Designing Transport-Level Encryption for Datacenter Networks

Tianyi Gao, Xinshu Ma, Suhas Narreddy, Eugenio Luo, Steven W. D. Chien, and Michio Honda
University of Edinburgh

***Abstract*—Cloud applications need network data encryption to isolate from other tenants and protect their data from potential eavesdroppers in the network infrastructure. This paper presents SMT, a protocol design for emerging datacenter transport protocols, such as NDP and Homa, to integrate data encryption. SMT integrates TLS-based encryption with a message-based transport protocol that supports efficient Remote Procedure Calls (RPCs), a common workload in datacenters. This architecture enables the use of per-message record sequence number spaces in a secure session, while ensuring unique message identities to prevent replay attacks. It also enables the use of existing NIC offloads designed for TLS over TCP, while being a native transport protocol alongside TCP and UDP. We implement SMT in the Linux kernel by extending Homa/Linux and improve RPC throughput by up to 41 % and latency by up to 35 % in comparison to TLS/TCP.**

1. Introduction

Datacenter transport protocols, pioneered by DCTCP [3], have evolved over the last decade to achieve high throughput for bulk transfer while maintaining low latency for small messages. The latest ones, including NDP [33] and Homa [59], are not extensions to TCP—they are message-based and employ clean-slate congestion control designs often with switch support to enable fine-grained network utilization. While most of these protocols have been implemented in user space or simulators, the availability of the Linux kernel implementation of Homa since 2021 [63], along with its demonstrated use in industry [53], makes widespread adoption of alternative transport protocols within reach.

However, what if the applications want data encryption to isolate themselves from other tenants and protect themselves from network infrastructure? There is widespread agreement that datacenter networks need encryption [19]. Major operators already adopt it, as seen in TLS deployment in Google [31] and Meta [55, 56] datacenters today. It protects users or operators from malicious insiders who may act as *man-in-the-middle* [57][1]. It is also essential in multi-tenant datacenters, where compromised tenant instances may attack other tenants or the shared network infrastructure [5, 39], which is often misconfigured [57, 73] or lacks timely security updates [1].

We present a secure message transport protocol (SMT). The design goal of SMT is to achieve performance-related properties of datacenter transports while supporting the same threat model as TLS/TCP—to protect the endpoints from data breach, packet injection, and replay attacks. SMT uses per-message TLS record sequence number space in the authenticated session, while guaranteeing message uniqueness to protect the applications from replay attacks. This design enables unordered encrypted messages while using existing TLS offload and segmentation offload available in commodity NICs [68]. This means that SMT can be adopted without compromising hardware offload currently used by TLS/TCP. SMT uses plaintext message identifiers and offsets in packet headers. This enables the network or the host stack to perform message-granularity operations, such as load balancing across multiple paths or CPU cores. SMT can be a native transport protocol without relying on TCP or UDP protocol number. This generalizes the design primitives of SMT, allowing them to be applied to secure other datacenter transports.

We have implemented SMT in the Linux kernel by extending Homa/Linux [63], because it provides a middle ground as an unencrypted but message-based datacenter transport protocol that could be transformed to another protocol like NDP (§ 2). This paper makes two main contributions:

- We identify a design point of an encrypted message-based datacenter transport protocol that is native and compatible with existing TLS offload while enabling the same security properties as TLS/TCP.
- We provide a proof-of-concept implementation of SMT that exhibits at most 41% higher throughput than TLS/TCP. We also report the application porting effort to use SMT through two applications: Redis key value store and NVMe-oF in-kernel storage subsystem.

2. Design Space

Datacenter applications exhibit Remote Procedure Call (RPC) workloads, where they send or receive structured messages in a request-response manner [27, 80]. RPCs are used for a range of purposes, including API calls between services [98, 52], access to in-memory key-value caches [51] or blob storage [66], and cluster management in microservice [99] or FaaS [18] platforms. RPCs are typically small (e.g., [81] reports that half of RPCs have median requests and responses under 1530 B and 315 B, respectively) and highly concurrent.

TCP is fundamentally unsuitable for RPCs because of two reasons. First, it disregards message boundaries that the application would have implied with separate `send` calls.

1. The incident indeed happened in 2013 and it accelerated adoption of traffic encryption [28].

	Encrypt.	Abstract.	Offload	Protocol	Parallelism	
TcpCrypt [11]	TcpCrypt	Stream	TSO	TCP	Conn.	
QUIC[40]	QUIC-TLS	Stream	N	UDP	Conn.	
TCPLS[77]	TLS	Stream	TSO	TCP	Conn.	
TLS/TCP[68]	TLS	Stream	Enc.+TSO	TCP	Conn.	
SMT	TLS	Msg.	Enc.+TSO	New	Msg.	
Homa[63]/NDP[33]	-	Msg.	TSO	New	Msg.	
MTP[43]	-	Msg.	N/A	New	Msg.	
Falcon [84]/UET [16]	PSP	Msg.	Full	UDP	Msg.	Custom NIC
SRD[82]	-	Msg.	Full	N/A	Msg.	Custom NIC
KCM[45]/μTCP[61]	-	Msg.	TSO	TCP	Conn.	

Table 1: Key properties of encrypted or message-based transport methods (discussed in § 2.1 and § 2.2).

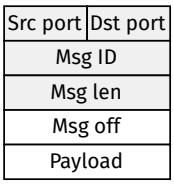

Figure 1: Generalized message-based transport packet format based on Homa [63] and MTP [43]. Shaded parts are identical between the packets that belong to the same message. Msg off identifies the position of this packet within the message.

Therefore, the application indicates the message length at the beginning of each message, so that the receiver, which may read partial or multiple messages at once from the bytestream, can reconstruct the original messages. Second, in-order bytestream abstraction causes head-of-line blocking (HoLB). It is not only triggered by packet loss or retransmission, but also on a CPU core. Since the host network stack parallelizes ingress and egress processing across the cores in a flow 5-tuple granularity to avoid packet reordering in a connection, a small message needs to wait for a preceding large one processed on the same core.

The application could increase message concurrency over parallel connections, but a large number of connections stress both the transport layer (e.g., cache pollution with connection metadata [32, 42, 6]) and application (e.g., per-socket syscalls [94, 63]). Also, parallel connections do not solve HoLB at a CPU core, once the number of those exceeds that of cores.

Despite those shortcomings, TCP is widely used in datacenters as a convenient reliable, congestion-controlled transport medium for RPC protocols, such as HTTP, gRPC and Thrift, also benefiting from NIC offloading for segmentation. TLS encryption is also common and thus all of those RPC protocols support it. Furthermore, container clusters often use encryption with mutual authentication (mTLS) to interconnect services over the service mesh [14].

Therefore, encrypted datacenter transport protocols must support RPC workloads efficiently. At the same time, it must retain the offload capabilities currently available in TCP and TLS, which is crucial for leaving sufficient CPU cycles for applications or improving energy efficiency, and threat model as existing encrypted communication. Support for those properties could facilitate departure from TLS/TCP.

In this section we explore the design space of such a transport protocol based on those requirements in either literature or new experiments where it is unknown. Three key observations that guide the design of SMT stand out:

- Existing encrypted transports fail to support hardware offload or message-based abstraction (§ 2.1).
- Homa provides a middle ground as an unencrypted message-based transport protocol for datacenters (§ 2.2).
- TLS offload available in commodity NICs can be generalized to new transport protocols (§ 2.3).

2.1. Transport-Level Encryption

Despite some momentum to integrate encryption into a transport protocol, since existing approaches have been designed for the Internet applications, they do not primarily focus on host-stack software overheads or HoLB at a CPU core, as reviewed from the top to the middle in Table 1 in the rest of this subsection.

TcpCrypt [10, 11], designed for the Internet before the wide adoption of TLS, extends TCP for connection authentication and data encryption. TcpCrypt encrypts TCP payload with AEAD using the key exchanged during connection setup. TcpCrypt is unsuitable for datacenters, because it inherits the HoLB problems in TCP and its cryptographic operations cannot be offloaded to commodity NICs.

QUIC is a transport protocol designed for the web. It runs in userspace on top of UDP and integrates a custom version of TLS 1.3 [92]. Although QUIC mitigates HoLB on a packet loss using multiple streams in the connection, it does not solve HoLB on a host CPU core due to connection-level core affinity. In addition, its complex protocol design incurs high software overheads [97, 88] and its cryptographic operations cannot be offloaded to today's commodity NICs.

TCPLS [77] provides similar features to QUIC, such as multiple streams, but over TCP to traverse more middleboxes. It extends the TLS 1.3 record type to aggregate and synchronize multiple TCP connections at the TLS endpoint. In addition to the inherent HoLB problems in TCP-based approaches (§ 2), TCPLS cannot utilize TLS offload due to its custom method of calculating the AEAD nonce [67].

kTLS [46] accelerates TLS/(DC)TCP by offloading encryption and decryption tasks to the kernel. It has been used by Facebook for datacenter networking [36], Netflix for video streaming [26] and Cisco/Cilium for network observability [24, 12]. It enables opportunistic NIC offload for cryptographic operations, along with segmentation offload. kTLS inherits the HoLB problems from TCP.

2.2. Message-Based Transport

Although existing transport-level encryption approaches are unsuitable for datacenter RPCs, there exist several attempts to enable message-based transport abstractions mostly without encryption. However, we must consider HoLB avoidance at both packet loss and CPU core, high-bandwidth and low-latency datacenter networking requirements, and generality. We review those attempts from the bottom to the middle of Table 1 in the rest of this subsection.

Kernel Connection Multiplexer (KCM) [45, 36] provides message-based abstractions through datagram socket APIs (e.g., `sendmsg/recvmsg`) over TCP connections. However, it incurs high CPU overheads for locating the framing headers in the stream using an eBPF program supplied by the application. It also leaves HoLB on packet loss or CPU core unresolved.

Minion/μTCP [61] enables TCP bytestream self-delimitation 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 a new socket API. μTCP mitigates HoLB caused by packet losses, but not the one on a CPU core. It also incurs high overheads to encode or decode the data with COBS.

SRD [82] is a transport implemented in a custom NIC. While performing in-NIC multipath congestion control, it delivers out-of-order packets to the software, which implements message abstraction. Falcon [84] and UET [16] are also hardware-based transports, but unlike SRD, they implement message abstraction, such as message-level reliability, in hardware. Those transports mainly focus on GPU-based HPC/AI workloads. For wider deployment scenarios, we seek an approach that is compatible with commodity NICs and can be used in bare metal or virtualized cloud instances and networks that currently use TLS/TCP.

Homa [63, 59] is a receiver-driven transport protocol that preserves message boundaries and mitigates HoLB on packet losses via out-of-order message delivery. It also mitigates HoLB on a CPU core using shortest remaining processing time (SRPT) scheduling, dynamically distributing messages across cores within the same flow 5-tuple instead of binding them to a fixed core.

Figure 1 depicts the simplified packet format of Homa; it also applies to MTP [43], another message-based transport designed for in-network compute (§ 7). To support arbitrary-sized, unordered messages, each packet contains message ID, message length and message offset, so that the receiver can reassemble the messages. Although Homa uses a new protocol number, its packet *overlays* a TCP header to utilize TCP Segmentation Offload (TSO), where a NIC splits a large segment (called TSO segment) into MTU-sized packets. Homa embeds the message ID in the TCP options space, which is copied to all the packets by TSO. It prepends the message offset to each packet payload, which is possible because the boundaries of packets generated by TSO are predictable. This is also necessary, as TSO does not write sequence numbers for undefined transport protocols [59, 62].

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 could be adapted to other message-based transports. For instance, NDP [33] shares similar stack and protocol requirements, such as packet scheduling for prioritizing specific data/control messages and first-RTT data transfer. NDP 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. Also, Homa is well documented and in active development for Linux upstreaming process [64].

2.3. Encryption Method

The choice of encryption method is crucial for designing a viable encrypted datacenter transport protocol. Of particular relevance is the deployment model and hardware offload.

IPSec provides host-to-host or site-to-site security as it operates at the network layer configured by the operator rather than the applications. This model thus differs from TLS whose authenticated sessions are established between individual applications. PSP [30], a more recent proposal for dataceneters, also performs packet-based encryption but in a more scalable way than IPSec, offering connection-granularity security. However, PSP needs specific NICs and it does not assume software-based encryption, because TLS is faster in software [47]. We wish to support various deployment models, much like the current TLS/TCP ones where TLS encryption can be decided by the application and its cryptographic operations can be optionally offloaded to the NIC when the NIC is trusted.

Furthermore, IPSec and PSP approaches are incompatible with confidential computing executed inside Trusted Execution Environment (TEE). TLS is used in such use cases [49, 91, 4]. Transport-level integration could be compatible with them, so long as the protocol is implementable in user-space.

Therefore, TLS appears the best option for smooth transition from TLS/TCP. However, a big question is whether it can be used for new, non-TCP transport protocols, such as Homa. We do not take this question just because Homa is a native transport protocol, but we believe enabling encrypted datacenter transport as a native transport makes emerging protocol design and deployment flexible. Although attempts have been made to repurpose the TCP protocol number for a non-TCP protocol to use NIC TSO (e.g., STT [20]), this approach would not gain widespread acceptance, as it complicates operation of network management or monitoring systems [5] and port number management in the host TCP implementation.

The key aspect of assessing the feasibility of using TLS with a new transport protocol is whether TLS offload in commodity NICs can be used, because hardware offloading is crucial for leaving as many CPU cycles as possible for applications or improving energy efficiency. We believe middleboxes in datacenters (e.g., load balancers) are more evolvable than those in the Internet, because they are made of software developed by the operator [23, 54] or service provider closely working with cloud operators, whereas home

gateways [35] and appliances in access networks [38, 22] are hard to enforce upgrade. NIC offload is also crucial to facilitate transition from TLS/TCP. If the operators or applications have to give up the hardware offload currently used for TLS/TCP, it would make the new transport slower than TLS/TCP accelerated by the offload. This would create a catch-22 situation, motivating no hardware vendor to support the acceleration of the new transport protocol.

We review two existing cryptographic NIC offload architectures with experimental validation.

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 [68].

In contrast, NVIDIA ConnectX-6 DX (CX6) and -7 (CX7), released in 2020 and 2023, respectively, feature a different hardware architecture, called *autonomous offload* [68]. This architecture allows the transport protocol to run in software, allowing it to evolve, while offloading data processing in an application-level protocol like TLS. Linux has supported this architecture, and its software interfaces and hardware requirements for other vendors are documented [46]. 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 [90]). The distinctive software interfaces described in [46] allow us to infer the TLS offload architecture of other NICs in their Linux drivers. Broadcom, Microsoft/Fungible and Netronome NICs appear to support this architecture, while Intel might not.

We tested CX6 and CX7 NICs by generating a TLS/TCP TSO segment using kTLS. In the driver, we modified the protocol number field in the IP header just before the packet descriptor was linked to the hardware. We confirmed that the resulting packets have correctly encrypted payload while preserving the original TCP header structure with or without TSO. This observation indicates feasibility of enabling a new encrypted transport protocol that can benefit from *existing* hardware acceleration.

3. SMT Design Challenges

SMT focuses on message-based socket abstractions where the application sends multiple independent messages in parallel and the receiver can process them in any order, while ensuring reliable message delivery through packet retransmissions. SMT provides TLS-based security guarantees for such an abstraction implemented by Homa [63], which achieves datacenter-friendly properties of RPC efficiency, host stack parallelism, and generality to extend to other message-based datacenter transports (§ 2.2).

Achieving those datacenter transport properties while adding security is challenging due to the TLS protocol semantics and stack and NIC features.

3.1. TLS Protocol Semantics

TLS assumes in-order bytestream abstraction for the underlying transport and guarantees the original order of the records, rejecting out-of-order or duplicated records, which would have been tampered or replayed but delivered by TCP due to TCP-level correctness based on sequence number and checksum. This means that simply *stacking* TLS over a message-based transport like Homa is not viable, because out-of-order message delivery to the TLS endpoint causes record rejection, whereas performing a TLS handshake for every message is impractical.

Stacking TLS over a message-based transport also precludes TSO performed together with TLS offload. 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. If those framing headers were encrypted, the transport protocol could not reassemble the TLS records from packets.

3.2. Host Stack and NIC Features

Message-based transport (§ 2.2) could send multiple independent messages in any order by the scheduler or congestion control algorithm 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 (e.g., when a new data is written by the application and the 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.

However, message-based transport would take message-level locking without socket-level one for message-level parallelism within the stack, as done in Homa. Further, receiver-driven transport protocols, such as NDP and Homa, run a dedicated packet scheduler thread 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 of new data, it sends data chunks in the softirq context.

Those stack features pose challenges in using TLS offload in the NIC. Autonomous Offload (AO) (§ 2.3) maintains a flow context backed by in-NIC memory, which stores the encryption key and self-incrementing record sequence number. Figure 2 illustrates how AO works. When the software sends a segment that the NIC needs to encrypt with a different record sequence number than its current internal one, it must prepend a *resync* descriptor in the queue (Figure 2 bottom) to adjust that internal one. TCP uses this feature for retransmissions where the NIC sees the previous record sequence numbers.

Message-based transports could send multiple messages across different CPU cores, which push their packets to different NIC queues. This makes enforcing the NIC to encrypt a record with a specific or predictable record sequence number

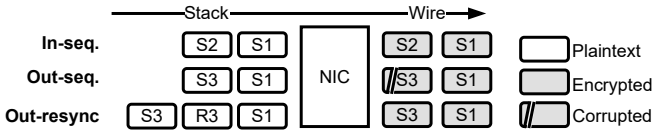

Figure 2: Encryption with autonomous offload [68]. Each rectangle represents one TLS record that contains one or more packets or TSO segments. The HW expects S2 after S1 to produce a correct *next* encrypted segment (In-seq); if S3 arrives, it generates a corrupted one (Out-seq.). A resync descriptor (R3) changes the seqno that HW expects to S3 (Out-resync). Note that each segment in the Wire actually consists of multiple packets split by TSO.

hard. Prepending a resync descriptor to every segment (to reset the sequence number expectation from another message) does not solve the problem, because the NIC provides no atomicity or ordering guarantee in reading the descriptors across the queues. Consider two segments that belong to different messages but to the same 5 tuple, S4 and S5 (not illustrated). They are sent in parallel by different CPU cores (e.g., scheduler and softirq) and thus to different NIC queues. Although each segment prepends a resync descriptor (see Figure 2), R4 or R5, it is not guaranteed that the descriptor pair of resync and segment (e.g., R4 and S4) are read by the NIC atomically; the NIC could read R4 after R5 then read S5, resulting in incorrect encryption.

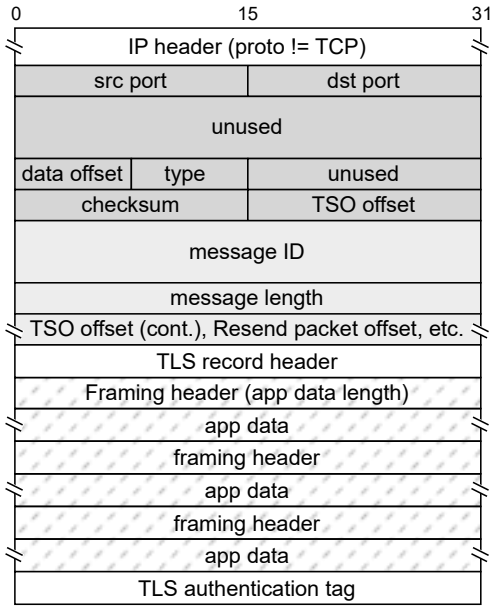

Figure 3: SMT TSO segment with one TLS record being split to 3 packets. Dark and light gray parts overlay TCP common header and options space, respectively, and are replicated over every packet by TSO. The NIC encrypts the dashed area. TLS record header is actually 5 B and the authentication tag is 16 B.

4. SMT Design

SMT addresses the aforementioned challenges by transport-level encryption, where the transport protocol *embraces* encryption based on TLS. This architecture enables two key features of SMT: message format that can use both TSO and TLS offload (§ 4.3) and the use of per-message record sequence number space in the secure session for unordered message delivery without costly per-message handshake (§ 4.4). We provide detailed security analysis in § 6.

4.1. Threat Model

We assume the same threat model as TLS/TCP, protecting endpoints from data breaches, packet injection, and replay attacks. We assume the host subsystem that executes the transport protocol— the OS kernel in our implementation— is trusted. When the OS kernel cannot be trusted, SMT can be implemented in user-space protected by a TEE environment— using a trusted network stack like rkt-io [91]. While we also assume the NIC is trusted, this assumption can be removed; in such cases, TLS offload must be disabled so that the NIC processes only encrypted packets.

4.2. Session Initiation

SMT 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 flow 5 tuple that consists of source-destination address and ports plus protocol. Since the handshake process is based on TLS 1.3, it can support mutual authentication as with mTLS [15].

After the handshake, the application registers the initialization vectors and session keys negotiated over the handshake to the SMT socket[2]. After that, a plaintext message written to the socket is encrypted and sent by SMT. The SMT 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. Alternatively, it can also be done over the first RTT SMT data at the expense of forward secrecy of that data (not subsequent ones), which we discuss in § 4.5.

4.3. Offload-Friendly Encrypted Message Format

SMT uses the TCP header structure with a new transport protocol number indicated in the network layer header to use TSO, like Homa [63]; we confirmed that it can also be compatible with TLS offload in § 2.3. When the application posts a message to an SMT socket, the question is how to segment this message, which can span across multiple

2. Same as kTLS: https://docs.kernel.org/networking/tls.html.

TSO segments or TLS records, into packets while applying encryption, possibly by the NIC. It must be done such that the receiver can reassemble the original message from them. This is not a concern for TLS over TCP, because, when a TLS receiver sees a series of records, they are already in-order based on the underlying bytestream abstraction. However, message-based transports with TSO and TLS offload place unusual demands, as discussed next.

SMT segments an application message in two stages—to TSO segments (§ 2.2) and packets. The receiver reassembles the message in the reverse order. SMT 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, which are at most 65 K B. When the NIC does not support TLS offload, the records are encrypted by the CPU at this stage.

Each TSO segment has *TSO offset*, which indicates its position within the message. Since SMT embeds it in the overlayed TCP header, which is copied to all the packets by TSO, all the packets that have been generated from the same TSO segment have the same TSO offset value. SMT needs *packet offsets* for the receiver to reassemble the TSO segment from the packets. We use the IPID in the network header because it is incremented over the packets generated by TSO. If the NIC generates TCP sequence numbers for non-TCP packets when performing TSO, we could use that, as it also works for IPv6.

As a simple example, Figure 3 illustrates a message that consists of one TSO segment and TLS record splits into three packets.

The receiver first reassembles a TSO segment based on the packet offsets in the tuple of TSO offset and message ID. It then decrypts the TLS records and reassembles the message based on the TSO offset values.

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 packet offset within the segment, we embed that value in the unused space of the overlayed TCP header (Resend packet offset in Figure 3, plaintext area).

Note that the use of framing headers is based on our current implementation. We could remove it, because the receiver can reassemble the TSO segments using solely packet offsets. This would improve performance of large messages because of simpler buffer operations.

4.4. Per-Message Record Sequence Number Space

To avoid costly handshake performed for every message or record rejection caused by out-of-order message delivery (§ 3.1), SMT uses per-message record sequence number space within the TLS session. Each record sequence number space offers the order-preserving guarantee of TLS on top of reliable message delivery of message-based transports like Homa and NDP. The record sequence number monotonically increments in 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 next in-sequence record sequence number.

However, this parallelism introduces challenges for TLS which is designed for TCP abstraction. TLS requires a unique record sequence number for each record in one handshake to prevent replay attacks. However, using multiple (i.e., per-message) record sequence number spaces itself means that the receiver may see the same record sequence number between the messages.

4.4.1. Message Uniqueness Guarantee. Transport-level encryption, instead of stacking TLS on a non-encrypted transport protocol, enables solving this issue. SMT introduces a composite 64-bit record sequence number that integrates a message ID whose uniqueness is guaranteed throughout the secure session with an intra-message record index.

To comply with TLS, TLS record sequence number with fixed 64 bits length is the only free variable available to encode both message ID and intra-message record index. We dedicate a portion of these bits to message IDs and assign the remaining bits to indexes of the records within the message (intra-message record index) as illustrated in Figure 4, It requires bit allocation trade-off between maximum message sizes and the number of unique message IDs. Since the maximum record size is 16 K B, supporting larger maximum message sizes needs more bits for the record indexes; supporting more messages needs more bits for the message IDs. Figure 5 plots this trade-off.

In our current implementation, we opt for 48-bit message IDs. This leaves 16 bits for the intra-message record index. This allocation allows a single message to accommodate up to 65K individual TLS records, supporting message sizes up to approximately 98 MB even with 1.5 K B (small) TLS records, and approximately 1 GB with 16 K B one (maximum record size). For reference, the default maximum message size of Homa is 1 MB. Each endpoint can use different message ID length as long as the receiver endpoint knows what the sender uses. That could be negotiated during the handshake. Revealing the message ID length to an eavesdropper does not increase the security risk.

The composition itself introduces very little performance overhead, because the intra-message record index occupies the lower bits, allowing the hardware's self-incrementing counter to operate correctly just like TLS/TCP.

4.4.2. TLS Hardware Offload. Per-message record sequence number spaces enable the use of AO-based TLS offload, avoiding the problem with non-atomic reads between the descriptors across the queues (§ 3.2), because the messages in the same 5 tuple do not have to share the flow context, which dictates in-sequence record sequence numbers, across the queues. This approach also enables efficient use of in-NIC memory, because it allows a flow context to be reused by another message in the same session (i.e., record sequence number space) simply performing a resync operation. This is not the case when switching the keys (e.g., with another handshake); it requires allocation of

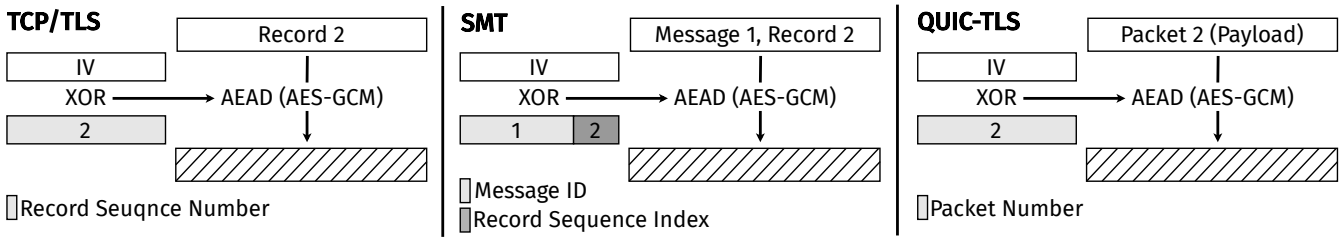

Figure 4: Use of record sequence numbers across TCP/TLS, SMT, and QUIC-TLS [92]: TCP/TLS uses the 64-bit record sequence number; SMT encodes message ID and intra-message record index (§ 4.4.1); QUIC uses the packet number (§ 6.3).

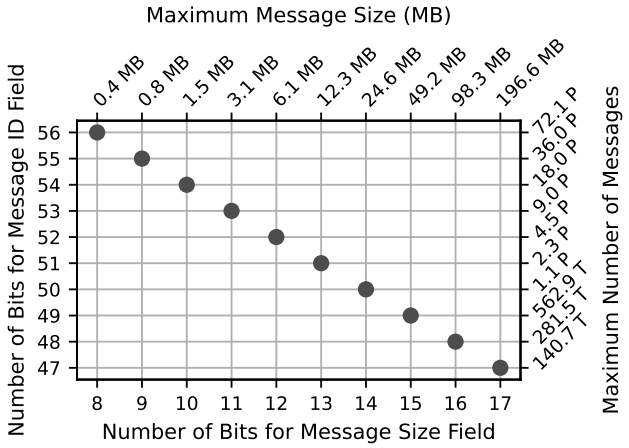

Figure 5: Trade-off between maximum message size and number of unique message IDs based on bit allocation in the 64-bit composite record sequence number.

a new flow context, which is more expensive than resyncing existing one.

Although the NIC can maintain millions of active flow contexts in its memory and packet transmissions effectively hide the cache evictions and admissions [68, 69], it cannot maintain an unlimited number of them. Therefore, taking the advantage of efficient reuse of the context, we design a flexible trade-off between parallelism and in-NIC memory usage. We create flow contexts for each message until a certain threshold, at least one per NIC queue. Messages that go to the same queue in the same flow 5 tuple may share the same context, but those sent to different queues do not. However, even when sharing the context, since they are serialized in the queue, a resync operation guarantees its target segment or record. Note that the segments that constitute the same message always go to the same queue, because the message-based transport could ensure in-order delivery *within* the message to avoid packet-level reordering.

Our current implementation allocates one flow context per queue for each flow 5 tuple, but we may revise this in the future or for other NICs.

4.5. Key Exchange and 0-RTT Data

Efficient key exchange is essential for cloud applications to initiate data transfer with minimal latency. While TLS 1.3, the default key exchange method for SMT, supports fast resumption and SMT can send multiple messages in the same session (§ 4.2), its effectiveness decreases in dynamic communication patterns where endpoint churn limits session reuse. To understand the overhead of key exchange, we take the latency breakdown of the TLS 1.3 initial handshake by timestamping the `picotls` library (Table 2). To accelerate key exchange, we first introduce three techniques that help reduce handshake latency (§ 4.5.1). We then show how to eliminate 1-RTT by pre-distributing long-term public key shares to internal DNS resolvers in the datacenter (§ 4.5.2).

4.5.1. Key Management and Authentication.

Key pre-generation. To reduce costs in S2.1 and C1.1, servers and clients could maintain a list of standby key pairs created prior to a handshake [96]. This is feasible in datacenters that centralize administrative control such that a choice of security parameters is made upfront.

ECDSA authentication. ECDSA significantly reduces handshake latency—by hundreds of μs in S2.5, C4.1, and C4.2—particularly in mutual authentication, where the server performs one Sign and two Verify operations.

Short certificate chain. To reduce latency in C3.2, we could use a short certificate chain and configure all endpoints with the CA's verification key, avoiding certificate lookup and long-chain validation. This speeds up the Verify Cert operation by approximately 52 % in our tests. Since an internal CA manages certificates within the datacenter, backward compatibility features can also be omitted.

4.5.2. 0-RTT Data and Key Exchange. Datacenter transport protocols, such as Homa and NDP, send an RPC already on the first RTT without transport-level handshake. 0-RTT data could be achieved by extending the TLS 1.3 key exchange with DNS-based distribution of a server's long-term Diffie-Hellman (DH) public key. This approach, inspired by TLS Encrypted Client Hello (ECH) [76], removes one RTT from the initial handshake.

In our design, the client first performs a DNS query to retrieve *SMT-ticket*, which includes: (i) the server's long-term ECDH public key share, (ii) its certificate, and (iii) a signature over the *SMT-ticket*s signed by the certificate's

Server	ID	Operation	Overhead (µs)
Handle CHLO	S1	Process CHLO	1.8
Generate SHLO	S2.1	Key Gen	67.9
	S2.2	ECDH Exchange	265.0
	S2.3	SHLO Gen	75.2
	S2.4	EE & Cert Encode	13.6
	S2.5	CertVerify Gen	137.6* / 1344.0+
	S2.6	Secret Derive	48.6
Handle Finished	S3	Process Finished	44.4
Client			
Generate CHLO	C1.1	Key Gen	61.3
	C1.2	Others Gen	5.5
Handle SHLO	C2.1	Process SHLO	2.6
	C2.2	ECDH Exchange	88.7
	C2.3	Secret Derive	48.8
Verify Cert	C3.1	Decode Cert	0.1
	C3.2	Verify Cert	483.4
Verify CertVerify	C4.1	Build Sign Data	1.4
	C4.2	Verify CertVerify	196.3* / 67.1+
Handle Finished	C5	Process Finished	42.6

Table 2: Server- and client-side TLS handshake overheads ($*$ with 256-bit ECDSA and $+$ with 2048-bit RSA).

private key. Note that the datacenter or cloud provider could operate its own root CA that also acts as the internal DNS resolver. With trusted CA public key pre-installed across the datacenter, the client can verify the *SMT-ticket* and send a ClientHello with its ephemeral key. These steps can occur before the handshake begins, as server information is often known in advance.

Using the server's long-term key and its own ephemeral key, the client derives an *SMT-key* and immediately sends encrypted application data. If forward secrecy is enabled, the server replies with a ServerHello containing its ephemeral key, enabling both sides to derive an *fs-key* and switch to forward-secret encryption. If forward secrecy is disabled, the *SMT-key* is used for all payload encryption for the duration of the session.

We retain TLS 1.3's session resumption mechanism, which updates cryptographic keys and thus resets the message ID space.

4.5.3. Forward Secrecy. The 0-RTT handshake trades some forward secrecy for lower latency, as client 0-RTT data is encrypted using the *SMT-key*, which lacks strong forward secrecy. To mitigate the risk, we limit *SMT-ticket* validity period. Following industry practice—such as Cloudflare's hourly rotation of session ticket keys for 0-RTT data [93]—we recommend a maximum lifetime of one hour. To further reduce replay risk caused by *SMT-key*, servers can record the CHLO random value, as specified in TLS 1.3 [75].

4.6. Implementation

The current SMT implementation[3] consists of a 2800 LoC patch to the Homa/Linux kernel module and a 300 LoC patch to the NVIDIA `mlx5` driver, requiring only these two kernel modules to be recompiled and reloaded.

Note that the device driver modification is to adjust the offset to start the encryption specifically for TLS offload and generalize flow context management that currently relies on TCP sequence number and lacks sufficient flexibility of allocation. We therefore believe its adoption once Homa is upstreamed [64].

To implement the key exchange method in § 4.5, we extend `picotls` with a new extension, to indicate the use of SMT-ticket, reusing the pre_shared_key field to specify its identity in the handshake.

5. Evaluation

We measure the performance of SMT in comparison to TLS/TCP and other systems.

HW&OS. We use two identical machines connected back-to-back. Each machine is equipped with two Intel Xeon Silver 4314 CPUs and NVIDIA/Mellanox ConnectX-7 100 Gb/s NIC. They install Linux kernel 6.2. We use one NUMA node, and separate cores for softirq contexts and application threads. The network MTU size is 1.5 K B unless otherwise stated. All experiments use AES-128-GCM (128-bit length key) cipher for both SMT and TLS. We don't use receive-side offload for kTLS, because not only SMT does not support it (§ 7), but it often impractical due to incompatibility with tunnelling protocols or packet delivery delay when the NIC waits for the complete record.

Performance metric. Our primary performance metric is the protocol and encryption overhead added to the base unencrypted variant (i.e., Homa), which we compare with that of TLS over TCP. This makes our measurements worthwhile, although Homa variants often do not perform better than the TCP variants due to immaturity of Homa itself, because those characteristics could still be valid even after the Homa implementation evolves. Later in this section, we also provide a snapshot of the performance of Homa and TCP variants with other aspects of SMT evaluation, which include application porting effort.

5.1. Unloaded RTT

We first measure RTT of a single RPC without concurrent RPCs, using our custom application to highlight software overheads of the network stack, including the transport protocol, without the effect of queuing or application-level processing delays. Figure 6 shows the results. We ran three trials, 8 seconds each, and plot the middle one in average latency; same for the next experiment.

SMT outperforms kTLS by 13–32 % with TLS offload and 10–35 % without it. Since Homa is faster than TCP by

3. https://github.com/uoenoplab/smt.

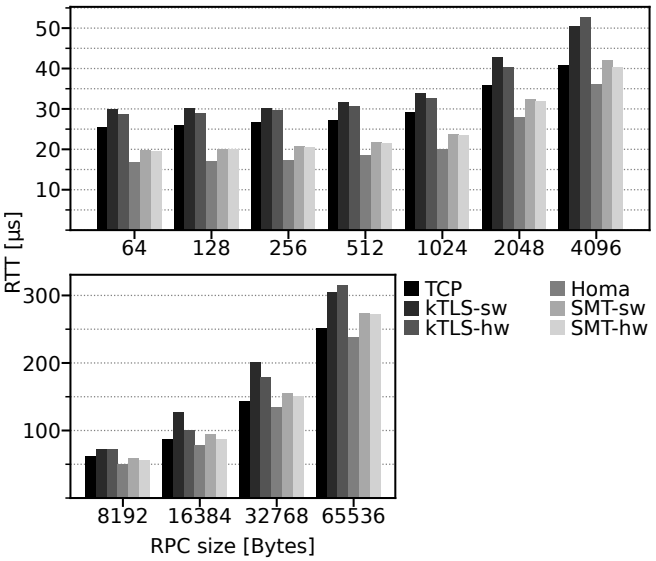

Figure 6: Unloaded RTTs of various sized RPCs. Standard deviations are 2–12 %.

5–35 %, SMT does not diminish the advantage of Homa over TCP. The margin is smallest with 65 K B RPCs, because the Homa receiver waits for the arrival of the entire RPC (that consists of multiple packets) before delivering (copying) data to the application, whereas TCP overlaps packet reception and application-data delivery due to its streaming abstraction. This is not a fundamental limitation of Homa and there exists work to remedy this issue [53, 65], and once it is merged into Homa, we expect that Homa and SMT outperform TCP variants by larger margins.

The benefit of hardware offloading is small (up to 7 % in SMT) in this experiment. For small messages, this is due to low encryption overheads and per-segment cost incurred to populate offloading metadata (§ 3). For large messages, the bottleneck is not encryption but data copy. To confirm the similar characteristics of larger message, we experimented with 500 K B RPCs (not plotted); it exhibited little (1 %) latency benefit of hardware offloading. In the next experiments where CPU cores are more loaded due to concurrent RPCs (§ 5.2) or complex application-level processing (§ 5.3), we observe a larger benefit of hardware offloading.

5.2. Throughput

Next, we measure the performance of SMT in the presence of concurrent RPCs and multiple application threads. We generate concurrent RPCs using 12 threads allocated for the applications and 4 threads for the stack at each of the server and client. Recent report [81] shows 90 % of RPCs in production are smaller than 10 K B. We thus evaluate three representative sub-10 K B sizes—small, near-MTU, and multi-MTUs.

Figure 7 shows the throughput over different numbers of concurrent RPCs for three RPC sizes. For 64 B messages, SMT exhibits higher throughput than kTLS by 16–40 % with TLS offload and 16–40 % without it; those improvements with 1 K B messages are 17–41 % and 16–39 %, respectively.

SMT exhibits lower throughput than kTLS with 8 K B messages, by 5–15 % with TLS offload and 3–13 % without it, because, as before, Homa is unoptimized yet for large messages in comparison to TCP.

In SMT, advantage of HW is largest with 1KB cases (5–11 %), because 8 K B cases (4–9 % improvement), throughput are constrained by the lack of pipelining.

In 64 B RPCs with 50–100 concurrent requests, SMT exhibits a slightly larger benefit of HW than kTLS/TCP due to lower protocol overheads of the base transport (Homa) than TCP (as seen in § 5.1) and thus higher relative crypto overheads.

Impact of a larger MTU. We ran the same tests as Figure 7 right (50–150 concurrent 8KB RPCs) with 9 K B MTU (thus one message fits into a single packet). Compared to 1.5 K B MTU cases, SMT exhibited 13–28% and 16–31% higher throughput with and without TLS offload, respectively, because of the reduced number of packets per message.

CPU usage. We tested CPU usage with the setting of Figure 7 middle, but limiting the RPC rate of all the systems to 1.2Mreq/s to measure the resource usage over the same request rate. SMT-SW exhibited 3.5 % lower CPU usage than kTLS-SW at the client and 10.5 % at the server. SMT-HW exhibited 2 % lower CPU usage than kTLS-HW at the client and 8 % at the server. SMT-HW reduced the CPU usage of SMT-SW by 4 % at the server and 1.5 % at the client. Our current implementation does not show memory saving with hardware offload, because its software encryption is done in-place.

5.3. Redis

What does using SMT in a real-world application look like? We report our experience of adding support for SMT in Redis, a widely used key-value store. The vast majority of effort was supporting vanilla Homa; once it is done, support for SMT was trivial because of transport-level encryption; it simply adds `setsockopt` to register the key (§ 4). We thus mainly discuss adding support for Homa in Redis.

Redis adopts a single-threaded design and monitors clients using an `epoll` event loop; each client connects to the server over a TCP connection and the connection is reused over multiple requests. Since a Homa socket can communicate with multiple clients, Redis/Homa could directly block on `recvmsg` syscall. However, to share the same database between both TCP and Homa clients, we register the SMT socket (that handles all the SMT clients) to the `epoll` socket. When a request arrives over TCP, the Redis server reassembles messages by locating the Redis headers in the bytestream; when that arrives over Homa, since Homa preserves message boundaries, Redis/Homa does not need to maintain the partial read offset.

Our modification in Redis is straightforward, because Homa and SMT provide true file descriptors and the Redis instance can monitor both TCP and SMT clients in the

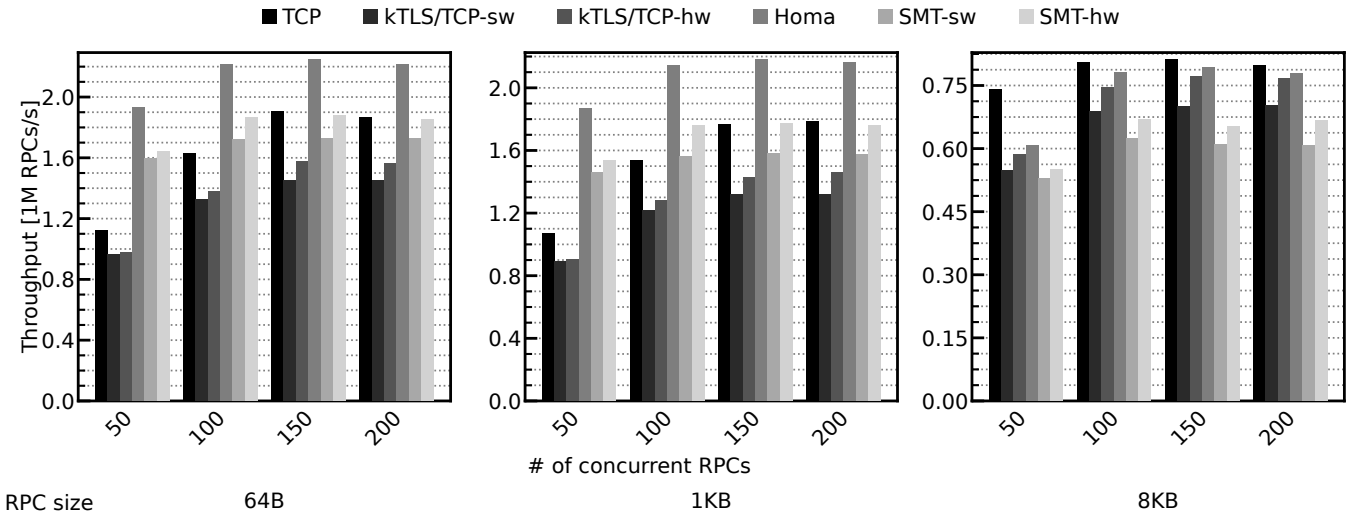

Figure 7: Concurrent RPC throughput.

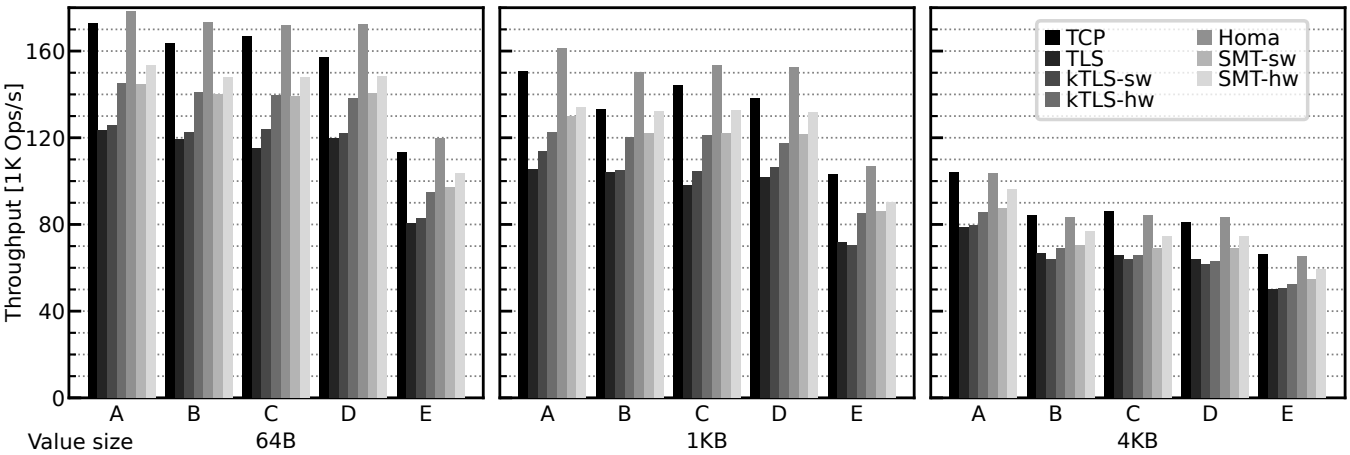

Figure 8: Redis throughput on YCSB A (update-heavy), B (read-mostly), C (read-only) and D (read-latest) workloads.

original `epoll` event loop. This is not the case for supporting a kernel-bypass TCP stack as done in mTCP [60], Demikernel [95] and Paste [37], which need to replace the whole event loop, disallowing the clients to access the same database over the regular kernel TCP stack. The same goes for TCPLS; its I/O descriptors cannot be registered to the OS-driven event loop.

Results. Figure 8 shows throughput measured by a YCSB [17, 74], which emulates real-world key-value store workloads. We added support for Homa and SMT to it. Its default value size is 1 K B, but to see the impact of sizes, we also test with smaller (64 B) or larger (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.

We compare SMT with or without TLS offload against TCP and TLS/TCP. Redis uses user-space TLS that does not support hardware offload, but we added support for kTLS to make a fair comparison to SMT.

SMT outperforms Redis/TLS in all the workloads and value sizes. SMT without TLS offload outperforms user-space TLS by 5–24% and kTLS without offload by 8–22%. When TLS offload is enabled, SMT outperforms kTLS by 5–18%. Recall that the throughput of Homa and SMT was constrained to around 700 K RPCc/s by the softirq thread in § 5.2 (Figure 7). Since Redis has a considerable amount of application-level processing overheads (e.g., request parsing and database manipulation), the overall rates are below that rate, and thus Homa and SMT always outperformed the TCP counterparts.

In 64 B RPCs, since the application-level processing and data encryption/transmission happen in the same thread that becomes the bottleneck, CPU cycles freed up by encryption offload directly improved performance, whereas at higher request rates with SMT, the relative cost of the receive path of the server's stack became higher and thus we saw a smaller benefit of encryption offload.

TCP (without TLS) performs slightly better than Homa with 4 K B items, because it is optimized for large transfers. However, SMT, even without TLS offload, always outperforms TLS. Similar to 8 K B RPC cases in Figure 7, this

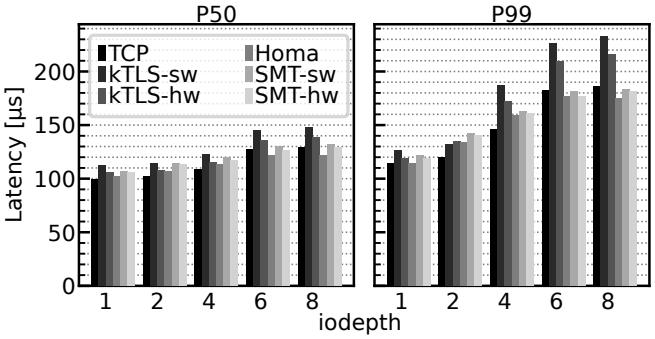

Figure 9: P50 and P99 latency of NVMe-oF.

highlights the better processing locality achieved by transport-level integration of cryptographic operations.

5.4. In-Kernel Client: NVMe-oF

To show the applicability of SMT to in-kernel applications, we added experimental support for it in NVMe-oF. NVMe-oF is a remote block storage service to connect fast NVMe-based SSD devices and network clients. It is implemented in the kernel to avoid moving the data between the user and kernel spaces. If the NVMe-OF stack was implemented in user space, to handle a read request, the data needs to be moved out of the kernel block layer and then moving the data back into the kernel to send it over TCP.

Similar to Redis, the most effort was about supporting Homa, and once it was done, SMT support was trivial. It was slightly harder than Redis due to the lack of Homa APIs for kernel clients, which we thus implemented. We needed to modify the NVMe-oF layer, because it expects stream abstraction of the network I/O, whereas SMT inherits Homa's RPC abstraction. Since the NVMe stack is implemented inside the Linux kernel block layer, we can use unmodified client applications on top of it.

Our current implementation is in early stage and still expensive, including one extra data copy compared to TCP and lack of support for multiple I/O queues.

Results. We use FIO [7], a widely-used storage benchmark tool, to generate random read requests to the remote SSD node over TCP, kTLS, Homa or SMT. We use the default NVMe block size, 4 K B, and force the data to be read from the NVMe SSD, not from the page cache.

Figure 9 plots the P50 and P99 request latency over varying iodepth, the number of requests sent without waiting for the response to the previous requests. We ran each iodepth 5 times, 30 seconds each, and plot the middle one in P50 latency. Although we were not able to observe the advantage of Homa or SMT when iodepth is 1–4 at P50 or 1–2 at P99, we saw up to 7 % (with TLS offload) or 15 % (without it) of P50 latency reduction, and up to 16 % (with TLS offload) or 21 % (without it) of P99 latency reduction.

Unlike Redis cases, we were not able to observe clear advantage of hardware TLS offloading, likely because the benefit was masked by other NVMe device or stack overheads that increase the end-to-end latency. We leave further analysis and improvement of NVMe-oF/SMT as future work.

5.5. Comparison with TCPLS (and QUIC)

TCPLS [77] augments TCP by extending TLS 1.3 to achieve the similar features to QUIC (§ 2.1). We compare SMT with TCPLS, because it outperforms all the QUIC implementations they tested, including Quicly (the fastest one), Msquic and mvfst, by at least 2.4× [77]. Figure 10 plots unloaded latency to highlight software overheads. SMT without TLS offload exhibits 5–18 % lower latency than TCPLS. SMT with TLS offload achieves 12–18 % lower latency than TCPLS (cannot use offload, see § 2.1).

5.6. Key Exchange Performance

We implemented handshake methods that support 0-RTT data with and without forward secrecy (§ 4.5). Figure 12 shows the RTT of the initial handshake and session resumption for each method, compared to the baseline, which performs a standard TLS 1.3 handshake over Homa (without pre-key generation). We use ECDH key exchange with secp256r1, the aes128gcmsha256 cipher suite, and ECDSA with the secp256r1 signature algorithm.

The SMT initial handshake outperforms standard TLS (Init-1RTT) by 37–44% when forward secrecy is enabled (Init-FS), otherwise (Init) 52–55 %. In addition to RTT saving, it eliminates C1.1 in Table 2 through key pre-generation, and C3.1 and C3.2 by verifying the certificate from SMT-ticket in advance on the client side. On the server side, it removes S2.1 by using a pre-generated ephemeral key share.

For resumption (denoted as Rsmp), our implementation also uses pre-generated keys at both ends. The margin between Rsmp-FS and Rsmp (no forward secrecy) is 338–387µs; it is reasonable, because the additional costs of S2.2 and C2.2 are similar.

6. Security Analysis

Security properties of SMT are based on TLS 1.3 [75]. This section first details how those properties are achieved in SMT design in § 6.1. We then discuss attacks outside the explicit defense scope of TLS 1.3 in § 6.2. Finally, we compare SMT with other TLS-based transport-level encryptions in terms of security properties in § 6.3.

6.1. TLS Security Properties

RFC8446 [75], which describes the TLS 1.3 specification, defines the following security properties in Section 1: *authentication*, *confidentiality*, and *integrity*. SMT ensures the authentication properties using TLS 1.3 handshake and thus inherits protection against attacks that affected earlier TLS versions (e.g., POODLE [58], BEAST [21], and Lucky 13 [2]). SMT ensures confidentiality and integrity using AEAD encryption, which provides both the properties.

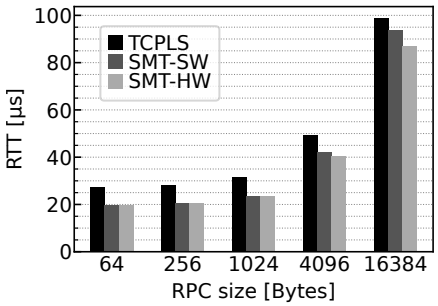

Figure 10: TCPLS comparison.

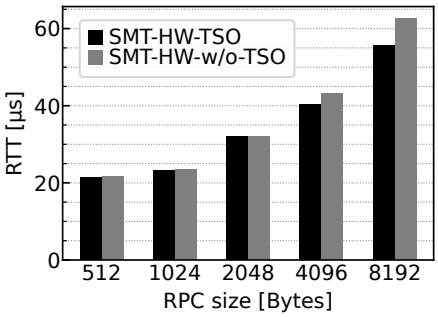

Figure 11: Effect of TSO.

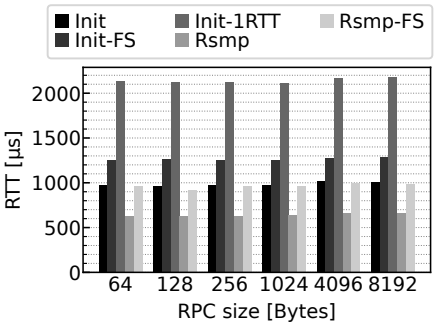

Figure 12: Key exchange latency.

RFC8446 defines additional guarantees that enhance confidentiality and integrity properties of TLS 1.3 in Section E.2: *order protection*, *non-replayability*, *length concealment*, and *forward secrecy after key change*. Since SMT needs to ensure some of those guarantees or mechanisms differently than TLS/TCP to support message-based transport, we discuss those in detail.

Order protection. This property ensures the attacker cannot make the receiver accept the out-of-order record sequence number. In TLS/TCP, the (single) byte stream in the connection is mapped to the TLS session with a single record sequence number space. Therefore, out-of-order data does not reach the TLS layer and the ordering guarantee of TLS prevents the altered TLS records that preserve the TCP-level correctness (i.e., sequence number, length and checksum) from being accepted at the TLS layer. To support the message abstraction that ensures reliable, in-order byte delivery *within* the message but different messages can be reordered, SMT applies the order protection in a per-message basis using per-message record sequence number space in a secure session (§ 4). Within the record sequence number space, the order and completeness of records are guaranteed by monotonically incrementing record sequence numbers, like TLS/TCP.

Non-replayability. SMT ensures this guarantee by composite message identities (§ 4.4). Per-message record sequence number space means the *relative* record sequence number can duplicate across the messages in the TLS session. To avoid replay, SMT ensures uniqueness of message ID throughout the session (§ 4.4.1). When the receiver detects the message ID already seen previously, it simply discards it without decryption, much like TCP discards the packet with the past sequence number. When the receiver receives a new message ID but with altered payload, it detects replay or injection when decrypting it, much like TLS receiver does so after receiving an altered segment but correct at the TCP layer (i.e., sequence number, length and checksum are correct).

Length concealment. TLS applications can conceal the true length of the application data to protect form side-channel attacks (although AEAD is safe without size changing in terms of confidentiality). SMT is compatible with TLS padding. Each record can include padding as regular TLS/TCP. When padding is used, the message length field (Figure 3) should include padding length to be aligned with the purpose of padding (i.e., hiding the true data length from the plaintext metadata); the mismatch between the true application message size and what the header indicate is not a problem, because the receiver can identify the padding length at the time of decryption (TLS records are reassembled based on packet IDs and TSO offsets, which happens prior to message-level reassembly (§ 4.3).

Forward secrecy after key exchange. This guarantee protects the user data from the leakage of server's private key. SMT ensures this guarantee by ephemeral (EC)DHE key exchange enforced by the TLS 1.3 handshake, except for the (optional) 0-RTT data discussed in § 4.5.

6.2. Metadata and Traffic Analysis Considerations

As noted in RFC8446, TLS 1.3 does not provide specific guarantees for traffic analysis attacks, although it provides a *mechanism* to conceal application data length through padding (see § 6.1). However, it is known that traffic metadata, such as packet timing, sizes, rate and bursts, could reveal meaning information from encrypted traffic, such as (candidate) identifies of website [44, 86] and video streaming [34, 50], often using ML-based methods [85, 9]. TLS 1.3 also does not provide defense to side-channel attacks caused by microarchitectural effects.

SMT also does not provide protection against those attacks except for the message length concealment mechanism. A notable metadata that could be exploited by traffic analysis is plaintext message ID and length field (Figure 3), which could be more meaningful compared to the sequence number field in TCP headers. However, if necessary, that can be obfuscated using TLS 1.3 padding, as discussed in § 6.1.

It is reported that plaintext TLS 1.3 record header could leak some degree of information to "weak" attackers [25]. The relevant concern is that TLS record headers are more easily identifiable when aligned with packet boundaries. In TCP, when multiple messages together form TCP segments, the positions of the record headers are unlikely to align with the beginning of the packet payload, except for the first one. SMT currently places TLS record headers aligned with message boundaries (§ 4.3), which would ease the eavesdropper to identify their positions. Nevertheless, we do not believe the impact of this alignment is particularly high in comparison to TLS/TCP, because TCP applications would use `TCP_NODELAY` anyways, which reduces opportunities of

the aforementioned record boundary obfuscation. We leave the analysis of effectiveness of traffic analysis attacks on SMT traffic as future work.

6.3. Other TLS-Based Encryptions

It is worthwhile to discuss how other TLS-based protocols guarantee the TLS 1.3 security properties, particularly for those that SMT guarantees differently from TLS/TCP. DTLS runs on UDP so it is impractical to assume that the underlying transport results in in-order, complete delivery beyond a single UDP datagram boundary. For the non-replayability guarantee, the DTLS receiver maintains a sliding window that defines the range of acceptable record sequence numbers and tracks those already received. This enables the receiver to uniquely identify valid records and discard any that either fall outside the window or are detected as replays. SMT doesn't use the sliding window, because Homa provides reliable, ordered byte delivery (like TCP) within the message. This allows each record sequence number space to assume the same underlying transport property as TCP.

QUIC does not use a record sequence number mechanism; per-stream ordering is handled above encryption. Replay protection is provided by a monotonically increasing packet number incorporated into the AEAD nonce. Receivers track processed packet numbers to discard duplicates. Unlike TLS/TCP, QUIC accepts higher packet numbers even if earlier ones are missing, tolerating reordering. Therefore, QUIC's encryption mechanism, as shown in Figure 4, can be seen as per-packet record sequence number space in contrast to per-message one in SMT and per-connection one in TLS/TCP.

Note that header protection encrypts the packet number to prevent middlebox interference and ossification, not as a replay defense. Unlike QUIC, SMT does not enforce header protection, because middlebox ossification is less relevant in datacenters. This design preserves the property of message-based transports that enables per-message load balancing between the host stack CPU cores or multiple paths (§ 7)

7. Discussion

Message 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. This means that the application must compute the checksum and embed it in their messages.

SMT intrinsically obviates this problem, because encrypting and decrypting the message with TLS ensures integrity is verified. Moreover, the cryptographic operations can be offloaded to the NIC hardware.

Segmentation. IPv6 does not have an equivalent field to the IPv4 IPID, which we currently use for reassembling the TSO segment (§ 4.3). We can still use TLS offload without TSO, which we plot the impact in Figure 11. Note that the penalty of disabling TSO in SMT or Homa would not be as large as in TCP cases. This is because, although TSO does both segmentation and checksumming, Homa does not use the latter or guarantee message integrity (SMT intrinsically does it based on message encryption and decryption).

We can use TSO for every pair of packets, as the receiver can reassemble them based on the presence of the TLS record header. For larger TSO segments, we use GSO to split them into two-packet-sized TSO segments at the bottom of the stack. These smaller TSO segments contain incrementing TSO offsets generated by GSO (Figure 3). If the NIC vendor activates sequence number embedding to non-TCP packets, which could not require significant changes to the NIC implementation, SMT will enable full TSO.

Receive-side offload. Receiver-side cryptographic NIC offload is challenging because the NIC does not route incoming non-TCP packets through its cryptographic engine, although it would be a straightforward task for the vendor to implement. For example, Pensando Elba features a P4-based ingress packet processing engine [8] situated before the cryptographic engine. Although we confirmed that sender-side offload accelerates RPC throughput by up to 5–13 %, receive-side offload would further accelerate it.

Encryption protocol. SMT 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 SMT 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. SMT is compatible with In-Network Compute (INC) enabled by MTP [43], unlike other encrypted transports discussed in § 2.1. This is because it leaves message ID and length unencrypted and thus allows a network node to identify message boundaries of application-level messages with bounded resource usage, for example, for congestion signalling or load balancing. SMT is compatible with packet trimming used by NDP [33] and UET [16], because trimmed packets carry useful information (i.e., plaintext transport metadata) for the receiver to identify the sender demands.

Hardware-based transport. SMT is designed to be used as replacement of TLS/TCP in ordinary cloud environments that provide virtualized or baremetal instances or networks, allowing opportunistic use of NIC offload for segmentation and encryption. If implemented entirely in hardware like Falcon/PSP, SMT would achieve lower small RPC latency due to end-to-end SRPT scheduling inherited from Homa; SMT has some communication overheads for compatibility with existing offload, like TCP-structured header, whereas Falcon/PSP avoids those overheads with clean-slate packet format supported by the custom NIC.

Post-quantum resistance. SMT inherits post-quantum resistance of TLS 1.3 using an appropriate handshake that prevents captured handshakes from being attacked. The user may opt for longer keys for slightly better quantum resistance. In this case, the benefit of hardware offload would be larger.

Although we used 128-bit key in this paper (§ 5), the NIC we used also supports 256-bit keys for TLS offload.

8. Related Work

Much of the related work has been discussed in § 2; the remaining topics are covered here.

RDMA network security. ReDMArK [78] demonstrates packet injection attacks in RDMA networks, highlighting the need for encryption, such as IPSec or sRDMA [89], which mitigates these attacks by using symmetric cryptography and embedding MAC in the RDMA header. sRDMA employs symmetric cryptography for authentication and encryption, and extends the RDMA packet header to embed MAC. The application establishes RDMA connections (QPs) with the agent in the local SmartNIC whose CPUs perform authentication and encryption jobs, attaching or removing outer headers. [83] proposes encrypting RDMA packets with DTLS using hardware acceleration, but SMT does not opt for DTLS, because it needs TSO and large message support.

Key exchange acceleration. SSLShader [41] and SmartTLS [48] accelerate the TLS handshake using GPU and SmartNIC, respectively. Those can be used for SMT if key exchange is performed based on TLS, although we explored lightweight methods based on symmetric keys (§ 4.5).

Host stack enhancements. Host stack improvements, such as batching [32], zero copy [94], flexible core allocation [13] and better NIC abstraction [79], complement SMT, though TCP-specific optimizations like congestion control [3] and handshake improvements [70] are not applicable. ByteDance has reported their effort of improving Homa for their RPC traffic [53], 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 transparently applicable to SMT; they report Homa's throughput is lower than TCP when the message size is larger than 50 K B, which we also observe similarly in § 5.

Transport protocol design. It is worth discussing design patterns of transport protocols. Multipath TCP [72] focuses on robustness against middlebox interference prevalent in the Internet [72]. Its compatibility with TSO also enables datacenter usage for large transfers [71], although sharing the problems with TCP for RPCs (§ 2.2). SCTP [87] defines its own protocol number, which is viable in its primary target, telecommunication networks, but has resulted in low adoption in the Internet due to middlebox interference. It is also not datacenter friendly due to high software overheads and protocol complexity. SMT's design point is support for existing hardware offload and RPCs without consideration of middleboxes that block transport protocols other than TCP or UDP, which are prevalent in the Internet.

Transport multiplexing. Aquila [29] and EQDS [62] 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). SMT can be multiplexed within these systems, providing abstraction and encryption to the application, and can also be used between edge functions.

9. Conclusion

We explored a new design point of secure datacenter transport protocols to transition from the current TLS/TCP ecosystem for more efficient secure datacenter networking. We found that the TLS record protocol can be used with new message-based transports designed for datacenter RPCs together with existing TLS offload available in commodity NICs, but doing so required tightly coupling transport protocol and encryption to preserve the security properties of TLS/TCP.

The current SMT implementation inherits performance issues from Homa [53], but we expect those will be mitigated due to its active development.

Acknowledgments

We are grateful to the anonymous reviewers and shepherd 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] *2017 Equifax data breach*. https://en.wikipedia.org/wiki/2017_Equifax_data_breach. 2017.

[2] Nadhem J Al Fardan and Kenneth G Paterson. "Lucky thirteen: Breaking the TLS and DTLS record protocols". *IEEE S&P*. 2013.

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. "Data Center TCP (DCTCP)". *ACM SIGCOMM*. 2010.

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. "SCONE: Secure Linux Containers with Intel SGX". *USENIX OSDI*. 2016.

[5] Behnaz Arzani, Selim Ciraci, Stefan Saroiu, Alec Wolman, Jack Stokes, Geoff Outhred, and Lechao Diwu. "PrivateEye: Scalable and Privacy-Preserving compromise detection in the cloud". *USENIX NSDI*. 2020.

[6] Shinichi Awamoto and Michio Honda. "Opening Up Kernel-Bypass TCP Stacks". *USENIX ATC*. 2025.

[7] Jens Axboe. *Flexible IO Tester (FIO)*. https://github.com/axboe/fio.

[8] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, et al. "Disaggregating Stateful Network Functions". *USENIX NSDI*. 2023.

[9] Sanjit Bhat, David Lu, Albert Kwon, and Srinivas Devadas. "Var-CNN: A Data-Efficient Website Fingerprinting Attack Based on Deep Learning". *PETS* (2019).

[10] Andrea Bittau, Daniel B. Giffin, Mark J. Handley, David Mazieres, Quinn Slack, and Eric W. Smith. *Cryptographic Protection of TCP Streams (tcpcrypt)*. RFC 8548. 2019. URL: https://www.rfc-editor.org/info/rfc8548.

[11] Andrea Bittau, Michael Hamburg, Mark Handley, David Mazieres, and Dan Boneh. "The Case for Ubiquitous Transport-Level Encryption". *USENIX Security*. 2010.

[12] Daniel Borkmann and John Fastabend. *Combining kTLS and BPF for Introspection and Policy Enforcement*. Linux Plumbers Conference 2018.

[13] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. "Towards μ s tail latency and terabit ethernet: disaggregating the host network stack". *ACM SIGCOMM*. 2022.

[14] Cilium. *eBPF-based Networking, Security, and Observability*. https://github.com/cilium/cilium.

[15] Cilium. *Improving the security of Cilium Mutual Authentication*. https://cilium.io/blog/2024/03/20/improving-mutual-auth-security/.

[16] Ultra Ethernet Consortium. *Ultra Ethernet Specification v1.0.1*. https://ultraethernet.org/wp-content/uploads/sites/20/2025/10/UE-Specification-1.0.1.pdf. 2025.

[17] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking cloud serving systems with YCSB". *ACM SoCC*. 2010.

[18] Lazar Cvetković, François Costa, Mihajlo Djokic, Michal Friedman, and Ana Klimovic. "Dirigent: Lightweight Serverless Orchestration". *ACM SOSP*. 2024.

[19] Bruce Davie. *The Challenge of East-West Traffic*. https://systemsapproach.substack.com/p/the-challenge-of-east-west-traffic. 2023.

[20] Bruce Davie and Jesse Gross. *A Stateless Transport Tunneling Protocol for Network Virtualization (STT)*. Internet-Draft draft-davie-stt-05. 2014.

[21] Thai Duong and Juliano Rizzo. "Here Come The $\oplus$ Ninjas". *Unpublished manuscript* (2011).

[22] Korian Edeline and Benoit Donnet. "A bottom-up investigation of the transport-layer ossification". *IEEE TMA*. 2019.

[23] Danielle E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. "Maglev: A fast and reliable software network load balancer". *USENIX NSDI*. 2016.

[24] John Fastabend. *Seamless transparent encryption with BPF and Cilium*. Linux Plumbers Conference 2019.

[25] Bryan Alexander Ford. "Metadata Protection Considerations for TLS Present and Future". *TLS 1.3 Ready or Not (TRON) Workshop*. 2016.

[26] Drew Gallatin. *Serving Netflix Video Traffic at 400Gb/s and Beyond*. https://nabstreamingsummit.com/wp-content/uploads/2022/05/2022-Streaming-Summit-Netflix.pdf.

[27] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems". *ACM ASPLOS*. 2019.

[28] Barton Gellman and Ashkan Soltani. *NSA infiltrates links to Yahoo, Google data centers worldwide, Snowden documents say*. The Washington Post, https://www.washingtonpost.com/world/national-security/nsa-infiltrates-links-to-yahoo-google-data-centers-worldwide-snowden-documents-say/2013/10/30/e51d661e-4166-11e3-8b74-d89d714ca4dd_story.html. 2013.

[29] 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. "Aquila: A unified, low-latency fabric for datacenter networks". *USENIX NSDI*. 2022.

[30] Google. *Announcing PSP's cryptographic hardware offload at scale is now open source*. https://cloud.google.com/blog/products/identity-security/announcing-psp-security-protocol-is-now-open-source.

[31] Google. *Encryption in transit for Google Cloud*. https://cloud.google.com/docs/security/encryption-in-transit. 2025 (last updated).

[32] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. "MegaPipe: A New Programming Interface for Scalable Network I/O". *USENIX OSDI*. 2012.

[33] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. "Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance". *ACM SIGCOMM*. 2017.

[34] David Hasselquist, Ethan Witwer, August Carlson, Niklas Johansson, and Niklas Carlsson. "Raising the Bar: Improved Fingerprinting Attacks and Defenses for Video Streaming Traffic". *PETS* (2024).

[35] Seppo Hätönen, Aki Nyrhinen, Lars Eggert, Stephen Strowes, Pasi Sarolahti, and Markku Kojo. "An experimental study of home gateway characteristics". *ACM SIGCOMM*. 2010.

[36] Tom Herbert. *Data center networking stack*. The Technical Conference on Linux Networking (Netdev

1.2), https://legacy.netdevconf.info/1.2/session.html?tom-herbert/. 2016.

[37] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. "PASTE: A Network Programming Interface for Non-Volatile Main Memory". *USENIX NSDI*. 2018.

[38] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. "Is It Still Possible to Extend TCP?": *ACM IMC*. 2011.

[39] Kevin Hsieh, Mike Wong, Santiago Segarra, Sathiya Kumaran Mani, Trevor Eberl, Anatoliy Panasyuk, Ravi Netravali, Ranveer Chandra, and Srikanth Kandula. "NetVigil: Robust and Low-Cost Anomaly Detection for East-West Data Center Security". *USENIX NSDI*. 2024.

[40] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. 2021. URL: https://www.rfc-editor.org/info/rfc9000.

[41] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. "SSLShader: Cheap SSL Acceleration with Commodity Processors". *USENIX NSDI*. 2011.

[42] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. "mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems". *USENIX NSDI*. 2014.

[43] Tao Ji, Rohan Vardekar, Balajee Vamanan, Brent E. Stephens, and Aditya Akella. "MTP: Transport for In-Network Computing". *USENIX NSDI*. 2025.

[44] Marc Juarez, Sadia Afroz, Gunes Acar, Claudia Diaz, and Rachel Greenstadt. "A Critical Evaluation of Website Fingerprinting Attacks". *ACM CCS*. 2014.

[45] *Kernel Connection Multiplexer*. https://www.kernel.org/doc/Documentation/networking/kcm.txt.

[46] *Kernel TLS offload*. https://www.kernel.org/doc/html/latest/networking/tls-offload.html.

[47] Jakub Kicinski. *[RFC net-next 00/15] add basic PSP encryption for TCP connections*. https://lore.kernel.org/all/20240510030435.120935-1-kuba@kernel.org/. 2024.

[48] Duckwoo Kim, SeungEon Lee, and KyoungSoo Park. "A case for smartnic-accelerated private communication". *ACM APNet*. 2020.

[49] Hugo Lefeuvre, David Chisnall, Marios Kogias, and Pierre Olivier. "Towards (Really) Safe and Fast Confidential I/O". *ACM HotOS*. 2023.

[50] Feng Li, Jae Won Chung, and Mark Claypool. "Silhouette: Identifying youtube video flows from encrypted traffic". *ACM NOSSDOV*. 2018.

[51] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. "Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories". *USENIX OSDI*. 2020.

[52] Chien-Chih Liao and Sangeeta Kundu Pawel Krolikowski. *Better Load Balancing: Real-Time Dynamic Subsetting*. https://www.uber.com/en-GB/blog/better-load-balancing-real-time-dynamic-subsetting/.

[53] Xiaochun Lu and zijian Zhang. *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.

[54] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. "Snap: A microkernel approach to host networking". *ACM SOSP*. 2019.

[55] Meta. *Building Facebook's service encryption infrastructure*. https://engineering.fb.com/2019/05/29/security/service-encryption/. 2019.

[56] Meta. *Post-quantum readiness for TLS at Meta*. https://engineering.fb.com/2024/05/22/security/post-quantum-readiness-tls-pqr-meta/. 2024.

[57] Jeffrey C. Mogul and John Wilkes. "Physical Deployability Matters". *ACM HotNets*. 2023.

[58] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. "This POODLE bites: exploiting the SSL 3.0 fallback". *Security Advisory* (2014).

[59] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. "Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities". *ACM SIGCOMM*. 2018.

[60] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. "AccelTCP: Accelerating network applications with stateful TCP offloading". *USENIX NSDI*. 2020.

[61] Michael F Nowlan, Nabin Tiwari, Janardhan Iyengar, Syed Obaid Amin, and Bryan Ford. "Fitting Square Pegs Through Round Pipes: Unordered Delivery Wire-Compatible with TCP and TLS". *USENIX NSDI*. 2012.

[62] Vladimir Olteanu, Haggai Eran, Dragos Dumitrescu, Adrian Popa, Cristi Baciu, Mark Silberstein, Georgios Nikolaidis, Mark Handley, and Costin Raiciu. "An edge-queued datagram service for all datacenter traffic". *USENIX NSDI*. 2022.

[63] John Ousterhout. "A Linux Kernel Implementation of the Homa Transport Protocol". *USENIX ATC*. 2021.

[64] John Ousterhout. *Begin upstreaming Homa transport protocol*. https://lwn.net/Articles/997858/.

[65] John Ousterhout. *Kernel-Managed User Buffers in Homa*. The Technical Conference on Linux Networking (Netdev 0x17), https://netdevconf.info/0x17/docs/netdev-0x17-paper38-talk-slides/HomaBuffersNetDev.pdf. 2023.

[66] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. "Facebook's Tectonic Filesystem: Efficiency from Exascale". *USENIX FAST*. 2021.

[67] Maxime Piraux, Olivier Bonaventure, and Florentin Rochet. *TCPLS: Modern Transport Services with TCP and TLS*. Internet-Draft draft-piraux-tcpls-00. 2021.

[68] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. "Autonomous NIC offloads". *ACM ASPLOS*. 2021.

[69] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. "The benefits of general-purpose on-NIC memory". *ACM ASPLOS*. 2022.

[70] Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. "TCP fast open". *ACM CoNEXT*. 2011.

[71] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. "Improving datacenter performance and robustness with multipath TCP". *ACM SIGCOMM*. 2011.

[72] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. "How hard can it be? designing and implementing a deployable multipath TCP". *USENIX NSDI*. 2012.

[73] Sivaramakrishnan Ramanathan, Ying Zhang, Mohab Gawish, Yogesh Mundada, Zhaodong Wang, Sangki Yun, Eric Lippert, Walid Taha, Minlan Yu, and Jelena Mirkovic. "Practical Intent-driven Routing Configuration Synthesis". *USENIX NSDI*. 2023.

[74] Jinglei Ren. *YCSB-C*. https://github.com/basicthinker/YCSB-C.

[75] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. 2018. URL: https://www.rfc-editor.org/info/rfc8446.

[76] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. *TLS Encrypted Client Hello*. Internet-Draft draft-ietf-tls-esni-17. 2023.

[77] Florentin Rochet, Emery Assogba, Maxime Piraux, Korian Edeline, Benoit Donnet, and Olivier Bonaventure. "TCPLS: Modern Transport Services with TCP and TLS". *ACM CoNEXT*. 2021.

[78] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. "ReDMArk: Bypassing RDMA Security Mechanisms". *USENIX Security*. 2021.

[79] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. "Enso: A Streaming Interface for NIC-Application Communication". *USENIX OSDI*. 2023.

[80] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. "ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta". *USENIX OSDI*. 2023.

[81] 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. "A Cloud-Scale Characterization of Remote Procedure Calls". *ACM SOSP*. 2023.

[82] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. "A cloud-optimized transport protocol for elastic and scalable hpc". *IEEE micro* (2020).

[83] Anna Kornfeld Simpson, Adriana Szekeres, Jacob Nelson, and Irene Zhang. "Securing RDMA for High-Performance Datacenter Storage Systems". *USENIX HotCloud*. 2020.

[84] Arjun Singhvi, Nandita Dukkipati, Prashant Chandra, Hassan M. G. Wassel, Naveen Kr. Sharma, Anthony Rebello, Henry Schuh, Praveen Kumar, Behnam Montazeri, Neelesh Bansod, Sarin Thomas, Inho Cho, Hyojeong Lee Seibert, Baijun Wu, Rui Yang, Yuliang Li, Kai Huang, Qianwen Yin, Abhishek Agarwal, Srinivas Vaduvatha, Weihuang Wang, Masoud Moshref, Tao Ji, David Wetherall, and Amin Vahdat. "Falcon: A Reliable, Low Latency Hardware Transport". *ACM SIGCOMM*. 2025.

[85] Payap Sirinam, Mohsen Imani, Marc Juarez, et al. "Deep Fingerprinting: Undermining Website Fingerprinting Defenses With Deep Learning". 2018.

[86] J Smith, P Mittal, and A Perrig. "Website Fingerprinting in the Age of QUIC". *PETS* (2021).

[87] Randall R. Stewart, Michael Tüxen, and karen Nielsen. *Stream Control Transmission Protocol*. RFC 9260. 2022. URL: https://www.rfc-editor.org/info/rfc9260.

[88] Lizhuang Tan, Wei Su, Yanwen Liu, Xiaochuan Gao, and Wei Zhang. "DCQUIC: Flexible and Reliable Software-defined Data Center Transport". *IEEE INFOCOM WKSHPS*. 2021.

[89] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. "sRDMA–Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access". *USENIX ATC*. 2020.

[90] Mellanox Technologies. *Mellanox Corporate Update—Unleashing the Power of Data*. 2020.

[91] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. "Rkt-Io: A Direct I/O Stack for Shielded Execution". *ACM EuroSys*. 2021.

[92] Martin Thomson and Sean Turner. *Using TLS to Secure QUIC*. RFC 9001. 2021. URL: https://www.rfc-editor.org/info/rfc9001.

[93] Filippo Valsorda. *An overview of TLS 1.3 and Q&A*. https://blog.cloudflare.com/tls-1-3-overview-and-q-and-a. 2016.

[94] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. "StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs". *USENIX ATC*. 2016.

[95] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. "The demikernel datapath os architecture for microsecond-scale datacenter systems". *ACM SOSP*. 2021.

[96] Jipeng Zhang, Junhao Huang, Lirui Zhao, Donglong Chen, and Çetin Kaya Koç. "ENG25519: Faster TLS

1.3 handshake using optimized X25519 and Ed25519". *USENIX Security*. 2024.

[97] Xumiao Zhang, Shuowei Jin, Yi He, Ahmad Hassan, Z. Morley Mao, Feng Qian, and Zhi-Li Zhang. "QUIC is not Quick Enough over Fast Internet". *ACM WWW*. 2024.

[98] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. "CRISP: Critical Path Analysis of Large-Scale Microservice Architectures". *USENIX ATC*. 2022.

[99] Xiangfeng Zhu, Yuyao Wang, Banruo Liu, Yongtong Wu, Nikola Bojanic, Jingrong Chen, Gilbert Louis Bernstein, Arvind Krishnamurthy, Sam Kumar, Ratul Mahajan, and Danyang Zhuo. "High-level Programming for Application Networks". *USENIX NSDI*. 2025.

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary of Paper

This paper addresses the lack of transport-level encryption for modern datacenter transport protocols that use a message abstraction. The authors propose SMT, which adds transport-level encryption onto the Homa protocol, while remaining compatible with TLS to take advantage of hardware offloading available on commodity NICs.

A.2. Scientific Contributions

- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

A.3. Reasons for Acceptance

1) This paper addresses a long known issue. The authors provide a solution for transport-level encryption in datacenters that is feasible for adoption.
2) This paper proposes a valuable step forward in an established field. The authors provide a solution for transport-level encryption in message-based protocols while enabling the ability to leverage hardware offloading meant for stream-based protocols.