continue to be forwarded in both directions. So this attack is of limited utility; an attacker wishing to inject state teardown would need to control at least one on-path device on each side of a target device to spoof both stop and corresponding stop confirmation signals.

The proposed state machine is concerned only with states and transitions common to transport- and function- independent state maintenance. Devices may augment the transitions in this state diagram depending on their function. For example, a firewall that decides based on some information beyond the signals used by this state machine to shut down a flow may transition it directly to a blacklist state on shutdown. Or, a firewall may fail to forward additional packets in the uniflow state until an association signal is observed.

Although all of these signals are required to drive the state machine described by this document, note that association/confirmation and bidirectional stop signaling have separate utility. A transport protocol may expose the end of a flow without any proof of association or confirmation of return routability of the initiator. Alternately, the transport protocol could rely on short timeouts to clean up stale state on path, while exposing continuous association and confirmation signals to quickly reestablish state.

3.1.1 Implementation Considerations for Association and Confirmation Signaling

An association signal indicates that the endpoint that received the first packet seen by the device has indeed seen that packet, and is interested in continuing conversation with the sending endpoint. This signal is roughly an in-band analogue to consent signaling in Interactive Connectivity Establishment (ICE) [29], that is carried to every device along the path.

A confirmation signal indicates that the endpoint that sent the first packet seen by the device is reachable at its purported source address, and is necessary to prevent spoofed or reflected packets from driving the state machine into the associated state. It is roughly equivalent to the final ACK in the TCP three-way handshake.

These two signals are related to each other, in that association requires the receiving endpoint of the first packet to prove it has seen that packet (or a subsequent packet), and to acknowledge it wants to continue the association; while confirmation requires the sending endpoint to prove it has seen the association token.

Transport-independent, path-verifiable association and confirmation signaling can be implemented using three values carried in the packet headers: an association token, a confirmation nonce, and an echo token.

The association token is a cryptographically random value generated by the endpoint initiating a connection, and is carried on packets in the uniflow state. When a receiving endpoint wishes to send an association signal, it generates an echo token from the association token using a well-known, defined function (e.g. a truncated SHA-256 hash), and generates a cryptographically random confirmation nonce. The initiating endpoint sends a confirmation signal on the next packet it sends after receiving the confirmation nonce, by applying a function to the echo token and the confirmation nonce, and sending the result as a new association token.

Devices on path verify that the echo token corresponds to a previously seen association token to recognize an association signal, and recognize that an association token corresponds to a previously seen echo token and confirmation nonce to recognize an association signal.

If the association token and confirmation nonce are predictable, off- path devices can spoof association and confirmation signals. In choosing the number of bits for an association token, there is a tradeoff between per-packet overhead and state overhead at on-path devices, and assurance that an association token is hard to guess. This tradeoff must be evaluated at protocol design time.

There are a few considerations in choosing a function (or functions) to generate the echo token from the association token, to verify an echo token given an association token, and to derive a next association token from the echo token and confirmation nonce. The functions could be extremely simple (e.g., identity for the echo token and addition for the nonce) for ease of implementation even in extremely constrained environments. Using one-way functions (e.g., truncated SHA-256 hash to derive echo token from association token; XOR followed by truncated SHA-256 hash to derive association token from echo token and confirmation nonce) requires slightly more work from on-path devices, but the primitives will be available at any endpoint using an encrypted transport protocol. In any case, a concrete implementation of association and confirmation signaling must choose a set of functions, or mechanism for unambiguously choosing one, at both endpoints as well as along the path.

There are two possible points in the design space here: these signals could be continually exposed throughout the flow, or could be exposed only on the first few packets of a connection (those corresponding to the cryptographic and/or transport state handshakes in the overlying protocols).

In the former case, an on-path device could re-establish state in the middle of a flow; e.g. due to a reboot of the device, due to a NAT association change without the endpoints' knowledge, or due to idle periods longer than the TO_ESTABLISHED timeout value. The on-path device would receive no special information about which packets were associated with the start of association. In this case, the series of exposed association tokens, echo tokens, and confirmation nonces can also be observed to derive a running round-trip time estimate for the flow.

In the latter case, an on-path device would need to observe the start of the flow to establish state, and would be able to distinguish connection-start packets from other packets.

3.1.2 Signal mapping for PLUS

The PLUS Basic Header provides all signaling required for the proposed state machine. A PLUS-aware on-path device can associate a packet to a flow based on a 5-tuple and CAT. If a packet is observed that does not have a flow in uniflow state yet, that packet is assumed to be the first packet of a flow and flow state is created (in uniflow state).

A PLUS-aware on-path device forwarding a packet with a PLUS Basic Header with a matching 5-tuple and CAT as a flow in the uniflow state, but in the opposite direction, moves that flow to the associating state. It stores the PSN of the packet that caused this transition, and waits for a packet in the a other direction containing a PSE as a confirmation siganl to transition the flow to the associated state. In associated state, every packet with a PLUS Basic Header identified by the 5-tuple and CAT resets the timer.

The S flag in the PLUS Basic Header provides the stop signal. When the S flag is observed and the flow moves to the stopwait state, the network devices also needs to store the PSN on the packet causing the transition. When it sees a packet in the opposite direction with the S flag set and the PSE set to exactly the stored PSN, it transitions the flow to closing state.

3.1.3 Signal mapping for TCP

A mapping of TCP flags to transitions in to the state machine shows how devices currently using a model of the TCP state machine can be converted to use this state machine.

TCP provides start-of-flow association only. A packet with the SYN and ACK flags set in the absence of the FIN or RST flags, and an in-window acknowledgment number, is synonymous with the association signal. A packet with the ACK flag set in the absence of the FIN or RST flags after an initial SYN, and an in-window acknowledgment number, is synonymous with the confirmation signal.

For a typical TCP flow the state machine is performing the flowing steps:

1. The initial SYN places the flow into uniflow state.
2. The SYN-ACK sent in reply acts as a association signal and places the flow into associating state.

European Commission | Horizon 2020 European Union funding for Research & Innovation

3. The ACK sent in reply acts as a confirmation signal and places the flow into associated state.
4. The final FIN is a stop signal.
5. And the ACK of the final FIN is a stop confirmation signal, moving the flow into stopping state.

Note that abnormally closed flows (with RST) do not provide stop confirmation, and are therefore not provided for by this state machine. Due to TCP's support for half-closed flows, additional state modeling is necessary to extract a stop signal from the final FIN.

Note also that the association and stop signals derived from the TCP header are not integrity protected, and association and confirmation signals based on in-window ACK are not particularly resistant to off-path attacks. The state machine is therefore more susceptible to manipulation when used with vanilla TCP as when with a transport protocol providing full integrity protection for its headers end-to-end, such as PLUS or QUIC.

3.1.4 Signal mapping for QUIC

QUIC [16] is a new, encrypted transport protocol that is currently under standardization in the IETF. As such it is a moving target; this discussion refers to the most recent published version of the draft specifications as of this writing.

QUIC's handshake is visible to on-path devices, as it begins with an unencrypted TLS handshake that exposes a 64-bit connection ID, which can serve as an association and echo token. The function of the confirmation nonce is not fully exposed to the path at this point, but could be implemented by echoing the random initial packet number.

However, there is no stop signal, and the emerging consensus of the working group is that a stop signal is unnecessary. Therefore, QUIC state maintenance must rely on timeouts alone to drop state. When implemented for QUIC, the state machine described here should use a TO_ASSOCIATED equal to or nearly equal to TO_IDLE; the close transitions will never occur.

3.2 Latency Measurements

The PLUS Basic Header's PSN and PSE provide a TCP-equivalent explicit signal for measuring upstream and downstream round-trip time, as detailed in section 1.3. In case of QUIC the wire image already includes a packet number which works like the PLUS PSN. The project proposed to include a packet number echo to restore the ability to use this information for latency measurements. However, this proposal was rejected in part for being too high-overhead. In addition, the working group is currently discussing an approach to also encrypt the packet number, which would break the semantics of a PSE. We therefore turn our attention to low-overhead explicit support for latency measurement.

It is possible to provide one RTT sample per RTT to on-path observers using a *s*pin bit, as we have proposed to add to the QUIC transport protocol [47]. This mechanism works by adding a single bit to the header (in the case of QUIC as currently proposed [16], the short header only that is used after the handshake) containing the sender's *s*pin value. Each endpoint maintains its spin value as follows:

- The server initializes its spin value to 0. When it receives a packet from the client, if that packet increments the highest packet number seen by the server from the client, it sets the spin value to the spin bit in the received packet.

- The client initializes its spin value to 0. When it receives a packet from the server, if the packet increments the highest packet number seen by the client from the server, it sets the spin value to the opposite of the spin bit in the received packet.

Note that the requirement to only update the spin value on packets which increment the max packet number seen ensures that the spin value will not oscillate spuriously in the presence of packet reordering.

When a client or server sends a packet, it sets the spin bit to its current spin value. In this way, when a flow is sending at full rate (i.e., not application limited), the spin bit in each direction changes value once per round-trip time (RTT). An on-path observer can observe the time difference between edges in the spin bit signal in a single direction to measure one sample of end-to-end RTT. Note that this measurement, as with passive RTT measurement for TCP, includes any transport protocol delay (e.g., delayed sending of acknowledgements) and/or application layer delay (e.g., waiting for a request to complete). It therefore provides devices on path a good instantaneous estimate of the RTT as experienced by the application. A simple linear smoothing or moving minimum filter can be applied to the stream of RTT information to get a more stable estimate.

An on-path observer that can see traffic in both directions (from client to server and from server to client) can also use the spin bit to measure “upstream” and “downstream” component RTT; i.e, the component of the end-to-end RTT attributable to the paths between the observer and the server and the observer and the client, respectively. It does this by measuring the delay between a spin edge observed in the upstream direction and the corresponding edge observed in the downstream direction, and vice versa.

The proposed spin bit is currently under discussion in the QUIC working group for addition to the protocol. See our proposal [47] for more details.

3.3 Packet Flow Binding using Connection ID

The PLUS header, as well as the currently proposed QUIC wire image, expose, in addition to the source/destination IP address/port and protocol type 5-tuple, a connection ID that can be used to group packets into a flow even if parts of the 5-tuple changes, e.g. because of mobility or NAT rebindings.

The DTLS protocol [31] does not specify the use of a connection ID, therefore the end-to-end security association is bound to the 5-tuple only, and relies on its stability. This assumption is acceptable in cases where the traffic pattern is dense enough to keep the NAT binding alive (Linux default timeout for UDP traffic is only 3 minutes, as opposed to TCP’s 5 days), maybe at the cost of creating synthetic keep-alive traffic, for example, using the TLS heartbeat extension [49]. This strategy is not particularly robust because choosing the right keep-alive clocking depends on many external factors which might change after the endpoints are deployed. Obviously, the instability of the 5-tuple can always be worked around by re-establishing the security association, but key agreement is a compute-intensive operation.

Horizon 2020
European Union funding
for Research & Innovation

While most endpoints can bear the costs of 5-tuple ephemerality (either via keep-alive or re-handshaking), the class of IoT devices that use a Constrained Application protocol (CoAP) [35] over DTLS [48] stack, are resource constrained, and battery operated cannot. For them, waking up from a sleep cycle only to find the security session can't be de-referenced anymore on server side because the 5-tuple is gone is a complete disaster in terms of battery management. It should also be noted that the state that is dropped on the NAT box immediately becomes dead state in the server, consuming precious resources on an endpoint that could also be memory constrained. So, a connection ID is an unavoidable necessity for these use cases and [32] is a recent proposal by the project to address that.

Further, in typical IoT deployments, where a certain (potentially very high) number of sleepy nodes wakes up every t seconds, send a small update to a collector node, and then go back to sleep, the connection ID is also a very valuable tool for diagnostic. If t is higher than the UDP NAT timeout, each time the node sends the update, the 5-tuple is different and the connection-id is the only stable identifier that can be used to detect anomalies in the flows' aggregate.

3.4 Low Latency Support

Based on the observable trend that the number of latency sensitive application that are not well supported by todays network management approaches is strongly increasing, the PLUS Basic header provides the L flags to indicate their latency needs to latency-aware networks.

Similar to the L flags in PLUS, a new PHB (Per-Hop Behavior) group called Loss-Latency Tradeoff (LoLa) [51] has been proposed as an extension to the DiffServ [28] marking scheme. The LoLa group, similar proposed for as the PLUS L flag, provides a radical simplification of the DiffServ model compared to traditional marking schemes. Traffic belongs to one of two classes: loss-sensitive (Lo) or latency-sensitive (La). Applications explicitly mark their flows as belonging to one of the classes using the respective DiffServ codepoint (DSCP), and thereby signal their intended service treatment to the network. The network should treat these two classes differently, for example by using two queues with different configuration: one short queue (or one that uses an AQM optimized for low queueing delay) which might induce higher loss rates, and one larger queue that is optimized for high throughput and low loss. This trade-off forms the basis of a cooperative game in which the participants have no incentive to lie. In fact, users cannot get an advantage in the treatment of their flows by cheating about their real nature because they would have to suffer self-inflicted high delay or high loss as a consequence of the incorrect marking.

Another nice property of LoLa is that it is mobile network friendly. The scheme fits well with the QoS model defined by 3GPP LTE [1] where the Lo and La markings have a straightforward mapping into the set of QoS class identifiers (QCI) - i.e., QCI 6, 8 or 9 for Lo and QCI 7 for La - and can be used by the radio resource manager in the eNodeB to inform its scheduling decisions. This makes it particularly attractive for use on the mobile access network where today Internet-bound flows are typically bundled together into one "default bearer" whose QCI (typically, 9) has latency and loss rate targets (i.e., 300 ms and 10-6, respectively) that are incompatible with real-time traffic requirements. An additional application of LoLa to the mobile network is its potential impact on handover. In [27], the authors explore the behaviour of TCP flows that go through different kinds of handover in LTE and find that seamless handover is low-latency friendly whereas lossless handover helps with high-throughput flows. If available, the Lo and La signals would help the mobile network choosing the most appropriate handover

Horizon 2020
European Union funding
for Research & Innovation

strategy for a certain user at a certain time.

One drawback of LoLa as currently specified is that it is based on DSCP. Although use of DSCP is perfectly reasonable within the boundaries of the same organisation – e.g., if applied to flows between users attached to the Radio Access Network (RAN) and application servers sitting in the SGi-LAN of the same mobile operator, when the marking needs to survive organisational boundaries, then a strongly authenticated end-to-end signal as proposed in [45], would provide a better alternative, which avoids easy theft-of-service or, even worse, denial-of-service by on-path attackers. Alternative bearers for the LoLa signal, besides an explicit path layer mechanism as proposed by PLUS, could include the QUIC [17] header, or the UDP options [38] framework (also see below in section 3.6.

3.5 Congestion Measurement

Congestion Exposure (ConEx) [23] is an approach that has been standardized as an IPv6 extension header [18]. The extension header carries an Destination Option that provides two bits to indicate that a previously sent packet was either congestion marked with the ECN Congestion Experienced (CE) codepoint or lost. A network node can use this information, similar to counting retransmission for TCP flow, to measure congestion for monitoring purposed or perform congestion policing based on these measurement information and per-user congestion allowances.

The PLUS Extended Header provides similar information with the PCF Type 1. However, instead of using one bit for each lost or marked packet it exposes directly the counters for lost or marked packets (since flow start) as maintained by the sender and estimated to measure congestion in the network.

A similar approach to the ConEx-based one-bit signaling was also proposed for congestion measurements in QUIC. However, with the potential encryption of the packet number, this information can not be used to detect reordering on-path anymore, and therefore a different mechanism is currently under discussion that can be used to detect reordering as well as loss.

In this proposal one or multiple bits are used to provide square-wave signal(s) with different frequencies. E.g if one signal toggles between one and zero with every packet this could be used to reliably detected single-packet reordering. A signal that changes after a higher fixed number of packets provides better information about packet loss and the reorder extend (meaning how far a packet got reordered). This with such a signal the choice of the square wave determines how reliably and accurate loss and reordering can be detected and measured, of course depending on the expected loss and reordering pattern. The project is currently investigating the trade-offs and actively participates in the discussion in the QUIC working group.

3.6 PMTU Discovery

Protocol headers added by middleboxes in the network path can reduce the maximum packet size that is supported for a particualr network path. TCP can discover an appropriate Path MTU when middleboxes are present through mechanisms such as MSS-Clamping or Packetization Layer PMTU. However, many modern transports are based on UDP, and no currently standardised methods exist to allow a UDP sender to find a suitable the maximum segment size for datagrams carried through all middleboxes a path. In PLUS we address this problem with PCF Type 2 that actively enables middlebox to communicate their maximum support MTU to the endpoints.

Datagram PLPMTUD [11] is an IETF work in progress that seeks to add MTU Discovery for Datagram Packetization Layers (PLs) and thereby UDP based applications. In the absent of an explicit middlebox cooperation mechanism, it uses both active probes and ICMP-based signals from the network to perform MTU searching on the network. Signals may be provided from the network when an ICMP Packet Too Big (PTB) message is observed or an upper layer signal can provide a known MTU to the Datagram Packetization Layer.

Datagram PLPMTUD requires that the Packetization Layer be able to:

- Signal local MSS
- Send Probe packets
- Provide Reception Feedback
- Parse PTB Signals

UDP Options [39] is an IETF work in progress that seeks to add TCP style options to UDP. The UDP Option space is created by using redundancy in the UDP specification between the payload length field in the IP header and the length field in the UDP header. This redundancy is used in the proposed specification to indicate an option offset for trailing data appended to a UDP Datagram. UDP Options are encoded using a Type Length Value (TLV) syntax similar to TCP Options.

UDP Options can be used to carry additional metadata to application data on a UDP flow, resembling the signalling provided in a PLUS PCF. The primary role of UDP Options is for end-to-end communication, such as e.g. a timestamp option similar as specified for TCP. Most options are at least 2 bytes long, however, there are three options types used for framing and describing the option space. These options are the End of Option List (EOL), No Operation (NOP), intended for padding to byte alignment boundaries, and the OCS (Option Checksum) used to manage integrity of the option area. For PLPMTUD a new UPD option could be used that carries the needed information for the path probing and thereby exposed the desired PMTU to the network.

The project more generally explores how UDP options can be used for middlebox cooperation on a network path, with a current focus on experiments on the use of UDP options to allow middleboes to engage in Datagram PLPMTUD.

4 Security and Privacy Analysis of Middlebox Cooperation Mechanisms

The deployment of explicit middlebox cooperation mechanisms, such as provided by PLUS, provides information to network devices on the path, usually in the clear as we cannot require the establishment of a cryptographical relationship between the endpoints and any middlebox function on the path. In contrast, the deployment of encrypted transport protocols is intended to reduce the information available to devices on path, to prevent malicious manipulation or inadvertant disclosure of information beyond the endpoints. Adding explicit communication between endpoints and on-path devices must therefore be done carefully. Any additional information exposure must consider the security and privacy of the data carried by and the internal operation of the overlying transport. Further, it is especially important to prevent unintended exposure of information from the encrypted payload that could be used for unauthorized on-path inspection, and design these middlebox cooperation carefully to prevent the creation of new attack channels against transports using middlebox cooperative features.

This chapter examines the kinds of attacks that could be performed against new and existing protocol that provide middlebox cooperative features, with PLUS as an example for a protocol that was explicitly designed to enable middlebox cooperation, as well as other examples that can already be found in existing protocols. We consider the middlebox cooperation mechanisms in light of an idealized attacker model given in RFC 7624 [6], and compare it to the kinds of information leaked implicitly by current transport protocols.

This is an abbreviated version of a more complete analysis contained in a project-internal whitepaper “Red Team Analysis of Middlebox Cooperation Protocols” by Thomas Fossati, Roman Müntener and Stephan Neuhaus, with additional contributions by Gorry Fairhurst and Mirja Kühlewind.

4.1 Terms and Attacker Model

For the purposes of this chapter, we will assume that any protocol has zero or more of the following components:

Cleartext header fields are fields appearing on each packet in the protocol that are visible to path elements in the clear, with semantics that are predefined by the protocol specification. Header fields are collected in a header that either has a fixed or variable size.

Scratch space is defined as fields or “space” in a packet header that may be re-written by middleboxes on the path. The difference between scratch space and e.g. header injection is that scratch space is set by the sender of the packet and (re-)writing the space will change the bits but not add any additional bits.

Integrity protection may or may not be applied to any part of the packet, except scratch space. If used, it ensures that the receiver can detect unauthorized changes by middleboxes to information in the header fields or the rest of the protocol’s content. Part of the packet that are not integrity protected, respectively, can be used as scratch space, even if not intended by the actual protocol specification as long as the intended end-to-end functionality is not hampered.

European Commission | Horizon 2020 European Union funding for Research & Innovation

We call the endpoint that initiates communication the client, and the other endpoint the server. We call the endpoint that sends a packet the sender, and the other endpoint the recipient.

We take RFC 7624 [6] as the basis of our attacker model. In [6] attacker is wholly passive, i.e., can look, but not modify packets. In contrast, we also consider an adversary that is allowed *certain* modification of the packet if this furthers its goals. As such, our attackers:

- can observe every packet of all communications at any hop in any network path between the endpoints;
- can observe data at rest in any intermediate system between the endpoints controlled by the sender and the recipient;
- can share information with other such attackers; and
- may take other actions with respect to these communications (e.g., blocking, modification, injection, etc.), as long as these actions do not cause the communication to be totally disrupted.

That means, we also consider an attacker that can perform *active attacks*, short of denial of service. Examples of blocking actions that fall short of denial of service include:

- dropping a single packet to force retransmission by the sender's transport layer;
- blocking requests to certain DNS servers to force use of other DNS servers;
- blocking TLS-protected communications to force unprotected communications;
- shutting down a selected number of links in a multi-path scenario and thus forcing the multi-path transport to use specific links, on which surveillance may be easier.

In general, the attacker wants to to remain undetected if at all possible, but this requirement might be relaxed if the consequences of detection are outweighed by the usefulness of the attack. This can happen for example if the attacker is so powerful that detection is without consequence (low risk), or if the attack, if successful, would give the attacker information of extremely high value (high pay-off).

We assume that communication is encrypted and integrity protected by default at or above the transport layer, because this is increasingly the case. By default the only information that an attacker can extract is metadata. However, it has been shown that metadata can be used to infer information about the data itself. For example, flow records that carry only the 5-tuple and the duration of the flow have been used to identify Skype traffic [43].

Out-of-band identification methods, e.g. linking a flow's 5- or 6-tuple with an identifier or using some other communication to export this linkage, is not considered, because it is practically impossible for users and remote endpoints to detect and defeat.

In the following section we will discuss different attacks that can be run based on this attacker model and evaluate the risk of different middlebox cooperation approaches.

4.2 Side Channel Exfiltration

Providing a mechanisms for network elements to signal information to the endpoints, in-band, by appending information to a packet of a flow, either explicitly by providing some scratch space or through the (mis-)use of unprotected header fields, bears the risk that this (side) channel can be utilized to exfiltrate information about the encrypted part of the communication or non-public metadata about the application or user, such as personal user data that is known by the middlebox.

One scenario is for a middlebox on the path—most likely a node close to the sender—to inject information that it knows about the sender into the scratch space. This information may then be read by middleboxes later on-path, thus leaking identifying information about the user to other on-path devices.

However, this kind of side channel exfiltration is complicated by at least two factors. First, the amount of scratch space per packet is usually small compared to the actual payload data; and second, changing bits that were indented for a different use could change the semantics of the protocol and must therefore be done carefully to not cause unintended side efforts. For example, if information that is indicating the intended Quality of Service treatment, like the DiffServ codepoint or the PLUS (L)atency Flag, is overwritten, this might have an negative impact on the traffic itself, e.g. a packet that unexpectedly is more likely to be dropped at a congested router router might have an highly negative impact on the congestion control decreasing the flow throughput overly drastically.

In the case of PLUS, the maximum scratch space per packet is restricted by the 6 bit PCF length field to 63 bytes of not integrity protected header space. However, the extended header is controlled by the sender and as such not every packet might carry a that large scratch space or any scratch space at all. Further, the PCF type may restrict the amount of valid values for a certain PCF which further increased the ability to detect unauthorized side channel exflitration. Still to be effective, integrity protection has to be checked and acted upon. The PLUS spec only says the receiver *should* (as in RFC 2119 [8]) drop packets that fail the integrity verification. The absence of a strict requirement on the receiving stack to fail hard makes room—at least in theory—for 3rd party manipulation of any header bits or payload data.

In TCP, however, none of the header bytes are integrity protect. This does not mean that all header bits can be used as side channel because changing the bits can change the protocol behavior or make it disfunctional. However, there are opportunities for a middlebox to change bits without being detectable, e.g. rewriting the initial sequence number, also see [41] for other metadata injection attacks for both TCP as well as IP.

Even if the header is integrity protection, side channel exflitration can work as long as such changes are undone before the packet reaches the remote endpoint. In this case the attacker can exfiltrate data to the network portion between the attacker's ingress and egress nodes. This can be done in a way that is undetectable by either endpoint. In a more generic case, this type of attack requires, however, a out-of-band side channel for coordination between the attacker's nodes.

While these side channels seems tempting on the first view, it actually is often easier for path devices to utlise tunnel headers for adding metadata that might be needed for a variety of purposes.

However, if done carefully this attack is hard to detect, and therefore could be desirable to the attacker. But it is an *active* attack that might not be preferred by a mostly passive adversary.

4.3 Coercion

Another concern about middlebox cooperation is that access networks could require endpoints to supply packets with specific information or else they would refuse to forward the packets, or refuse to forward them speedily. This would follow the familiar pattern in which privacy advocates lobby for, and get, opt-in solutions to data tracking, only to find that companies only offer their services to users who do in fact opt in. People who choose not to opt in are then unable to use this service. If there is one dominant service of its kind available (e.g., Facebook or Twitter), or if there is only one viable alternative, people may find they have no choice and need to opt in.

Access network providers or even core networks could do the same to ease data exfiltration. They could force endpoints to provide (extra) scratch space, or face the penalty of being dropped or of being put into the slow lane if they do not comply.

This attack is detectable and also active, and would therefore not be preferred by the mosty passive adversary.

4.4 Application Fingerprinting

As described in [34], state of art encrypted traffic analysis based on machine learning can successfully identify the type of transported application (e.g., HTTPS, SMTP, P2P, VoIP, SSH, Skype) with good accuracy and without any need to access the clear-text.

In this context, and despite its limitations (i.e., fuzzy, coarse grained), even a 1-bit signal, such as the L Flag in PLUS, could be used to improve the precision of the classifier. This signal is non-malleable (can not be changed on-path). In contrast, the DiffServ [28] field used to carry the LoLa signal can be and often is updated at edge routers to map to available QoS treatments or to aggregate classes (including in some cases of completely bleaching the field). As such integrity protected header information that is exposed to the path, make the information provided it slightly more reliable signal compared using the DiffServ equivalent.

On the other hand not all bits in the header can be used to improve the traffic analysis of encrypted traffic. Such the utility of these bits for fingerprinting does strongly depending on the information they carry and may or may not be of high value for an attacker.

4.5 Linkability

By *unlinkability* we mean that “[w]ithin a particular set of information, the inability of an observer or attacker to distinguish whether two items of interest are related or not (with a high enough degree of probability to be useful to the observer or attacker)” [10, Section 3.2.]. Linkability on the other hand can lead to identifiability, or it can help classification algorithms to become more accurate by providing ground-truth information about the packets that would not be easily available otherwise.

PLUS as well as some other new protocols, like QUIC, have *connection identifiers* that explicitly provide linkability. A connection identifier is a transport layer construct that allows endpoints and middleboxes to uniquely identify an end-to-end connection/session even if the underlying 5-tuple changes due to NAT re-bindings, connection migration with multi-homing or multi-path. For endpoints, the ability for a connection to survive such a migration is especially important for encrypted protocols where session re-negotiation involves a handshake that is expensive in terms of computation effort and latency. Examples of middleboxes that could utilise this information include load balancers and firewalls.

The presence of a connection identifier obviously introduces linkability. Therefore, a good protocol design should take this into account and seek, as much as possible, to avoid linkability by passive on-path adversaries between multiple source addresses or ports during mobility or NAT rebind scenarios. This may not be always possible—e.g., when clients are unaware of the address change (for example, when passing through NATs.) An example of a design that satisfies this requirement is the a HOTP-based construction [25].

4.6 Localization

While RTT is a useful metric for network monitoring [3], it could also leak information about the physical location of an endpoint. If the speed of signal propagation in a network is known to be v, one could in theory measure the RTT from multiple vantage points and then use that for geolocation. For example, assuming a connection has a RTT of x seconds, the endpoint cannot be further away than $0.5xv$ meters. Each RTT measurement thus yields an exclusion circle, outside of which the endpoint cannot be. If RTT measurements are then made from several sufficiently distributed vantage points, it could be suggested that triangulation would yield the endpoint position with some accuracy. This is an active attack that can be run with basically any protocol that triggers an immediate response to an initially send packet, such as the TCP handshake or just ICMP ping which is actually designed for this use case. However, an attacker would need to have access to a number of well located vantage points. Therefore for an passive attacker, explicitly exposing RTT, as proposed with the PSN and PNE for PLUS or the spin bit for QUIC, gives them access to information provided by vantage points that they would not otherwise have access to.

However, our experiments explored a model of propagation speed that is much more complete than our extremely simplified model as described above [42]. Based on a large set of traceroute-based measurement data from RIPE Altas we can conclude that RTT cannot reliably be used for geolocation. In one experiment, the endpoint was in Milan, and the only RTT measurement that could even locate that endpoint within Italy was one that was made by a process running *on the same machine* as the endpoint. Even a vantage point that was in the same equipment rack (but not on the same machine) as the endpoint could not locate that endpoint to within Italy. Measurements from anywhere else were essentially meaningless due to the large errors.

4.7 Analysis

Based on the previously described attacks and vulnerabilities, this section evaluates the security risk of building a path protocol that reveals information to the network, and hence makes this information available to middleboxes that may exist on the end to end path. Middlebox co-operation mechansims, no matter how it is encoded, will most likely have information with fixed formats and meaning. However, even information with fixed formats can be abused for data exfiltration and linkability. If data is integrity-protected, and data manipulation is required to not disrupt normal operation, the abuse potential is further limited or the data need to be restored to a close-to-original state before being delivered to the receiver. The potential for new abuse comes from introducing explicit non-integrity protected scratch space, which seems to present a larger risk to exfiltrate data.

For intra-network exfiltration, the adversary has to control a portion of the path through which the packets travel. Such an adversary could also use other means to accomplish the same effect, without even changing the original packets. One easy way would be for an adversary to send control datagrams to an on-path device. Or, network operators can encapsulate packets or add tunnel headers, or set tags in existing encapsulations.

As such the additional risk provided by explicit middlebox cooperation mechanism is rather limited and the security risks of each function need to be weighed against the benefit from making specific information visible to the network. Integrity protect and endpoint control can further mitigate these risks as data manipulation can be detected and the utilized vulnerable functions disable, if use if optional, or the connection terminated.

European Commission | Horizon 2020 European Union funding for Research & Innovation

5 Post Sockets: A protocol-independent API for flexible deployment of new protocols

The Post Sockets API described in this chapter provides a flexible interface to the transport layer, regardless of the transport protocol in use or the capabilities it provides. The primary aim of Post Sockets is to provide a compelling, modern, application-centric interface that provides the flexibility to slot present and future transport protocol stacks under an application without requiring a rewrite. Post's design is focused on the following principles:

- Explicit Message orientation, with framing and atomicity guarantees for Message transmission.
- Asynchronous reception, allowing all receiver-side interactions to be event-driven.
- Explicit support for multistreaming and multipath transport protocols and network architectures, including protocols for mobility support.
- Long-lived Associations, whose lifetimes may not be bound to underlying transport connections. This allows associations to cache state and cryptographic key material to enable 0-RTT resumption of communication, and for the implementation of the API to explicitly take care of connection establishment mechanics such as connection racing [50] and peer-to-peer rendezvous [33].
- Protocol stack independence, allowing applications to be written in terms of the semantics best for the application's own design, separate from the protocol(s) used on the wire to achieve them. This enables applications written to a single API to make use of transport protocols in terms of the features they provide, following the approach of the the IETF Transport Services (TAPS) working group [12].

This last feature is the most important for the deployability of middlebox cooperation protocols in the Internet. First, it allows protocols which may or may not deploy on the Internet due to interference to be dynamically selected based on the properties of the paths over which they will run. For example, consider the case of attemtping to use a future reliable transport over PLUS on a network which blocks all UDP other than DNS to an internal resolver. Currently, an application would need to implement its own fallback mechanism; by dynamically selecting both the primary and fallback stack via policies in Post Sockets, this fallback can be delegated to the transport layer. Second, Post Sockets' policy mechanism allows the preference or avoidance of PLUS and/or certain PCFs to be set by the user and/or application developer.

This chapter represents the state of Post Sockets development reported in our workshop paper [46], and is taken largely from that paper. Post Sockets is under active development and standardization as the basis for an abstract, protocol-independent interface to the transport layer in the IETF's TAPS working group[1], in cooperation with the H2020 NEAT project.

[1]see `https://github.com/taps-api/drafts`

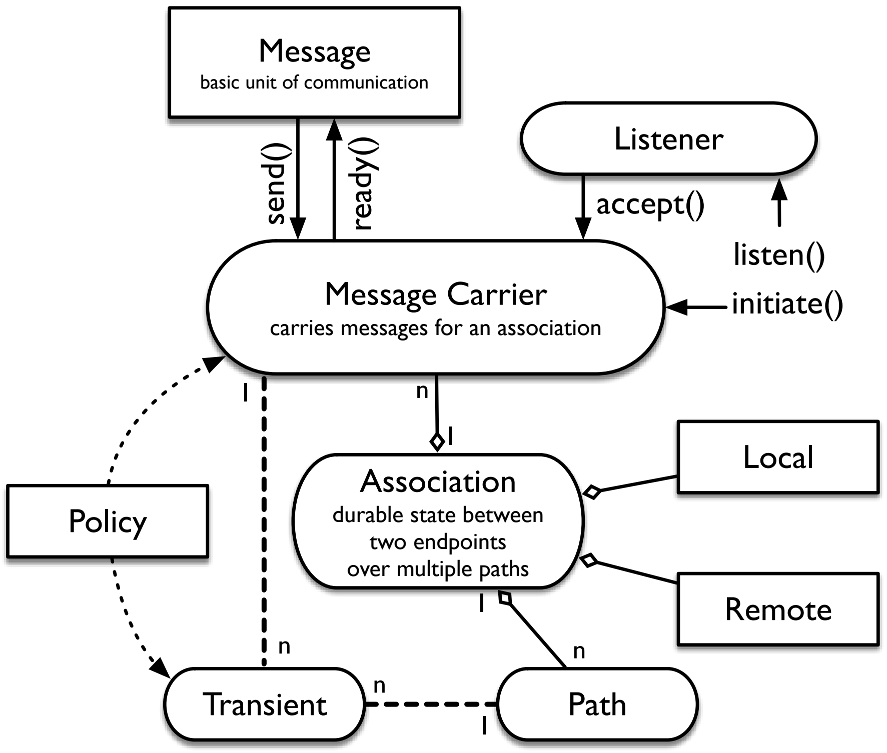

Figure 5: Abstractions and relationships in Post Sockets.

5.1 Overview and Concepts

Post Sockets replaces the traditional `SOCK_STREAM` abstraction with an Message abstraction, which can be seen as a generalization of the Stream Control Transmission Protocol's [36] `SOCK_SEQPACKET` service. The API is centered around a *Message Carrier* which logically group Messages for transmission and reception. A Carrier can be created actively via ***initiate()*** or passively via a ***listen()***; the latter creates a Listener from which new Carriers can be ***accept()***ed. For backward compatibility, these Carriers can also be opened as Streams, presenting a file-like interface to the network as with `SOCK_STREAM`. *Messages* may be created explicitly and ***send()*** over the Carrier, or implicitly through a simplified interface which uses default message properties (reliable transport without priority or deadline, which guarantees ordered delivery over a single Carrier when the underlying transport protocol stack supports it). Whenever a Message is fully reassembled at receiver side, an asynchronous callback will notify the application that a received message is ***ready()***.

Message Carriers are bound to an long-lived *Association* which stores information about the identity of a *Local* and a *Remote* endpoint, as well as cryptographic session resumption parameters. New Message Carriers will reuse an Association if they can be carried from the same Local to the same Remote over an existing or newly found network *Path*; this re-use of an Association together with implementation of a *Policy* or a set of Policies defined by the application or system may imply the creation of a new *Transient* representing a concrete Protocol Stack Instance (PSI) assigned to an active Carrier.

The relationships among these elements are shown in Figure 5 and detailed in this section.

5.1.1 Message Carrier

A Message Carrier (or simply Carrier) is a transport protocol stack-independent interface for sending and receiving Messages between an application and a remote endpoint. Sending a Message over a Carrier is driven by the application, while receipt is driven by the arrival of the last packet that allows the Message to be assembled, decrypted, and passed to the application. Receipt is therefore asynchronous; given the different models for asynchronous I/O and concurrency supported by different platforms, it may be implemented in any number of ways. The abstract API provides only for a way for the application to register how it wants to handle incoming messages.

A Message Carrier that is backed by current transport protocol stack state (such as a TCP connection) is said to be *active*: messages can be sent and received over it. A Message Carrier can also be *dormant*: there is long-term state associated with it (via the underlying Association; see next section), and it may be able to reactivated, but messages cannot be sent and received immediately.

To exchange messages with a given remote endpoint, an application may initiate a Message Carrier given its Remote and Local identities; this is an equivalent to an active open. There are four special cases of Message Carriers, supporting different initiation and interaction patterns:

Listener A Listener only responds to requests to create a new Carrier, analogous to a server or listening socket in the present sockets API. Instead of being bound to a specific remote endpoint, it is bound only to a local identity. Accepting an incoming request will fork a fully fledged Message Carrier.

Source A Source is a special case of Message Carrier over which messages can only be sent, intended for unidirectional applications such as multicast transmitters.

Sink A Sink is a special case of Message Carrier over which messages can only be received, intended for unidirectional applications such as multicast receivers.

Responder A Responder may receive messages from many remote sources, for cases in which an application will only ever send Messages in reply back to the source from which a Message was received. This is a common implementation pattern for servers in client-server applications.

A Message Carrier may be morphed into a Stream, in order to provide a strictly ordered, reliable service as with `SOCK_STREAM`. Morphing a Message Carrier into a Stream should return a file-like object as appropriate for the platform implementing the API. Typically, both ends of a communication using a stream service will morph their respective Message Carriers independently before sending any data, based on application layer knowledge about the configuration used by the other endpoint. This is mainly for backwards comparability with existing non-Post-Sockets stacks as well as an easy path for migration for existing application implementations. If supported by the underlying transport protocol stack, a Stream may be forked: creating a new Message Carrier associated with a new Message Carrier at the same remote endpoint.

Horizon 2020
European Union funding
for Research & Innovation

5.1.2 Message

A Message is an atomic unit of communication between applications. A Message that cannot be delivered in its entirety within the constraints of the network connectivity and the requirements of the application is not delivered at all. Messages can represent both relatively small structures, such as requests in a request/response protocol such as HTTP; as well as relatively large structures, such as files of arbitrary size in a file system. In the general case, there is no mapping between a Message and packets sent by the underlying protocol stack on the wire: the transport protocol may freely segment messages and/or combine messages into packets.

Applications can register callbacks to be asynchronously notified of three events on Messages they have sent: that the Message has been transmitted, that the Message has been acknowledged by the receiver, or that the Message has expired before transmission/acknowledgment. Not all transport protocol stacks will support all of these events.

A Message has a set of properties used by the sending transport, along with information about the properties of the Paths available, to determine when to send which Message down which Path. Post Sockets provides a set of core properties (Lifetime, Niceness, Latency Sensitivity, Reordering Sensitivity, Immediacy, and Idempotence). In addition, Messages may have additional arbitrary properties which can be used by the underlying protocol stacks for further purposes.

The core properties defined by Post Sockets are as follows:

Lifetime A wallclock duration before which the Message must be available to the application layer at the remote end or otherwise will be useless. As soon as it is known that a lifetime cannot be met, the Message is discarded. Messages without lifetimes are sent reliably if supported by the transport protocol stack. Lifetimes are also used to prioritize Message delivery. There is no guarantee that a Message will not be delivered after the end of its lifetime; for example, a Message delivered over a strictly reliable transport will be delivered regardless of its lifetime. Depending on the transport protocol stack used to transmit the message, these Lifetimes may also be signaled to path elements by the underlying transport, so that path elements that realize a lifetime cannot be met can discard frames containing the Messages instead of forwarding them. Messages with a lifetime

Niceness A priority among other messages sent over the same Message Carrier in an unbounded hierarchy most naturally represented as a non-negative integer. By default, Messages are in niceness class 0 which is highest priority. By prioritization of certain messages against others, e.g., blocking of smaller, latency-sensitive messages by large non-latency-sensitive messages can be avoided. Niceness may be translated to a priority signal for exposure to path elements (e.g., DSCP code-point) to allow prioritization along the path.

Latency Sensitivity Marking a message as *latency sensitive* signals that it is better to drop the message than to delay it; together with Lifetime, this is used by underlying transports over PLUS to set the L bit in the PLUS Basic Header.

Reordering Sensitivity Marking a message as *reordering insensitive* signals that the application does not care about the order the message is received in relative to other messages

Horizon 2020
European Union funding
for Research & Innovation

over the same Carrier; this can be used both as a hint to the underlying transport in selecting paths, as well as to set the R bit in the PLUS Basic Header of underlying transports that are PLUS-encapsulated.

Immediacy Marking a message as *immediate* signals to the transport protocol stack that its application semantics require it to be sent out immediately, instead of waiting to be combined with other messages or parts thereof (i.e., for media transports and interactive sessions with small messages).

Idempotence If marked as *idempotent* the underlying transport protocol stack knows that its application semantics make it safe to send in situations that may cause it to be received more than once (i.e., for 0-RTT session resumption as in TCP Fast Open, TLS 1.3, and QUIC).

5.1.3 Transient

A Transient represents a binding between an active Carrier and the instance of the transport protocol stack that implements it. A Transient contains ephemeral state for a single transport protocol stack over a single Path at a given point in time. A Carrier may be served by multiple Transients at once, e.g., when implementing multi-path communication such that the separate paths are exposed to the API by the underlying transport protocol stack. Each Transient serves only one Message Carrier, although multiple Transients may share the same underlying protocol stack; e.g., in a multi-streaming protocol.

Transients are generally not exposed by the API, though they may be accessible for debugging and logging.

5.1.4 Association

An Association contains the long-term state necessary to support communications between a Local and a Remote endpoint, such as cryptographic session resumption parameters or rendezvous information; information about the policies constraining the selection of transport protocols and local interfaces to create Transients to carry Messages; and information about the Paths through the network available between them. Three inputs are needed to establish an Association: a *remote*, a *local*, and a *policy*.

Remote A Remote represents information required to establish and maintain a connection with the far end of an Association: name(s), address(es), and transport protocol parameters that can be used to establish a Transient; transport protocols to use; information about public keys or certificate authorities used to identify the remote on connection establishment; and so on. Each Association is associated with a single Remote, either explicitly by the application (when created by the initiation of a Carrier) or a Listener (when created by forking a Carrier on passive open).

A Remote may be resolved, which results in zero or more Remotes with more specific information. For example, an application may want to establish a connection to a website identified by a URL. This URL would be wrapped in a Remote and passed to a call to initiate a Carrier. The first pass resolution might parse the URL, decomposing it into a

Horizon 2020
European Union funding
for Research & Innovation

name, a transport port, and a transport protocol to try connecting with. A second pass resolution would then look up network-layer addresses associated with that name through DNS, and store any certificates available from DANE. Once a Remote has been resolved to the point that a transport protocol stack can use it to create a Transient, it is considered fully resolved.

Local A Local represents all the information about the local endpoint necessary to establish an Association or a Listener: interface, port, and transport protocol stack information, as well as certificates and associated private keys to identify it.

Policy A Policy describes restriction and requirements from the application to select and configure Transients for a communication between a Local and a Remote. For instance, an application may require or prefer certain transport features [13] in the PSI(s) used by the Transient(s) for a given Message Carrier. It may prefer Paths over one interface to those over another (e.g., WiFi access over LTE when roaming on a foreign LTE network, due to cost). Policy information, encapsulating application intent and constraint, is thus expressed as implementation-specific configuration for the Message Carrier and the Transient(s).

Policies are the primary interface between applications and PLUS as presently defined. First, since the policy determines which transport protocol(s) will be tried and selected, it determines which signals will be exposed to the path (e.g., certain protocols may use the R and L bits in the Basic Header based upon their own sensitivities to reordering and loss and/or the per-Message hints provided by the application), as well as whether a PLUS-encapsulated protocol will be used at all. Second, policies can hint to PLUS what information should be requested or exposed via the Extended Header; a PSI implementing PLUS could provide a policy option to allow the application to configure the use and frequency of the Loss and Congestion Exposure PCF, for instance, allowing fine-grained control of the tradeoff between the passive measurability of the traffic and the overhead required.

5.1.5 Path

A Path represents information about a single path through the network known by an Association, in terms of source and destination network and transport layer addresses within an addressing context, and the provisioning domain [5] of the local interface. This information may be learned through a resolution, discovery, or rendezvous process (e.g., DNS, ICE), by active or passive measurements taken by the transport protocol stack, or by some other path information discovery mechanism.

The set of available properties is a function of the transport protocol stacks such as the MTU, expected one-way delay, expected probability of packet loss, expected maximum available data rate or reserved data rate. These path properties may be derived from information learned from the middlebox cooperation protocol, among other sources.

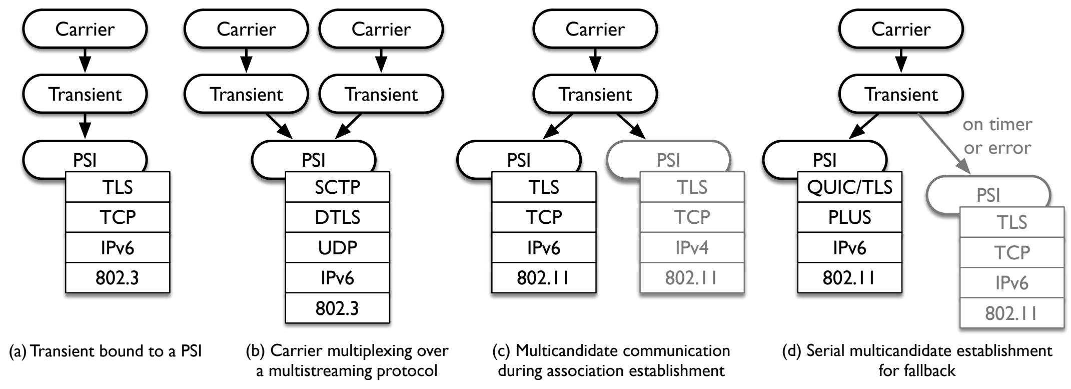

Figure 6: Protocol stack instances; multistreaming and happy-eyeballing

5.1.6 Protocol Stack Instance (PSI)

A PSI encapsulates an arbitrary stack of protocols (e.g., TCP over IPv6, SCTP over DTLS+PLUS over UDP over IPv4). PSIs provide the bridge between the interface (Carrier) plus the current state (Transients) and the implementation of a given set of transport services [13]. A given implementation makes one or more possible protocol stacks available to its applications. Selection and configuration among multiple PSIs is based on system-level or application policies, as well as on network conditions in the provisioning domain in which a connection is made. For example, figure 6(a) shows a TLS over TCP stack, usable on most network connections. Protocols are layered to ensure that the PSI provides all the transport services required by the application. A single PSI may be bound to multiple message carriers, as shown in figure 6(b): a multistreaming transport protocol like QUIC or SCTP can support one carrier per stream. Where multistreaming transport is not available, these carriers could be serviced by different PSIs on different flows. Multiple PSIs are bound to a single transient during establishment, as shown in figure 6(c). Here, the losing PSI in a happy-eyeballs race will be terminated, and the carrier will continue using the winning PSI. A special case of racing is fallback, as shown in figure 6(d). Here, one stack is given preference, and the second will only be initiated after a timer expires or this stack fails. Fallback can be used to spur deployment of PLUS-encapsulated transport protocols, by providing a generic mechanism to transparently use TCP when PLUS is not available or useful on a given path.

5.2 Deployment Incentives and Standardizations Efforts

The current widely deployed BSD sockets interface forces a hard binding between applications and transport protocols, making is impossible to deploy a new protocol without changing every application to support it. Post Sockets addresses this problem, providing an initial, necessary step for rapid large-scale deployment of new transport protocols and flexible use of stacked protocols, e.g. enabling transport protocol header encryption and/or explicit middlebox cooperation. This means an application that makes use of this new transport-independent interface can benefit from new protocols and protocol mechanism automatically without any changes in the code or configuration. Further, the transport layer underneath the Post Sockets interface can provide additional functionality such as protocol racing, path testing, or rendezvousing

European Commission | Horizon 2020 European Union funding for Research & Innovation

which reduces the application's complexity because these mechanism do not then need to be implemented for each application separately.

Post Sockets is the basis of a current project contribution to the IETF TAPS working group, the Transport Services Architecture and Interface, which aims to standardize the concepts elaborated in Post Sockets as an abstract API, in order to drive future deployment of conceptually similar implementations of networked applications to enable transport stack flexibility.

6 Conclusion

This deliverable provides a specification of the PLUS middlebox cooperation protocol as developed by the MAMI project. It further presents the project's current efforts to apply mechanisms developed for use with PLUS and based on our experimentation experience with PLUS to existing and new transport protocols. These efforts include extensions of the QUIC wire image to support measurability, signaling schemes for in-network low latency support, extensions to encryption protocols such as DTLS to support in-network functions like load balancing, as well as a general concept for state management that can be applied to different protocols that provide certain signals to the network.

The project is currently performing further experiments on middlebox cooperation. These are intended as a proof of concept showing that the proposed solutions and approaches for specific middlebox cooperation use cases are viable. Despite PLUS forming a consistent specification for the problem of middlebox cooperation, we do not expect the deployment of PLUS as specified in this deliverable without a more radical trend for redesigning protocol stacks in the Internet protocol standardization community. In our experiments, we will show how these concepts can be applied to other protocols, especially the QUIC protocol which is a new, encrypted transport that is currently under standardization in the IETF with an expected large impact on the near-future Internet traffic.

This deliverable further provides a generalized security analysis of middlebox cooperation mechanisms, whether explicitly designed for cooperation (such as PLUS, or certain QUIC header information), or implicitly used for cooprtation. This analysis defines an attacker model and discusses different attacks using side channel exfiltration or coercion, as well as vulnerabilities that could be used for fingerprinting, linkability, or localization. This analysis showed that the security risks of new path layer middlebox cooperation mechanisms that provide specific information to in-network functions need to be weighed against the benefit from making these information visible to the network, for each function separately.

Finally, this deliverable also documents parts of the project's current work aimed at providing a more flexible transport layer to support applications in runtime selection of protocols as well as interfaces and paths. Runtime protocol selection, in particular, allows fallback in case a middlebox cooperation approach like PLUS is not available on a given path or interface. To enable these selection mechanisms, and thereby support the deployability of new protocols, it is necessary to redesign the application-transport interface to flexibly support new protocol and middlebox cooperation features when and where they are available, while not increasing the complexity of each application.

References

[1] Technical Specification Group Services and System Aspects; Policy and charging control architecture (Release 14). Technical specification, 3rd Generation Partnership Project, June 2017.

[2] M. Allman, R. Beverly, and B. Trammell. Principles for Measurability in Protocol Design. *ACM SIGCOMM Computer Commuication Review*, 47(2), April 2017.

[3] M. Allman, R. Beverly, and B. Trammell. Principles for measurability in protocol design. *SIGCOMM Comput. Commun. Rev.*, 47(2):2–12, May 2017.

[4] M. Allman, W. M. Eddy, and S. Ostermann. Estimating Loss Rates with TCP. *SIGMETRICS Perform. Eval. Rev.*, 31(3):12–24, Dec. 2003.

[5] D. Anipko. Multiple provisioning domain architecture. RFC 7556, IETF, June 2015.

[6] R. Barnes, B. Schneier, C. Jennings, T. Hardie, B. Trammell, C. Huitema, and D. Borkmann. Confidentiality in the face of pervasive surveillance: A threat model and problem statement. RFC 7624, IETF, Aug. 2015.

[7] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger. TCP Extensions for High Performance. RFC 7323, RFC Editor, September 2014.

[8] S. Bradner. Key words for use in RFCs to indicate requirement levels. RFC 2119, IETF, Mar. 1997.

[9] B. Briscoe, R. Woundy, and A. Cooper. Congestion Exposure (ConEx) Concepts and Use Cases. RFC 6789, IETF, 2012.

[10] A. Cooper, H. Tschofenig, B. Aboba, J. Peterson, J. Morris, M. Hansen, and R. Smith. Privacy considerations for internet protocols. RFC 6973, IETF, July 2013.

[11] G. Fairhurst, T. Jones, M. Tuexen, and I. Ruengeler. Packetization layer path mtu discovery for datagram transports. Internet-Draft draft-ietf-tsvwg-datagram-plpmtud-00, IETF Secretariat, January 2018. `http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-datagram-plpmtud-00.txt`.

[12] G. Fairhurst, B. Trammell, and M. Kuehlewind. Services provided by ietf transport protocols and congestion control mechanisms. RFC 8095, RFC Editor, March 2017.

[13] G. Fairhurst, B. Trammell, and M. Kuehlewind. Services provided by IETF transport protocols and congestion control mechanisms. RFC 8095, IETF, March 2017.

[14] S. Hatonen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti, and M. Kojo. An experimental study of home gateway characteristics. In *Proc. ACM IMC*, 2010.

[15] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 181–194, New York, NY, USA, 2011. ACM.

[16] J. Iyengar and M. Thomson. Quic: A udp-based multiplexed and secure transport. Internet-Draft draft-ietf-quic-transport-01, IETF Secretariat, January 2017. `http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-01.txt`.

[17] J. Iyengar and M. Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. Internet-Draft draft-ietf-quic-transport, Internet Engineering Task Force, Jan. 2018. Work in Progress.

[18] S. Krishnan, M. Kuehlewind, B. Briscoe, and C. Ralli. IPv6 Destination Option for Congestion Exposure (ConEx). RFC 7837 (Experimental), May 2016.

[19] B. T. T. Kuehlewind. The wire image of a network protocol. Internet-Draft draft-trammell-wire-image-00, IETF, 2017.

[20] M. Kuehlewind. and B. Trammell. Manageability of the quic transport protocol. Internet-Draft draft-ietf-quic-manageability-01, IETF, 2017.

[21] M. Kuehlewind, B. Trammell, and J. Hildebrand. Transport-Independent Path Layer State Management. Internet-Draft draft-trammell-plus-statefulness-03, IETF Secretariat, March 2017. `http://www.ietf.org/internet-drafts/draft-trammell-plus-statefulness-03.txt`.

[22] M. Kühlewind, S. Neuner, and B. Trammell. On the State of ECN and TCP Options on the Internet. In *Proceedings of the 14th International Conference on Passive and Active Measurement*, PAM'13, pages 135–144, Hong Kong, China, 2013. Springer-Verlag.

[23] M. Mathis and B. Briscoe. Congestion Exposure (ConEx) Concepts, Abstract Mechanism, and Requirements. RFC 7713 (Informational), Dec. 2015.

[24] M. Mathis and J. Heffner. Packetization Layer Path MTU Discovery. RFC 4821, RFC Editor, March 2007. `http://www.rfc-editor.org/rfc/rfc4821.txt`.

[25] N. Mavrogiannopoulos, H. Tschofenig, and T. Fossati. Datagram transport transport layer security (DTLS) transport-agnostic security association extension. Internet-Draft draft-mavrogiannopoulos-tls-cid-01, IETF Secretariat, May 2017. `https://datatracker.ietf.org/doc/draft-mavrogiannopoulos-tls-cid/`.

[26] B. T. R. M. S. N. Mirja Khlewind, Tobias Bhler and G. Fairhurst. A path layer for the internet: Enabling network operations on encrypted protocols. In *In-proceedings of the International Conference on Network and Service Management (CNSM)*. IEEE, 2017.

[27] B. Nguyen, A. Banerjee, V. Gopalakrishnan, S. Kasera, S. Lee, A. Shaikh, and J. Van Der Merwe. *Towards understanding TCP performance on LTE/EPC mobile networks*, pages 41–46. Association for Computing Machinery, 2014.

[28] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the differentiated services field (DS Field) in the IPv4 and IPv6 headers. RFC 2474, IETF, December 1998.

[29] M. Perumal, D. Wing, R. Ravindranath, T. Reddy, and M. Thomson. Session Traversal Utilities for NAT (STUN) Usage for Consent Freshness. RFC 7675 (Proposed Standard), Oct. 2015.

[30] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, RFC Editor, September 2001. `http://www.rfc-editor.org/rfc/rfc3168.txt`.

[31] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, Jan. 2012.

[32] E. Rescorla, H. Tschofenig, T. Fossati, and T. Gondrom. The Datagram Transport Layer Security (DTLS) Connection Identifier. Internet-Draft draft-ietf-tls-dtls-connection-id, Internet Engineering Task Force, Dec. 2017. Work in Progress.

[33] J. Rosenberg. Interactive connectivity establishment (ICE): A protocol for network address translator (NAT) traversal for offer/answer protocols. RFC 5245, IETF, April 2010.

[34] W. M. Shbair, T. Cholez, J. François, and I. Chrisment. A Multi-Level Framework to Identify HTTPS Services. In *IEEE/IFIP Network Operations and Management Symposium (NOMS 2016)*, pages p240–248, Istanbul, Turkey, Apr. 2016. IEEE/IFIP, IEEE.

[35] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252, June 2014.

[36] R. Stewart. Stream Control Transmission Protocol. RFC 4960, IETF, September 2007.

[37] S. D. Strowes. Passively measuring TCP round-trip times. *Commun. ACM*, 56(10):57–64, Oct. 2013.

[38] D. J. D. Touch. Transport Options for UDP. Internet-Draft draft-ietf-tsvwg-udp-options, Internet Engineering Task Force, Jan. 2018. Work in Progress.

[39] J. Touch. Transport options for udp. Internet-Draft draft-ietf-tsvwg-udp-options-02, IETF Secretariat, January 2018. `http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-udp-options-02.txt`.

[40] B. Trammell. Abstract Mechanisms for a Cooperative Path Layer under Endpoint Control. Internet-Draft draft-trammell-plus-abstract-mech-00, IETF, Sep 2016.

[41] B. Trammell. Detecting and defeating tcp/ip hypercookie attacks. Internet-Draft draft-trammell-privsec-defeating-tcpip-meta-00, IETF Secretariat, July 2016.

[42] B. Trammell. On the suitability of rtt measurements for geolocation, Aug. 2017. https://github.com/britram/trilateration/blob/master/paper.ipynb.

[43] B. Trammell, E. Boschi, G. Procissi, C. Callegari, P. Dorfinger, and D. Schatzmann. Identifying skype traffic in a large-scale flow data repository. In *Proceedings of the Third International Conference on Traffic Monitoring and Analysis*, TMA'11, pages 72–85, Berlin, Heidelberg, 2011. Springer-Verlag.

[44] B. Trammell, D. Gugelmann, and N. Brownlee. Inline Data Integrity Signals for Passive Measurement. In *Proc. Sixth Int. Wksp. on Traffic Measurement and Analysis*, London, England, April 2014.

[45] B. Trammell and M. Kuehlewind. Path Layer UDP Substrate Specification. Internet-Draft draft-trammell-plus-spec-01, IETF Secretariat, March 2017. http://www.ietf.org/internet-drafts/draft-trammell-plus-spec-01.txt.

[46] B. Trammell, C. Perkins, and M. Kühlewind. Post sockets: Toward an evolvable network transport interface. In *Proceedings of Networking 2017 Workshop on Future Internet Transport*, Stockholm, Sweden, June 2017.

[47] B. Trammell, P. D. Vaere, R. Even, G. Fioccola, T. Fossati, M. Ihlar, A. Morton, and E. Stephan. The addition of a spin bit to the quic transport protocol. Internet-Draft draft-trammell-quic-spin-01, IETF Secretariat, December 2017. http://www.ietf.org/internet-drafts/draft-trammell-quic-spin-01.txt.

[48] H. Tschofenig and T. Fossati. Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things. RFC 7925, July 2016.

[49] M. Williams, M. Txen, and R. Seggelmann. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520, Feb. 2012.

[50] D. Wing and A. Yourchenko. Happy eyeballs: Success with dual-stack hosts. RFC 6555, IETF, April 2012.

[51] J. You, M. Welzl, B. Trammell, M. Kuehlewind, and K. Smith. Latency Loss Tradeoff PHB Group. Internet-Draft draft-you-tsvwg-latency-loss-tradeoff-00, IETF, March 2016.