Designing Transport-Level Encryption for Datacenter Networks

Tianyi Gao
University of Edinburgh
Edinburgh, United Kingdom
tianyi.gao@ed.ac.uk

Xinshu Ma
University of Edinburgh
Edinburgh, United Kingdom
x.ma@ed.ac.uk

Suhas Narreddy
University of Edinburgh
Edinburgh, United Kingdom
suryasuhas@gmail.com

Eugenio Luo
University of Edinburgh
Edinburgh, United Kingdom
e.luo-1@sms.ed.ac.uk

Steven W. D. Chien
University of Edinburgh
Edinburgh, United Kingdom
steven.chien@ed.ac.uk

Michio Honda
University of Edinburgh
Edinburgh, United Kingdom
michio.honda@ed.ac.uk

Abstract

This paper presents SDP, a protocol design for emerging datacenter transports, such as NDP and Homa, to integrate data encryption. SDP enables a new design point of transport-level encryption that supports an existing NIC offloading designed for TLS over TCP, native transport protocol number alongside TCP and UDP, and message-based abstraction that enables low latency RPCs, various in-network compute, and host stack load balancing.

CCS Concepts

• **Security and privacy** → **Security protocols**; • **Networks** → **Transport protocols**.

ACM Reference Format:
Tianyi Gao, Xinshu Ma, Suhas Narreddy, Eugenio Luo, Steven W. D. Chien, and Michio Honda. 2025. Designing Transport-Level Encryption for Datacenter Networks. In *9th Asia-Pacific Workshop on Networking (APNET 2025), August 07–08, 2025, Shang Hai, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3735358.3735389

1 Introduction

Datacenter transport protocols, pioneered by DCTCP [4], have evolved over the last decade to achieve high throughput for bulk data transfer while maintaining low latency for small messages. The latest ones, including NDP [14] and Homa [21], are not extensions to TCP—they are message-based and often receiver-driven, which enable fine-grained network utilization.

However, what if the applications want data encryption to isolate themselves from other tenants and protect themselves from network infrastructure? Many cloud operators host multiple tenants [12]. Furthermore, even hyperscalers do not build every datacenter component on their own; they source many, such as switches, cables, and interface modules, from external vendors, which could be later found vulnerable or malicious. It is thus common for datacenter applications or tenants to seek encryption for their network data, just as they would over the Internet [33].

APNET 2025, Shang Hai, China

ACM ISBN 979-8-4007-1401-6/25/08
https://doi.org/10.1145/3735358.3735389

We present a secure datacenter transport protocol (SDP). SDP's design goal is to meet general datacenter transport requirements while assuming the same threat model with TLS/TCP, which is to protect the endpoints from data breach, packet injection, and replay attacks. We assume the host subsystem that executes the transport protocol, which is the OS kernel for our implementation, is trusted.

Adding encryption to datacenter transport is challenging, because it may sacrifice properties specifically required by datacenter transports. SDP preserves three properties crucial to protocol designers and datacenter operators. The first is message-based transport support that is essential to avoid head-of-line blocking (HoLB) and to utilize in-network compute (INC), which requires the network to identify boundaries of application-level messages [35]. The second is compatibility with existing hardware offload, particularly for cryptographic operations. Although hardware-based transports with custom NICs have been introduced by hyperscalers [34], smaller operators would perfer a protocol that is open and can be accelerated by commodity NICs. Last but not least is the introduction of a native transport protocol. Although deploying a new transport protocol without UDP encapsulation in the Internet is almost hopeless due to *ossified* middleboxes [18, 22], this is not the case in datacenters. New protocols, alongside TCP and UDP, would ease transport-specific network management and INC.

SDP supports unordered, arbitrary-length encrypted messages over an authenticated session, but with plaintext message identifiers and offsets in packets. This enables the network or the host stack to perform message-granularity operations, such as load balancing. SDP uses TLS offload and segmentation offload available in commodity NICs [25]. This means that SDP can be adopted without compromising hardware offload currently used by TLS/DCTCP. SDP enables message-level parallelism throughout the stack by per-message TLS record sequence number space in the authenticated session, yet guaranteeing the message uniqueness to protect the applications from replay attacks. This paper makes two main contributions:

(1) Identifying a design point for an encrypted message-based datacenter transport protocol that is protocol-number-agnostic, general for other datacenter transports like NDP, and compatible with existing TLS offload.
(2) A proof-of-concept implementation of SDT that extends the Linux kernel implementation of Homa [24]. It outperforms TLS/TCP by up to 35% in latency and up to 17% in throughput.

2 Design Space

Existing encrypted transports share problems with hardware offload or abstraction (reviewed from the top to the middle in Table 1 in § 2.1). On the other hand, existing message-oriented transports have performance issues or lack encryption (from the bottom to the middle in the table in § 2.2).

2.1 Transport-Level Encryption

TcpCrypt [5] was designed for Internet applications before the wide adoption of TLS. It encrypts the TCP payload using the key exchanged over connection setup. TcpCrypt is unsuitable for datacenters due to its stream abstraction that does not preserve application message boundaries and creates HoLB between messages [24, 35], as discussed further in § 2.2. Additionally, its cryptographic operations cannot be offloaded to commodity NICs.

QUIC mitigates HoLB using independent streams over UDP. However, its complex protocol design leads to high software overhead [37], and its cryptographic operations cannot be offloaded to commodity NICs. Further, its encrypted header fields, designed to prevent protocol ossification caused by middleboxes [17], are unsuitable for in-network compute [15, 31, 36, 40].

Accelerating TLS/(DC)TCP is more datacenter-friendly, as introduced by Facebook [16]. It enables opportunistic use of NIC offload for cryptographic operations, along with segmentation offload, by placing TLS in the kernel (so called kTLS). Netflix employs kTLS to handle their video streaming traffic [1], while Cisco/Cilium uses it for their network observability service [6, 10]. Similar to TcpCrypt, kTLS/TCP causes HoLB in RPC workloads, as discussed in § 2.2.

Falcon [9] is a datacenter transport that provides RDMA abstractions and encrypts the connection packets using the PSP UDP encapsulation protocol. While it preserves message boundaries and supports unordered *packet* delivery, it does not support unordered *message* delivery (i.e., ordered bytes in each message that can span across multiple packets). This is problematic for INC, as described next. Besides, Falcon requires custom NICs or Intel IPU for loss recovery and congestion control. Due to its abstraction and NIC requirements, Falcon is not a suitable basis for encrypted datacenter transport.

None of the encrypted transports mentioned above are compatible with INC [35] due not to encryption, but to the lack of message-boundary visibility to the network nodes. INC requires that network nodes identify boundaries of application-level messages with bounded resource usage, for example, for message-granularity load balancing. Some INC applications, such as data mutation, require the session key to be shared with the INC node, but these transports cannot support such applications even with the shared key.

2.2 Message-Based Transport

Datacenter transports require message-based abstractions [16, 24, 36] for RPCs (request-response) and INC support [35]. RPCs are typically small (e.g., half of RPCs have median requests and responses under 1530 B and 315 B, respectively [32]). TCP is unsuitable for RPCs because it disregards message boundaries that the application would have implied (e.g., separate `write` calls). Therefore, the application must embed a framing header indicating the message length before each message so that the receiver, which may read partial or multiple messages at once (e.g., a single `read` call) from the bytestream, can correctly reconstruct the messages.

However, these serialized messages in the bytestream cause HoLB. It is not only triggered by packet loss and retransmissions, but also by a CPU hotspot. Since the host stack processes packets that belong to the same 5-tuple on the same CPU core to avoid packet reordering in a TCP connection, a small message could wait for a preceding large one to be pushed to the network (egress) or delivered to the application (ingress). The application could increase RPC concurrency using multiple TCP connections, but a large number of those incur high processing overhead in both the kernel (e.g., cache pollution from connection metadata [13, 19]) and application (e.g., scanning many sockets [24, 42]). In addition, the concurrency level is limited by the port number space or 65 K per server port.

Message-based transport protocols address this space, as reviewed from the bottom to the middle of Table 1 in the rest of this section, to establish the starting point for SDP design.

KCM [2, 16] provides message-based abstractions through datagram socket APIs over TCP connections. However, it incurs high CPU overheads for locating the framing headers in the stream using application-supplied eBPF program and leaves HoLB on either packet losses or CPU hotspots unresolved.

Minion/μTCP [22] enables the TCP bytestream to self-delimit using consistent overhead byte stuffing (COBS) with a single delimiter byte, slightly increasing the data length. This allows the application to retrieve out-of-order, yet meaningful data or messages from the kernel buffer using socket API extensions. Although μTCP mitigates HoLB at the socket buffer, it comes at a significant computational cost to encode or decode the data with COBS.

Amazon SRD [34] is a hardware-based transport implemented in their custom NIC and provides RDMA-based abstraction. It eliminates priority flow control and employs packet spraying. Similar to Falcon, SRD does not expose message-level structures to the network, making INC infeasible.

Homa [21, 24] is a receiver-driven transport protocol that preserves message boundaries and mitigates HoLB from packet loss via message-granularity delivery. It also mitigate HoLB at a CPU hotspot using shortest remaining processing time scheduling, dynamically distributing messages, even those in the same 5-tuple, across the cores. Although Homa is a native transport protocol, its packet *overlays* a TCP header to utilize TCP Segmentation Offload (TSO), where the NIC splits a large segment (TSO segment) into MTU-sized packets.

Fig. 1 shows a generalized packet format based on Homa and MTP [36], which proposes a similar structure for INC. Packets within the same message share the same ID and length (Homa places these in the TCP Options space of the TSO segment, copied to all packets), while each packet specifies its offset within the message (Homa includes this in the framing header). The packet length in the network layer header or an additional field (not shown) can be used to compute the message portion length.

We believe Homa is a practical basis for a message-based transport protocol for datacenters, in terms of abstraction and packet format. Homa's host stack can be easily adapted to other message-based transports. For instance, NDP [14] shares similar stack and protocol requirements, such as packet scheduling for prioritizing specific data/control messages and first-RTT data transfer. NDP

	Encrypt.	Abstract.	NIC offload	Wire proto.	Host LB	In-net com.	
TcpCrypt[5]	Inline	Stream	N	TCP	Conn.	N	
QUIC[18]	TLS	Stream	N*	UDP	Conn.	N	*FPGA NIC attempt [41]
TLS/TCP[25]	TLS	Stream	Crypto+TSO	TCP	Conn.	N	
Falcon [9]	PSP	Ordered conns.*	Crypto+TSO**	UDP	Conn.	N	*RDMA verbs **Custom NIC or Intel IPU
SDP	TLS	Msg.	Crypto+TSO	New	Msg.	Y*	*With shared key for data mutation [35]
Homa[24]/NDP[14]	-	Msg.	TSO	New	Msg.	Y	
MTP[35]	-	Msg.	TSO	UDP	-	Y	
SRD[34]	-	Dgram.	Full*	Unknown	-	Y	*Custom NIC
KCM[2]/µTCP[22]	-	Msg.*	TSO	TCP	Conn.	Y*	*high overheads

Table 1: Key properties of encrypted or message-based transports.

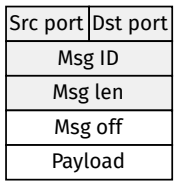

Figure 1: Generalized message-based transport packet format based on MTP [36] and Homa [24]. Shaded parts are identical between the packets that belong to the same message.

packet types map naturally to those of Homa: NACK in NDP and RESEND in Homa both request retransmission, while their PULL and GRANT request the next data. Furthermore, Homa is well documented and in active development.

2.3 Substrate for Offloading

We believe that, as long as the NIC offload requirement is met, enabling encrypted datacenter transport as a native transport protocol makes emerging transport design and deployment easier in datacenters. Although attempts have been made to repurpose the TCP protocol number for NIC offloading (e.g., STT [8]), it remains unclear whether this approach would gain widespread acceptance or offer sufficient generality, as it could confuse middleboxes (e.g., load balancers and network observability), network management or monitoring systems. To assess the feasibility of the new-protocol approach, we review the cryptographic NIC offload architectures.

Chelsio T6 released in 2016 supports TLS offload but strips TCP options provided by the stack, as it relies on the TCP full offload engine (TOE). It is thus unsuitable for not only new transport protocols but TCP extensions, a limitation noted by Netflix, Microsoft, and others [25].

In contrast, NVIDIA ConnectX-6 DX (CX6) and -7 (CX7), released in 2020 and 2023, respectively, feature a different hardware architecture known as *autonomous* offload [25]. This architecture allows the transport protocol to run in software, enabling it to evolve, while offloading data processing in an application-level protocol like TLS. Linux has acknowledged this architecture, and its software interface and hardware requirements for other vendors are documented [3]. These NICs are widely used today, with NVIDIA holding the largest NIC market share for NICs supporting 25 Gb/s and above (e.g., 65 % in 2019 [39]). Moreover, the distinctive software interfaces described in [3] allow us to infer the TLS offload architecture of other NICs based on their Linux drivers. Microsoft/-Fungible and Netronome NICs appear to support this architecture, while Intel and Broadcom do not.

We empirically confirmed that CX6 and CX7 NICs performs TLS offload correctly when the network layer header does not indicate TCP. This observation opens up the pathway to designing a new encrypted transport protocol that can benefit from *existing* hardware acceleration.

3 SDP Design Challenges

SDP focuses on message-based socket abstractions where the application sends multiple independent messages and the receiver can receive those in any order, while enabling reliable message delivery with retransmissions. SDP provides the security property over the abstraction based on Homa [24], because it provides datacenter friendly abstractions in terms of RPC and INC friendliness, host stack parallelism and generality to extend to other message-based datacenter transports (§ 2.2).

However, achieving those properties while adding security is challenging, due to the message-based transport features that differ from TCP and hardware features that assume TCP. Simply *stacking* TLS over a message-based transport like Homa is not viable. First, it precludes TLS offload, as enabling message-based abstractions with TSO requires that the transport layer place framing headers in the middle of the message (§ 2.2), whereas the NIC TLS offload cannot exclude such "gaps" from encryption. Second, transport-level *integration* of encryption is essential to prevent the application from replay attacks; we need to integrate the message identifier and TLS record sequence number management (§ 4.4).

Integrating encryption with a message-based transport is challenging due to the following stack or hardware features:

Unordered messages. Message-based transport (§ 2.2) means that the transport layer can send multiple independent messages in any order by the scheduler or congestion control algorithm even within the same flow 5 tuple. This is a stark contrast to TCP, which serializes all the transmissions, including retransmissions, to minimize packet reordering. TCP transmits packets in the syscall context (e.g., when a new data is written by the application and a sufficient window is available) or interrupt (softirq) context (e.g., when a received ack packet triggers transmission of new data in the send buffer), both of which are performed while locking the socket.

Message-level locking. However, message-based transport would take message-level locking without socket-level one for better message-level parallelism within the stack and flexible scheduling, as done in Homa. Further, receiver-driven transport protocols, such as NDP and Homa, run a dedicated packet scheduler thread that controls when the packet is dispatched to the network for fine-grained network utilization. For example, Homa sends small messages directly in the syscall context, but parts of large messages are pushed by the scheduler. When the Homa sender receives a *Grant* packet, in which the receiver grants the sender transmission

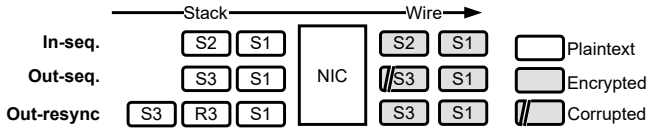

Figure 2: Encryption with autonomous offload [25]. Each rectangle represents one TLS record that contains one or more packets or TSO segments. The HW expects S2 after S1 to produce a correct *next* encrypted segment (In-seq); if S3 arrives, it generates a corrupted one (Out-seq.). A resync descriptor (R3) changes the seqno that HW expects to S3 (Out-resync).

of new data, it sends data chunks in the softirq context. Concurrency between syscall, scheduler, and softirq poses challenges.

Autonomous offload. The NIC based on Autonomous Offload (AO) maintains per-flow TLS contexts. Fig. 2 illustrates how AO works. When the NIC sees a packet and its metadata indicates TLS offload, it checks whether the packet is what it expects with the context; if it is not, the NIC generates a wrong TLS record header and authentication tag, leading to failure of decryption. When the software needs to send a packet that the NIC does not expect, it must put a *resync* descriptor before that packet in the queue (Fig. 2 bottom), to make the NIC expect that packet. In TCP, this feature is used for retransmissions where the NIC could see the previous record sequence numbers again.

4 SDP Design

Unlike TLS stacked on TCP, SDP *embraces* encryption based on the TLS record protocol in the transport protocol. This enables parallelism required by message-based transports (§ 4.3) while preserving the security properties of TLS/TCP (§ 4.4) and the use of existing TLS offload (§ 4.2 and § 4.5).

4.1 Session Initiation

SDP initiates a secure session using the standard TLS 1.3 handshake performed by the application, because datacenter transport protocols, such as Homa and NDP, send an RPC already on the first RTT without transport-level handshake. A session is identified by the source-destination 4 tuple.

After the handshake, the application registers the initialization vectors, session keys and record sequence numbers negotiated over the handshake to the SDP socket[1]. After that, a plaintext message written to the socket is encrypted and sent by SDP. The SDP receiver decrypts the message and the application reads the plaintext one.

Although the session initiation takes one RTT, the application can reuse the same shared key over multiple, concurrent messages in the session for a while.

4.2 Arbitrary-Sized Unordered Messages

SDP segments an application message in two stages—to TSO segments (§ 2.2) and packets. The receiver reassembles the packets in the reverse order.

[1]Same as kTLS: https://docs.kernel.org/networking/tls.html.

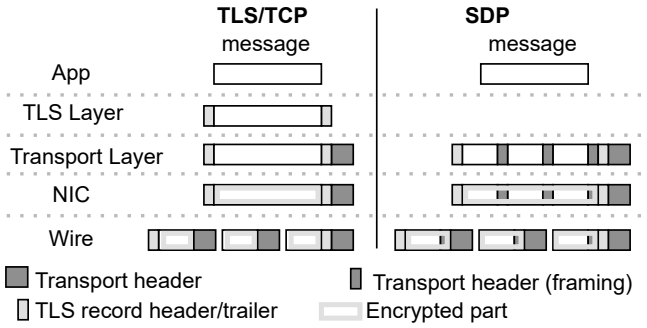

Figure 3: The send path of a single app message over TLS/TCP and SDP. TLS/TCP literally stacks TLS over TCP, whereas SDP integrates the TLS-based encryption inside the transport protocol.

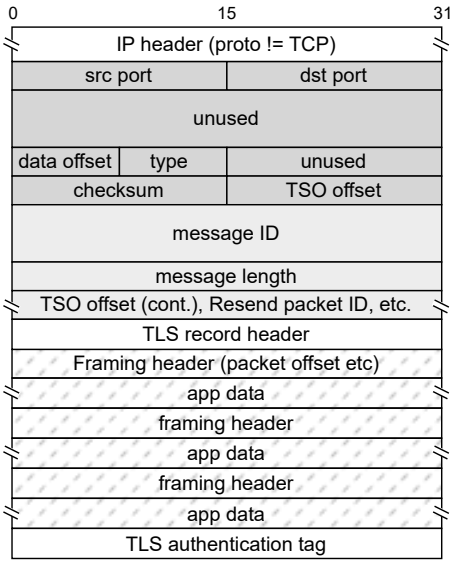

Figure 4: SDP TSO segment with one TLS record being split to 3 packets. Dark and light gray parts overlay TCP common header and options space, respectively, and are replicated over every packet by TSO. The NIC encrypts the dashed area. TLS record header is actually 13 B and the authentication tag is 16 B.

SDP creates TLS records, each of which is preceded by a record header and is at most 16 K B in size, to align with the boundaries of the TSO segments that are at most 65 K B.

As a simple example, Fig. 3 right illustrates how a message that consists of one TSO segment and TLS record splits into three packets, and Fig. 4 shows the detailed format of this segment. Each TSO segment has TSO offset, which indicates the position of the segment within the message. When the NIC does not support TLS offload, the records are encrypted by the CPU at this stage. SDP packets need IDs for the receiver to reassemble the TSO segment. We use the IPID in the network header because it is incremented over the packets generated by TSO.

The receiver first reassembles a TSO segment based on the packet IDs and TSO offset-message ID tuple. It then decrypts the TLS

records and reassembles the application message using the TSO offset in the TSO segments.

We must handle two cases of retransmissions: actual packet loss and spurious retransmission, which must be ignored by the receiver. Retransmission of a packet needs to have the original offset within the segment, we embed that offset in the unused space of "TCP" header (Resend packet ID in Fig. 4, plaintext area).

4.3 Message Parallelism

TLS records are boundary-preserved, but ordered based on sequence numbers in the stream. This means that, if a transport sent multiple messages in parallel in the same flow 5 tuple, which identifies the TLS session, they could cause HoLB, because they must be serialized into the sequence number space.

The single TLS record sequence number space is also problematic with the hardware offload. Consider two TSO segments that belong to different messages but to the same 5 tuple, S3 and S4. They are sent in parallel by different CPU cores and thus to different NIC queues. Although each segment prepends the resync descriptor (see Fig. 2), say, R3 and R4, it is not guaranteed that the pair of the resync and corresponding descriptors are read by the NIC atomically; the NIC could read R3 after R4 then read S4, resulting in an inconsistent authentication tag for S4.

SDP therefore uses per-message record sequence number space within the TLS session. In other words, the record sequence number increments within the message like regular TLS. Record sequence number spaces are mapped to the message IDs. When the receiver sees the first packet that belongs to an unseen message ID, it initializes the expecting sequence number with its record sequence number.

4.4 Replay Attack Protection

Although per-message record sequence number space enables message-level parallelism, this method alone enables replay attacks, where an attacker injects a previously-seen message to the receiver and the receiver accepts it (i.e., deliver to the application). In TLS/TCP, replay attacks are prevented by the receiver never accepting the duplicated TLS record sequence number.

SDP guarantees the exact-once semantics of messages with transport-level encryption. It maintains message IDs such that the same values are used only once in the lifetime of the secure session. Our current implementation, inheriting from Homa, uses a 64-bit identifier space.

4.5 TLS Hardware Offload

Per-message record sequence number space enables the use of NIC offload based on AO for encryption. Parallel messages can go to different queues without confusing the NIC's expectation. We create a flow context, which dictates the next TLS record to be seen, per message basis. Note that the NIC can maintain millions of active flow contexts and packet transmissions effectively hide the cache evictions and admission [25, 26].

In the scenario used in § 4.3, since those segments are associated with separate contexts, we do not have to manipulate the sequence number that the hardware expects. TSO segments in the same message always go to the same queue, because they are serialized based on message-level locking to avoid packet reordering (§ 3).

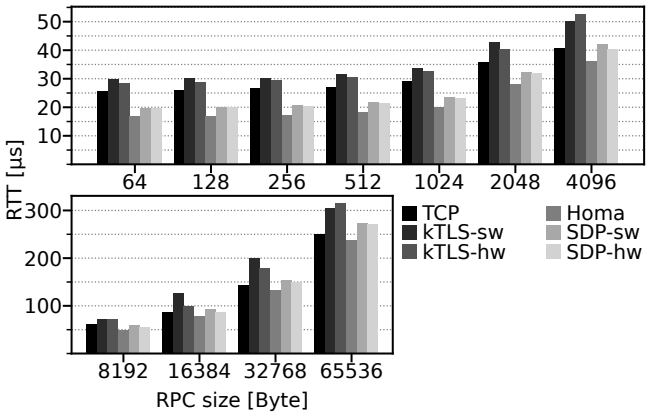

Figure 5: Unloaded RTTs of various sized RPCs. Standard deviations are 2–12%.

5 Evaluation

Implementation. Current SDP supports Linux kernel 6.2. It consists of approximately 2800 and 300 LoC patches to Homa/Linux and Nvidia `mlx5` driver, respectively.

HW&OS. We use two identical machines connected back-to-back. Each machine is equipped with two Xeon Silver 4314 CPUs and CX7 100 Gb/s NIC. Both machines enable Turbo Boost and disable Hyper-threading. They run Linux kernel 6.2. We configure RX interrupts to go to one CPU core based on Homa's recommendation, because Homa does its own load balancing without relying on RSS. All the threads and softirqs run on the same NUMA node as the NIC. The network MTU size is 1.5 K B.

Unloaded RTT. We first measure RTT of a single RPC without concurrent RPCs, using our custom app to highlight software overheads of the network stack, including the transport protocol, without the effect of queuing or app-level processing delays. Fig. 5 shows the results.

SDP outperforms kTLS by 13–32 % with TLS offload and 10–35 % without it. Since Homa is faster than TCP by 5–35 %, SDP does not diminish the advantage of Homa over TCP. The benefit of hardware offloading is small (up to 7 % in SDP) in this experiment, because CPU cores are idle; we see the benefit when those are loaded, as seen in Redis.

Redis. We added support for SDP in Redis. Because of SDP's native socket API, we were able to just add the SDP socket alongside TCP sockets that Redis monitors. This means that, the server instance or database can be shared between TLS/TCP (or just TCP) clients and SDP clients, which avoids disruption of existing operation.

Fig. 6 shows throughput over YCSB workloads[2]. The default value size is 1 K B, but to see the impact of it, we also test with 64 B and 4 K B values. To saturate the server, we use multiple threads and cores at the client, each opens its own socket to send requests to the server in parallel.

[2]We used YCSB-C [28] and extended it to support Homa and SDP.

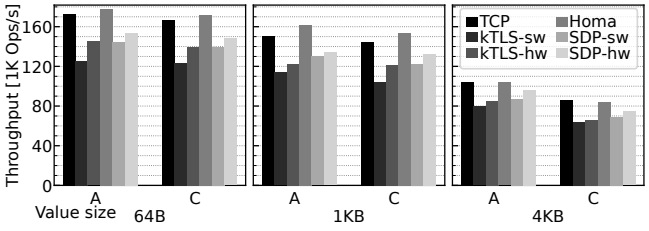

Figure 6: Redis throughput on YCSB A (Update-heavy) and C (Read-only) workloads.

SDP outperforms kTLS[3] by 5–13 % with TLS offload and by 8–17 % without it. Unlike the previous experiment, since Redis has app-level processing overheads (e.g., request parsing and database manipulation), we observed the improvement with TLS offload. TCP (without TLS) performs slightly better than Homa with 4 K B values, but SDP always outperforms kTLS, highlighting the better processing locality achieved by transport-level integration of encryption.

6 Discussion

Transport-level integrity. When TSO is used for a non-TCP protocol, the NIC does not embed a checksum in the overlaid TCP header field, meaning that checksum offload is impossible. Because of this, Homa does not guarantee message integrity and thus the application must compute the checksum and embed it in their messages. SDP intrinsically avoids this problem, because encrypting and decrypting the message ensures integrity is verified. Moreover, the cryptographic operations can be offloaded to the NIC hardware.

Encryption protocol. SDP would support PSP used in Falcon (§ 2.1) in addition to TLS, but the challenges of encrypting message-based transport protocols discussed in this paper remain valid unless the NIC explicitly supports the SDP packet format. This is because encryption would occur per TSO segment, which includes framing headers, while PSP uses a single encryption offset. We plan to explore using PSP instead of TLS once a NIC with PSP offload becomes available.

In-network compute compatibility. SDP supports INC enabled by MTP [35], unlike other encrypted transports discussed in § 2.1, because it allows the network to reconstruct application-level messages based on the plaintext message ID, TSO offset and message length, much like what the SDP receiver does (§ 4.2). Data mutation is also possible if the key is shared, because SDP has no dependencies between the messages, or its TLS record never spans across multiple messages. SDP is also compatible with packet trimming used by NDP [14], which can be seen as INC, because it leaves message ID and length unencrypted.

0-RTT data. Key exchange is critical for cloud applications to initiate data transfer as quickly as possible. While the default option relies on the standard TLS handshake and reuses the same shared key over a persistent source-destination 4- tuple (§ 4.1), a faster key exchange mechanism would be beneficial as clients and servers continuously join and leave. 0-RTT data, which sends an encrypted application message without prior handshake, could be possible at the expense of forward secrecy. It would use a *short-lived* server ECDH public key provided by the DNS server[4]. This could be feasible, because unlike the Internet, the datacenter or cloud provider often operate its own root CA that also acts as the internal DNS resolver. The client and server can use the 0-RTT data and its response to establish a forward-secure session.

[3]Native Redis uses user-space TLS that does not support TLS offload, but we added support for kTLS for fair comparison to SDP.

7 Related work

We discuss the remaining topics not discussed in § 2 here.

RDMA network security. ReDMArK [30] demonstrates packet injection attacks in RDMA networks, highlighting the need for encryption, such as IPSec or sRDMA [38], which mitigates these attacks by using symmetric cryptography and embedding MAC in the RDMA header.

Host stack enhancements. Host stack improvements, such as batching [13], zero copy [42], and fine-grained core allocation [7], complement SDP, though TCP-specific optimizations like congestion control [4] and handshake improvements [27] are not applicable. ByteDance has reported their effort of improving Homa for their RPC traffic [20], improving large send performance with pipelining, congestion control with better RTT measurement, loss detection, and buffer estimation to coexist with TCP traffic. Those techniques are directly applicable to SDP.

Transport multiplexing. Aquila [11] and EQDS [23] enable sharing of the same network fabric for all host traffic, including TCP and RDMA. EQDS operates as *edge functions*, scheduling traffic over UDP using an NDP-derived control loop, while Aquila uses a ToR-in-NIC (TiN) chip for hardware-based transport (GNet). SDP can be multiplexed within these systems, providing abstraction and encryption to the application, and can also be used between edge functions.

8 Conclusion

We explored a new design point of secure datacenter transport protocols that supports message abstractions crucial for both in-network and in-host load balancing, TLS offload available in commodity NICs, and native transport protocol, while preserving the security properties enabled by TLS/TCP. Our key takeaway is that it is essential to co-design encryption and transport protocol to enable those properties. We believe we made a substantial progress towards promoting alternative, secure transport protocols in datacenters.

Acknowledgments

We are grateful to the anonymous reviewers for valuable comments. We thank John Ousterhout for the discussion on Homa. We also thank Boris Pismenny for helping us understand TLS offload. This work was in part supported by EPSRC grant EP/V053418/1, Royal Society Research Grant, and gift from Google and NetApp.

References

[1] [n. d.]. ([n. d.]).
[2] [n. d.]. Kernel Connection Multiplexer. https://www.kernel.org/doc/Documentation/networking/kcm.txt. ([n. d.]).

[4]Inspired by Encrypted Client Hello [29].

[3] [n. d.]. Kernel TLS offload. https://www.kernel.org/doc/html/latest/networking/tls-offload.html. ([n. d.]).

[4] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM '10)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1851182.1851192

[5] Andrea Bittau, Michael Hamburg, Mark Handley, David Mazieres, and Dan Boneh. 2010. The Case for Ubiquitous Transport-Level Encryption. In *19th USENIX Security Symposium (USENIX Security 10)*.

[6] Daniel Borkmann and John Fastabend. [n. d.]. Combining kTLS and BPF for Introspection and Policy Enforcement. Linux Plumbers Conference 2018. ([n. d.]).

[7] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. 2022. Towards μ s tail latency and terabit ethernet: disaggregating the host network stack. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 767–779.

[8] Bruce Davie and Jesse Gross. 2014. *A Stateless Transport Tunneling Protocol for Network Virtualization (STT)*. Internet-Draft draft-davie-stt-05. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-davie-stt/05/ Work in Progress.

[9] Nandita Dukkipati, Neelesh Bansod, Chen Zhao, Yadong Li, Jay Bhat, Shiraz Saleem, and Anjali Singhai Jain. 2024. Falcon: A Reliable and Low Latency Hardware Transport. The Technical Conference on Linux Networking (Netdev 0x18), https://netdevconf.info/0x18/sessions/talk/introduction-to-falcon-reliable-transport.html. (2024).

[10] John Fastabend. [n. d.]. Seamless transparent encryption with BPF and Cilium. Linux Plumbers Conference 2019. ([n. d.]).

[11] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. 2022. Aquila: A unified, low-latency fabric for datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1249–1266. https://www.usenix.org/conference/nsdi22/presentation/gibson

[12] Google. 2022. Encryption in transit. https://cloud.google.com/docs/security/encryption-in-transit. (2022).

[13] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 135–148. http://dl.acm.org/citation.cfm?id=2387880.2387894

[14] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 29–42. https://doi.org/10.1145/3098822.3098825

[15] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. 2021. Prism: Proxies without the Pain. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 535–549. https://www.usenix.org/conference/nsdi21/presentation/hayakawa

[16] Tom Herbert. 2016. Data center networking stack. The Technical Conference on Linux Networking (Netdev 1.2), https://legacy.netdevconf.info/1.2/session.html?tom-herbert/. (2016).

[17] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. 2011. Is It Still Possible to Extend TCP?. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC '11)*. Association for Computing Machinery, New York, NY, USA, 181–194. https://doi.org/10.1145/2068816.2068834

[18] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. (May 2021). https://doi.org/10.17487/RFC9000

[19] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 489–502. http://dl.acm.org/citation.cfm?id=2616448.2616493

[20] Xiaochun Lu and Zijian Zhang. 2023. Leveraging Homa: Enhancing Datacenter RPC Transport Protocols. The Technical Conference on Linux Networking (Netdev 0x17), https://netdevconf.info/0x17/docs/netdev-0x17-paper36-talk-paper.pdf. (2023).

[21] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 221–235. https://doi.org/10.1145/3230543.3230564

[22] Michael F Nowlan, Nabin Tiwari, Janardhan Iyengar, Syed Obaid Amin, and Bryan Ford. 2012. Fitting Square Pegs Through Round Pipes: Unordered Delivery {Wire-Compatible} with {TCP} and {TLS}. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 383–398.

[23] Vladimir Olteanu, Haggai Eran, Dragos Dumitrescu, Adrian Popa, Cristi Baciu, Mark Silberstein, Georgios Nikolaidis, Mark Handley, and Costin Raiciu. 2022. An edge-queued datagram service for all datacenter traffic. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 761–777. https://www.usenix.org/conference/nsdi22/presentation/olteanu

[24] John Ousterhout. 2021. A Linux Kernel Implementation of the Homa Transport Protocol. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association. https://www.usenix.org/conference/atc21/presentation/ousterhout

[25] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. 2021. Autonomous NIC offloads. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 18–35.

[26] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. 2022. The benefits of general-purpose on-NIC memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1130–1147.

[27] Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. 2011. TCP fast open. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. 1–12.

[28] Jinglei Ren. [n. d.]. YCSB-C. https://github.com/basicthinker/YCSB-C. ([n. d.]).

[29] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. 2023. *TLS Encrypted Client Hello*. Internet-Draft draft-ietf-tls-esni-17. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-tls-esni/17/ Work in Progress.

[30] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. 2021. ReDMArk: Bypassing RDMA Security Mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 4277–4292. https://www.usenix.org/conference/usenixsecurity21/presentation/rothenberger

[31] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 785–808. https://www.usenix.org/conference/nsdi21/presentation/sapio

[32] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. 2023. A Cloud-Scale Characterization of Remote Procedure Calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 498–514.

[33] Amazon Web Services. 2024. Security Pillar: AWS Well-Architected Framework. https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/welcome.html. (2024).

[34] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. 2020. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE micro* 40, 6 (2020), 67–73.

[35] Brent E. Stephens, Darius Grassi, Hamidreza Almasi, Tao Ji, Balajee Vamanan, and Aditya Akella. 2021. TCP is Harmful to In-Network Computing: Designing a Message Transport Protocol (MTP). In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks (HotNets '21)*. Association for Computing Machinery, New York, NY, USA, 61–68. https://doi.org/10.1145/3484266.3487382

[36] Brent E Stephens, Darius Grassi, Hamidreza Almasi, Tao Ji, Balajee Vamanan, and Aditya Akella. 2021. TCP is Harmful to In-Network Computing: Designing a Message Transport Protocol (MTP). In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*. 61–68.

[37] Lizhuang Tan, Wei Su, Yanwen Liu, Xiaochuan Gao, and Wei Zhang. 2021. DC-QUIC: Flexible and Reliable Software-defined Data Center Transport. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 1–8. https://doi.org/10.1109/INFOCOMWKSHPS51825.2021.9484596

[38] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. 2020. {sRDMA}–Efficient {NIC-based} Authentication and Encryption for Remote Direct Memory Access. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 691–704.

[39] Mellanox Technologies. 2020. Mellanox Corporate Update—Unleashing the Power of Data. (2020).

[40] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. 2021. Concordia: Distributed Shared Memory with In-Network Cache Coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 277–292. https://www.usenix.org/conference/fast21/presentation/wang

[41] Xiangrui Yang, Lars Eggert, Jörg Ott, Steve Uhlig, Zhigang Sun, and Gianni Antichi. 2020. Making quic quicker with nic offload. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. 21–27.

[42] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 43–56. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata