Autonomous NIC Offloads

Boris Pismenny
Technion and NVIDIA
Haifa, Israel

Haggai Eran
Technion and NVIDIA
Haifa, Israel

Aviad Yehezkel
NVIDIA
Yokne'am Illit, Israel

Liran Liss
NVIDIA
Yokne'am Illit, Israel

Adam Morrison
Tel Aviv University
Tel Aviv, Israel

Dan Tsafrir
Technion and VMware Research
Haifa, Israel

ABSTRACT

CPUs routinely offload to NICs network-related processing tasks like packet segmentation and checksum. NIC offloads are advantageous because they free valuable CPU cycles. But their applicability is typically limited to layer≤4 protocols (TCP and lower), and they are inapplicable to layer-5 protocols (L5Ps) that are built on top of TCP. This limitation is caused by a misfeature we call "offload dependence," which dictates that L5P offloading additionally requires offloading the underlying layer≤4 protocols and related functionality: TCP, IP, firewall, etc. The dependence of L5P offloading hinders innovation, because it implies hard-wiring the complicated, ever-changing implementation of the lower-level protocols.

We propose "autonomous NIC offloads," which eliminate offload dependence. Autonomous offloads provide a lightweight software-device architecture that accelerates L5Ps without having to migrate the entire layer≤4 TCP/IP stack into the NIC. A main challenge that autonomous offloads address is coping with out-of-sequence packets. We implement autonomous offloads for two L5Ps: (i) NVMe-over-TCP zero-copy and CRC computation, and (ii) https authentication, encryption, and decryption. Our autonomous offloads increase throughput by up to 3.3x, and they deliver CPU consumption and latency that are as low as 0.4x and 0.7x, respectively. Their implementation is already upstreamed in the Linux kernel, and they will be supported in the next-generation of Mellanox NICs.

CCS CONCEPTS

• **Networks → Network adapters**.

KEYWORDS

NIC, operating system, hardware/software co-design

ACM Reference Format:
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 (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3445814.3446732

1 INTRODUCTION

Layer-5 networking protocols (L5Ps) built on top of TCP are commonplace and widely used. Examples include: (1) the transport layer security (TLS) cryptographic protocol [17, 107], which provides secure communications for, e.g., browsers via https [106]; (2) storage protocols like NVMe-TCP [92], which allow systems to use remote disk drives as local block devices; (3) remote procedure call (RPC) protocols, such as Thrift [116] and gRPC [32]; and (4) key-value store protocols, such as Memcached [24] and MongoDB [52].

L5Ps are frequently *data-intensive*, as it is in their nature to move bytes of network streams to/from memory while possibly transforming or computing some useful function over them. Such processing tends to be computationally heavy and therefore might adversely affect the performance of the applications that utilize and depend on the L5Ps. In most cases [55], the data-intensive processing consists of: encryption/decryption, copying, hashing, (de)serialization, and/or (de)compression. It is consequently beneficial to accelerate these operations and thereby improve the performance of the corresponding applications, which is our goal in this paper.

We classify prior approaches to accelerate L5P processing into three categories (§2). The first is *software-based*. It includes in-kernel L5P implementations, such as that of NVMe-TCP [35, 41] and TLS [16]. It also includes specialized software stacks that bypass the kernel and leverage direct hardware access [70, 71]. These type of techniques are good for reducing the cost of system software abstractions. But they are largely irrelevant for accelerating the actual data-intensive operations.

The second category consists of *on-CPU acceleration*. It encompasses specialized data-processing CPU instructions, such as those supporting the advanced encryption standard (AES) [15, 36], secure hash algorithms (SHA) [1, 37], and cyclic redundancy check (CRC) error detection [33, 113]. These instructions can be effective in accelerating L5Ps. But then they themselves become responsible for most of the L5P processing cycles, motivating the use of *off-CPU accelerators*, which comprise the third category, and which we further subdivide into two subcategories: accelerators that are off and on the networking path.

Off-path computational accelerators include various devices that may encrypt, decrypt, (de)compress, digest, and pattern-match the data [31, 40]. Their goal is to allow the CPU to offload much of the data-intensive computation onto them to make the code run faster. The problem is: (1) that the CPU must still spend valuable cycles on feeding the accelerators and on retrieving the results; (2) that developers might need to rewrite applications in nontrivial ways to effectively exploit the accelerators; and (3) that, regardless, the accelerators consume memory bandwidth and increase latency

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04...$15.00
https://doi.org/10.1145/3445814.3446732

because the CPU must communicate with them via mechanisms such as direct memory access (DMA). Therefore, the outcome of using off-path accelerators might be suboptimal [4].

NICs are *on-path* accelerators, and they do not suffer from the above problems. Because L5Ps are, in fact, network protocols, they necessarily operate NICs in any case, and so driving them (as accelerators) does not incur any additional overhead costs. Moreover, NICs are ideally situated for L5P acceleration, as they process the data while it flows through them, avoiding the aforementioned latency increase and additional memory bandwidth consumption associated with off-path accelerators. NICs already routinely seamlessly handle offloaded computation for the underlying layer≤4 protocols, such as packet segmentation, aggregation, and checksum computation and verification [14, 21, 83].

Despite their seemingly ideal suitability, L5P NIC offloads are not pervasive. The reason: existing designs assign NICs with the role of handling the L5P, which in turn necessitates that the NICs also handle layer≤4 functionality upon which the L5P depends, most notably that of TCP/IP [11, 22, 47, 48]. Such *offload dependence* is undesirable, as implementing TCP in hardware encumbers innovation in the network stack [86], and it slows down fixes when robustness [99] or security issues [53, 102] arise. For these reasons, Linux does not support TCP offloads [25, 69], and Windows recently deprecated such support [84].

We propose autonomous NIC offloads to address the above shortcomings and eliminate the aforementioned undesirable dependence. Autonomous offloads facilitate a hardware-software collaboration for moving data between L5P memory and TCP packets, optionally transforming or computing some function over the transmitted bytes. Autonomous offloads allow L5P software to outsource its data-intensive operations to the NIC, while leveraging the existing TCP/IP stack rather than subsuming it, and thus ridding NIC designers form the need to migrate the entire TCP/IP stack into the NIC. Autonomous offloads are applicable to setups where the L5P and NIC driver can communicate directly, as is the case with, for example, in-kernel L5P implementations or when userspace TCP/IP stacks are utilized.

The idea underlying autonomous offloads is for the L5P and NIC to jointly process L5P messages (which may consist of multiple TCP segments) in a manner that is transparent to the intermediating TCP/IP stack. When sending a message, the L5P code "skips" performing the offloaded operation, thereby passing the "wrong" bytes down the stack to the NIC. The NIC then performs the said skipped operation, resulting in a correct message being sent on the wire. In the reverse direction, under normal conditions, the NIC parses incoming messages and likewise performs the offloaded operation instead of the L5P while keeping the TCP/IP stack unaware.

A major challenge that we tackle when designing autonomous offloads is handling out-of-sequence traffic, which occurs when TCP packets are lost, duplicated, retransmitted, or reordered. We use the following three principles to address this challenge: (1) we optimize for the common case by maintaining a small context at the NIC to process the next in-order TCP packet; (2) we fall back on L5P software processing upon out-of-sequence packets; in which case (3) we employ a minimalist NIC-L5P interface that allows the L5P software to help the NIC hardware and driver to resynchronize and reconstruct the aforementioned context.

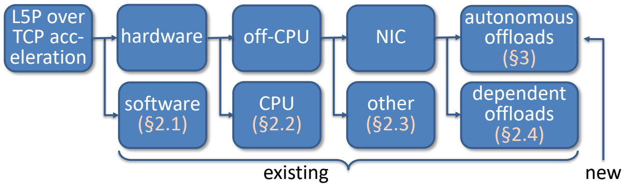


Figure 1: *Categorizes of L5P acceleration. Autonomous NIC offloads (this paper) are NIC offloads that do not require offloading the entire network stack.*

Context reconstruction for incoming traffic (which is harder than that of outgoing traffic) is driven by the NIC hardware and consists of: (1) speculatively identifying an L5P message header in the incoming data using some "magic pattern" characteristic of the L5P; (2) asking the L5P software to confirm this speculation using the aforementioned interface; (3) tracking the speculated L5P messages while waiting for confirmation; and (4) seamlessly resuming offloading activity once confirmation arrives.

Not all common L5Ps can be autonomously offloaded. In §3, we highlight the main ideas underlying autonomous offloads, and, importantly, we identify the properties that L5Ps should have to be autonomously offloadable. Then, in §4, we describe the software and hardware design of autonomous offloads in general, and, in §5, we present our concrete implementation for two L5Ps: NVMe-TCP and TLS, as well as their combination.

Our TLS autonomous offload is already implemented in the latest generation of Mellanox ConnectX ASIC NICs [79]; it offloads TLS authentication, encryption, and decryption functionalities. Our NVMe-TCP autonomous offload implementation will become available in a subsequent model; it offloads data placement at the receiving end (which thus becomes zero-copy) and also CRC computation and verification at both ends. Linux kernel support for the former has been upstreamed, while the latter is currently under review.

We experimentally evaluate the two offloads and their combination in §6, and we find that they provide throughput that is up to 3.3x higher and latency that is as low as 0.7x. When I/O devices become saturated, we show that our autonomous offloads provide CPU consumption that is as low as 0.4x. We further show that our offloads are resilient to loss, reordering, and performance cliffs when scaling to thousands of flows that far exceed NIC cache capacities. Finally, in §7 and §8, we respectively expand on the applicability of autonomous offloads and conclude.

2 BACKGROUND AND MOTIVATION

Considerable effort went into L5P acceleration. Here, we make the case for autonomous NIC offloads by categorizing existing approaches and showing that our proposal fills an important missing piece in the L5P acceleration design space.

Figure 1 depicts the categories. L5P acceleration has two flavors: software- (§2.1) and hardware-based. The latter occurs on- (§2.2) or off-CPU via specialized accelerators (§2.3) or NICs. Existing NIC acceleration requires implementing TCP and lower protocols on the NIC, which we thus call dependent NIC offloading (§2.4). In contrast, our NIC provides autonomous offloading, which keeps all protocols in software, and which strictly improves performance per dollar (§2.5).

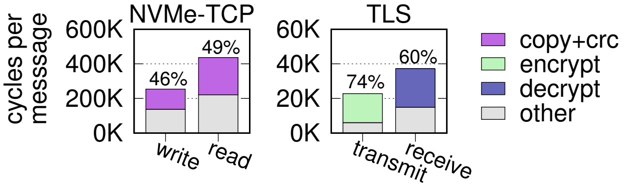


Figure 2: *L5P overheads. NVMe-TCP and TLS use 256K and 16K messages, respectively. Labels show how many cycles out of the total are compute-bound and offloadable; see §6 for more details.*

2.1 Software Acceleration

Software acceleration reduces overheads imposed on L5Ps by OS abstractions. It does not accelerate actual L5P computations, such as encryption, compression, and error detection.

In-kernel L5P implementations work around OS overheads. Examples include DRBD [105], SMB [28], NBD [10], iSCSI [2], and more recently TLS [16] and NVMe-TCP [35]. They reside alongside the OS's TCP/IP and improve performance through cross-layer data batching [41], fusing data manipulations [13], controlling flow group scheduling [12] and CPU scheduling. Notably, they eliminate system call overheads [101].

Instead of moving the entire L5P into the kernel, specialized software stacks bypass certain kernel overheads by: utilizing hardware queues exposed to userspace [46, 109]; implementing the TCP/IP stack in userspace [51, 57, 75, 95, 100]; replacing the POSIX API abstractions [7, 39, 95]; and exploiting knowledge of L5P workloads [70, 71].

In contrast to software, hardware acceleration targets data-intensive compute-bound processing, which accounts for much of the L5P cycles. Figure 2 measures these cycles in four in-kernel L5P workloads: (1) NVMe-TCP client write (compute: CRC of outgoing L5P message *m*); (2) NVMe-TCP client read (verify CRC of incoming *m* and copy its content to OS block layer); (3) TLS transmit (encrypt *m*); and (4) TLS receive (decrypt *m*). We see that the CPU spends 46%–74% of its cycles on the compute-bound part, even though these L5Ps are in-kernel. Such overhead can only be reduced via hardware acceleration. Thus, software and hardware accelerations are symbiotic, and L5Ps may benefit independently from both.

2.2 On-CPU Acceleration

On-CPU acceleration occurs via an in-core hardware implementation invoked by dedicated instructions. In the context of our above examples, it is available in commercial CPUs for AES encryption and decryption [36], and SHA and CRC32 digests [33, 37, 68, 119]. Dedicated instructions yield 2x–30x speedups [34, 126] but might still account for a significant fraction of L5P processing cycles. For instance, the results in Figure 2 are obtained with on-CPU accelerators, and yet the accelerated computations take up to 49% and 74% of NVMe-TCP and TLS message processing cycles, respectively.

2.3 Dedicated Off-CPU Accelerators

Off-CPU acceleration *offloads* part of the L5P computation from the CPU to another device accessible via PCIe or the on-chip interconnect. Offloading aims at freeing CPU cycles that would otherwise be devoted to the computation, allowing the CPU to do other work instead. We distinguish dedicated accelerators, discussed here, from on-NIC offloads (§2.4).

cipher	*QAT 1*	*QAT 128*	*AES-NI 1*
AES-128-CBC-HMAC-SHA1	249	3144	695
AES-128-GCM	249	3109	3150

Table 1: *Encryption bandwidth (MB/s) of AES-NI (on-CPU) vs. QAT (off-CPU) accelerators. Results for 16 KB blocks with 1 or 128 threads using a single core (2.40 GHz Intel Xeon E5-2620 v3 CPU).*

Dedicated off-CPU accelerators exist for various computational operations, such as: (de)compression, (a)symmetric encryption, digest computation, and pattern matching [27, 31, 44, 112]. Such off-CPU acceleration still requires some CPU work for each computation, to invoke the accelerator and retrieve the results. This work incurs latency and overhead that depend on the amount of data moved, the interface of the accelerator, and its location in the non-uniform DMA topology [4, 40, 117]. As a result, off-CPU accelerators can struggle to outperform on-CPU accelerators [125]. Realizing benefits from off-CPU acceleration thus frequently requires re-engineering applications to eliminate blocking operations and/or using multiple threads, all so as to keep the CPU busy while waiting for the accelerator [40].

To illustrate, Table 1 compares the throughput (single core OpenSSL speed test) of Intel off-CPU QuickAssist Technology (QAT) [44] accelerated cryptographic operations to on-CPU AES-NI acceleration [36]. For QAT, we show single- and multi-threaded clients, where the latter uses threads to overlap waiting for QAT with useful work. When running AES128-CBC-HMAC-SHA1 (AES in cipher block chaining mode, authenticated by SHA1 hash-based message authentication code), AES-NI does not accelerate the SHA1 computation. Thus, single-threaded QAT throughput is 2.7x lower than AES-NI, but 128-thread QAT outperforms AES-NI by 4.5x. In contrast, for AES128-GCM (AES in Galois/counter mode), single-threaded QAT throughput is 12.5x lower than AES-NI, and 128 QAT threads sharing the core only yield comparable throughput to single-threaded AES-NI. An actual application might require substantial re-engineering to support this level of threading.

2.4 Dependent NIC Offloads

As opposed to dedicated off-CPU accelerators, NIC offloads impose neither additional data transfers, nor more CPU work. CPUs operate NICs in any case, and data flows through them in any case, making them ideally positioned for data offloading. The problem is that all existing L5P NIC offloads [11, 22, 48] are *dependent*. Namely, they require the NIC to handle the L5P, which then requires the NIC to also implement the underlying layer≤4 functionality in hardware, including TCP/IP and related subsystems like firewalls and tunneling.

Dependent offloading is thus undesirable. Whereas data functions (like CRC) are relatively simple and well-suited for hardware, TCP/IP stacks are complex, evolving, interact with many OS subsystems, and incur considerable maintenance costs. To illustrate, Figure 3 shows the yearly number of lines of code (LoC) in Linux's TCP/IP stack: modified and in total. The code is constantly changing, with 5–25% LoC modification in each component, each year, for

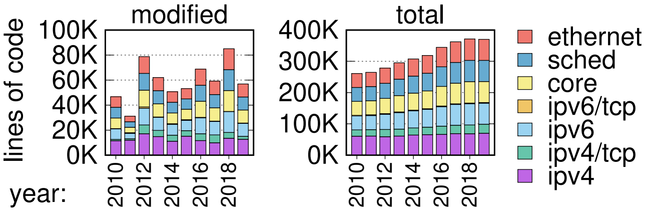


Figure 3: *Per-year Linux kernel LoC of TCP/IP processing code.*

the past decade. Having to additionally support NIC-based TCP/IP stacks would further increase this maintenance burden.

For these reasons, Linux kernel engineers resist supporting existing NIC TCP offload engines (TOEs) [25, 69]. Microsoft has deprecated TOE support for similar reasons [84]. And several operators recommend disabling TOEs due to performance issues and incompatibilities with OS interfaces [88, 89, 122].

Importantly, TOEs hinder innovation and are ill-suited for users who develop their network stack [115]. For instance, Netflix has made the following statement regarding TOEs [29]:

> *"TOEs are not a preferred solution for Netflix content delivery because we innovate in the protocol space [to improve] our customers' quality of experience (QOE). We have a team of people working on improvements to [the OS's] TCP and they have achieved significant QOE gains [...]. With the TCP stack sealed up in an ASIC, the opportunities for innovation [...] are quite limited. We also have concerns around potential security issues with TOE NICs."*

The security concerns arise as TCP/IP stacks are complex and might have security bugs [53, 102], which can be easily and quickly fixed/hot-patched in software, but not in hardware.

We remark that we focus on TCP, as it is the most widely used protocol, and it handles reordering and loss in byte streams, which is *the* challenge for autonomous offloads. But simpler level-4 protocols are also in scope. For example, FlexNIC [56] dependently offloads with DCCP [60], implementing, e.g., DCCP's acks logic in the NIC; an autonomous offload would utilize the OS logic instead.

2.5 Price Considerations

When considering to offload some protocol functionality, it makes sense to evaluate the cost of the relevant alternatives and see which is preferable in terms of price, with the overall goal of maximizing performance per dollar [22, 23, 38, 64, 87].

The focus of this study is exploiting NIC offloads that are implemented with application-specific integrated circuit (ASIC). We find that such offloads are cost-effective for clients relative to any potential alternative, because commercial NIC pricing data indicates that clients get ASIC NIC offloads essentially for free.

To back the above claim, Figure 4 shows the prices of different Mellanox NIC generations in the last decade, as specified in the Mellanox website [80]. Each NIC generation features additional offloads, listed in Table 2. The figure uses different colors for different NIC generations. It reveals that prices are typically determined by the NIC's throughput and number of ports, such that NICs from different generations usually have a similar price if they agree on these two properties. Price similarity exists despite the fact that offload capabilities substantially improve across generations, so customers do not have to pay more to enjoy additional offload capabilities.

gen.	*year*	*added offloads*
3 [74]	2011	stateless checksum [14], Large Segmentation Offload (LSO) for TCP over VXLAN and NVGRE [21]
4 [76]	2014	Large Receive Offload (LRO) [83], Receive Side Scaling (RSS) [82], VLAN insertion/stripping [18], accelerated receive flow steering (ARFS) [124], on-demand paging (ODP) [66], T10-DIF signature offload (T10-DIF) [93]
5 [77]	2016	header rewrite [72], adaptive routing for RDMA [85], NVMe over fabric [67], host chaining support [59], MPI tag matching and rendezvous [73], UDP Segmentation Offload (USO) [9]
6 [78]	2019	block-level AES-XTS 256/512 bit [42]

Table 2: *Generations of the Mellanox ConnectX NIC over the last decade, and some of the offload capabilities they introduced.*

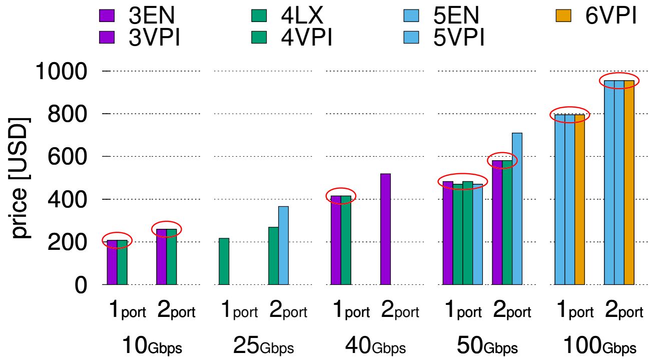


Figure 4: *Prices of standard ConnectX NICs available via Mellanox's pricing list as of March 2020 [81]. The legend numbers (3–6) indicate the ConnectX NIC generation. EN and LX models support Ethernet, and VPI models support both Ethernet and InfiniBand. Prices of older NICs are typically similar to prices of newer NICs that agree on throughput and number of ports (ellipses), even though the latter provide additional offload capabilities.*

We note that, for readability, we omit prices of (1) NIC bundles that add some hardware component to the basic NIC, (2) NICs suitable for the Open Compute Project board [26], and (3) NICs that have PCIe connectivity far exceeding their throughput. Including these does not change the conclusion.

3 AUTONOMOUS OFFLOADS

Autonomous offloads consist of a software/NIC architecture for moving data between L5P memory and TCP packets, optionally transforming or computing over the data. This architecture offloads data-intensive processing to the NIC, without having to migrate the entire TCP/IP stack into the NIC. Specific offload capabilities are cast into NIC silicon and are available for relevant L5P software as a NIC feature.

Autonomous offloads target L5P software that can communicate directly with the NIC driver, e.g., in-kernel L5Ps or userspace TCP/IP stacks. High-performance L5P software already adopts this type of design (§2.1). The main idea of the L5P-NIC collaboration

property	*limits the offload of, e.g.*
size-preserving on transmit	encapsulation and compression
incrementally computable	cipher block chaining (CBC)
constant size state	none
plaintext magic pattern	SSH's encrypted headers

Table 3: *Summary of autonomous offload properties and examples of limitations. We are not aware of any non-constant size state computation/protocol.*

is to process L5P messages in the NIC transparently to the intermediating TCP/IP stack. The NIC performs the offload on in-sequence packets, with some help from L5P software on out-of-sequence packets.

Offloading is possible for operations and L5Ps that satisfy certain preconditions (summarized in Table 3). Offloadable operations must be *size-preserving* for seamless interoperation with software TCP/IP on transmit, while on receive we can work around this precondition in some cases (§3.1). Offloadable operations must be *incrementally computable* over any byte range of an L5P message, given only some *constant-size state* and access to packet data (§3.2). In particular, the offload cannot assume L5P message alignment to TCP packets. Offloadable L5P messages must contain *plaintext magic pattern and length* fields to identify and track messages speculatively on the wire. The offload will rely on these fields to recover after loss and reordering on receive (§3.3).

These preconditions are satisfied by most of the common data-intensive operations [55]: (1) copying, (2) encryption and decryption, (3) digesting and checksumming, and some (4) deserialization and (5) decompression methods. Most L5Ps meet our requirements as well. Examples include (1) HTTP/2 [106] (encryption/deserialization/decompression); (2) gRPC [32] and Thrift [116] (copy/deserialization); (3) iSCSI [2], NBD [10], and SMB [28] (copy/encryption/digest); (4) Memcached [24] and MongoDB [52] (copy).

The following presents the high-level ideas of autonomous offloading and its preconditions. We detail the design in §4.

3.1 Data manipulation

An operation can be offloaded on one or both of the send/receive paths. To offload an operation when sending, L5P software "skips" performing the offloaded operation, thereby passing the "wrong" bytes down the stack to the NIC. The NIC performs the operation, resulting in a correct message being sent on the wire. For TCP-transparency, we require the offloaded operation to be *size-preserving*: it must never add or remove bytes from the stream. Were the operation to add/remove bytes from the stream, the NIC would have to handle these bytes' retransmission, acknowledgment, congestion control, etc., as the OS TCP stack is unaware of these bytes (Figure 5).

When receive offloading, the NIC parses incoming L5P messages within TCP packets, performs the offloaded operation, and passes packets pointing to partially processed L5P messages up the stack to L5P software. For TCP-transparency, we must preserve packet sizes observed by TCP, which is simple for size-preserving offloads. But, in contrast to the send side, receivers can offload non-size-preserving operations by DMAing offload results to pre-allocated L5P destination buffers while also DMAing the original unmodified data to the NIC driver receive ring. Consequently, receive-offload can be non-size-preserving, provided that L5P software can predict its output's size and prepare buffers for it.

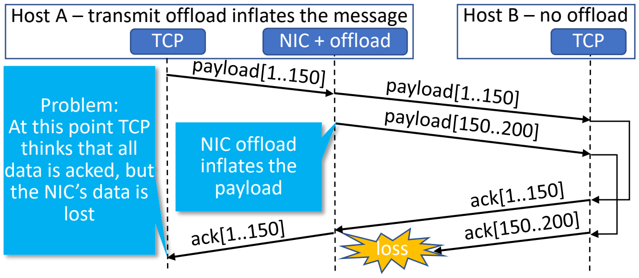

Host A – transmit offload inflates the message
TCP
NIC + offload
Host B – no offload
TCP
payload[1..150]
payload[1..150]
payload[150..200]
Problem: At this point TCP thinks that all data is acked, but the NIC's data is lost
NIC offload inflates the payload
ack[1..150]
ack[1..150]
ack[150..200]
loss

Figure 5: *The problem with non-size-preserving offloads.*

3.2 In-Sequence Packet Processing

We require that the offloaded operation can be performed over any byte range of an L5P message (i.e., in-order TCP packets of any size), given only some constant-size state. The state is composed of *dynamic* and *static* components. The dynamic state is a function of (1) the previous bytes in the current message and (2) the number of previous messages. It is (conceptually) updated after each packet (byte) is processed. The static state is fixed per-request or per-connection, e.g., TLS session keys or NVMe-TCP host read response destination buffers.

The above properties allow the NIC to perform the offload without having to buffer packets until obtaining a full L5P message, which is impractical (e.g., because messages can be big, potentially exceeding the TCP receive window). Specifically, the NIC maintains per-flow contexts (for both outgoing and incoming flows) holding the state required to perform the operation on the next *in-sequence* TCP packet. Once that packet is handled, the NIC updates the flow's (dynamic) state.

Our requirements are satisfied by most L5P data-intensive operations, which typically process the current L5P message independently of the payload of previous messages [17, 32, 92, 107, 111, 116]. The requirements mainly preclude offloading of operations such as AES cipher block chaining (CBC), which operate on fixed blocks and not an arbitrary range. However, modern ciphers, such as AES-GCM and ChaCha20-Poly1305, satisfy our requirements.

3.3 Out-of-Sequence Packet Processing

Receive: The NIC cannot perform the offloaded operation on an out-of-sequence packet. It also cannot buffer it until the in-sequence packet arrives. Instead, we fall back on L5P software to perform the operation: an out-of-sequence packet is passed to the OS, and L5P software performs the operation on its bytes when it receives the message containing the packet. Such messages, in which the operation was offloaded on some/none of the bytes, are called *partially/fully un-offloaded.* To resume offloading, the NIC *resynchronizes* itself into knowing the next expected TCP packet and its associated operation state, as explained next.

Resync: We require that L5P message headers contain (1) at least one *magic pattern* plaintext field that identifies a message header on the wire and (2) a length field. Combined, these properties enable a

```
l5o* l5o_create(sock*, l5_state*, u32 tcpsn)
void l5o_destroy(l5o*)
rr_state_id l5o_add_rr_state(l5o*, rr_state*)
void l5o_del_rr_state(l5o*, rr_state_id)
void l5o_resync_rx_resp(l5o*, u32 tcpsn, bool res)
```

Listing 1: *Operations the NIC driver provides to the L5P.*

```
l5_msg_state* l5o_get_tx_msgstate(sock*, u32 tcpsn)
void l5o_resync_rx_req(sock*, u32 tcpsn)
```

Listing 2: *Operations the L5P provides to the NIC driver.*

hardware-driven NIC context resynchronization process, which begins when the NIC receives out-of-sequence data and loses track of the flow's state. To resynchronize, the NIC speculatively identifies an L5P message in the incoming byte stream by the magic pattern, and confirms this identification with the L5P software. While waiting for the L5P's reply, the NIC keeps track of incoming messages by using the L5P header length field to derive the TCP sequence number of the next expected message (where another magic pattern should appear). Once the L5P confirms the speculation, the NIC can resume offloading from the next L5P message, since the dynamic state at message boundary depends only on the number of previous messages, which is included in the L5P's confirmation (see §4.3).

Transmit: To perform the offloaded operation on a retransmitted outgoing packet, the NIC's dynamic state is *recovered* to the correct state for that packet by the NIC driver with the help of the L5P software. To this end, the L5P software must store the state for an L5P message until the TCP acknowledgment of all of its packets.

4 DESIGN

This section describes the software and hardware designs that together form the autonomous NIC offloads architecture. We first describe the software and hardware interfaces (§4.1), followed by handling of transmitted (§4.2) and received (§4.3) packets, for both in-sequence and out-of-sequence (OoS) data.

4.1 Interfaces

The NIC maintains a per-flow *HW context*, which holds the state required to perform the offloaded computation for a specific packet of the flow (typically, the next in-sequence packet). Each HW context contains: (1) *tcpsn*, the TCP sequence number that this context can offload; (2) the L5P message type, length, and offset within the message at *tcpsn*; and (3) L5P state required to perform the offload, such as cipher keys. A context also stores a flow identifier, e.g., a TCP/IP 5-tuple.

L5P–NIC driver interface The NIC driver provides an interface to the L5P software for context creation, destruction, and recovery (Listing 1). After the L5 handshake is complete, the L5P calls *l5o_create* with the inputs required to process the next message in the stream (*l5_state*), and the TCP sequence number of the first byte in that message (*tcpsn*). To stop the offload, the L5P calls the *l5o_destroy* method. Offloading is typically terminated when the socket is destroyed.

In Request-Response (RR) protocols, offloading the computation for an incoming message (a response) requires the NIC to associate the message with the request that triggered it. For example, an offload copying the response payload directly to an application buffer needs to know the buffer's address. To this end, the NIC can internally map incoming messages to the required offloading state, which is configured using the *l5o_add_rr_state* method. The L5P provides this state to the NIC before sending the request, and deletes it after the response is received using the *l5o_del_rr_state* method.

The *l5o_resync_rx_resp* method is used for receive-side context recovery from OoS packets (§4.3). The L5P also provides an interface to the driver (Listing 2) for context recovery, which we discuss in §4.2 and §4.3.

Driver–NIC interface Offload-related commands are passed to the NIC via special descriptors, which are placed into the flow's usual send ring to ensure ordering. The NIC passes information to the driver through descriptors in the flow's receive ring. Both rings are accessed through DMA.

4.2 Transmitted Packets Processing

To send application data, the L5P encapsulates it into L5P messages, preserving all fields of the message that appear on the wire, including fields that are filled by the offload (e.g., CRC). The L5P then hands the data for transmission to the next layer protocol. Typically, this protocol is TCP, but it can also be an L5P (e.g., TLS), as our offloads compose (§5.3).

When the NIC driver is handed a TCP packet for transmission, it must figure out if the packet is in- or out-of-sequence with respect to the NIC's flow context. To this end, it extracts the context ID from the packet's metadata; this ID is passed down from the L5P, which obtained it on context creation. The driver shadows the NIC's context in software, and so it can check the packet's TCP sequence against the context's expected TCP sequence to identify OoS packets. If the packet is OoS, the driver recovers the NIC's context, as described below. Next, the packet is posted to the NIC's send ring, tagged with the HW context ID (which saves the NIC from looking up the HW context based on the packet fields). Finally, the NIC performs the offloaded operation and sends the packet.

Context recovery for OoS data To enable offloading of transmitted OoS data, the driver recovers the NIC's context to match the packet. As noted in §3, we assume that the state required to perform the offload can be obtained from the packet's L5P message (Figure 6). The driver obtains this state using the *l5o_get_tx_msgstate* upcall to the L5P, and then passes it to the NIC via a special descriptor. The driver also updates the context's expected TCP sequence (in both HW and its shadow) to match the packet's TCP sequence.

To answer *l5o_get_tx_msgstate* calls, the L5P software must maintain a map from TCP sequence numbers to their corresponding L5P messages (in our experience, this takes ≈ 200 LoC). The L5P holds a reference to the buffers which contain transmitted L5P message data, similarly to how TCP holds a reference to all unacknowledged data. The L5P releases its reference when the entire message is acknowledged.

4.3 Received Packets Processing

The NIC only performs the offload for in-sequence packets. OoS packets are handled by software. When a packet with a valid TCP/IP

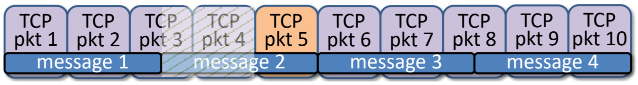


Figure 6: *Packet 5 is retransmitted; the data required to offload it is marked with a diagonal pattern.*

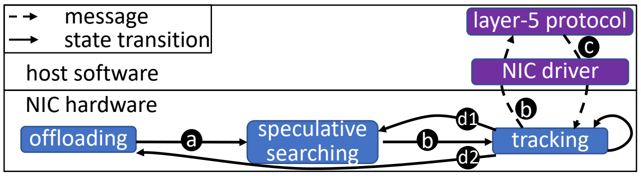


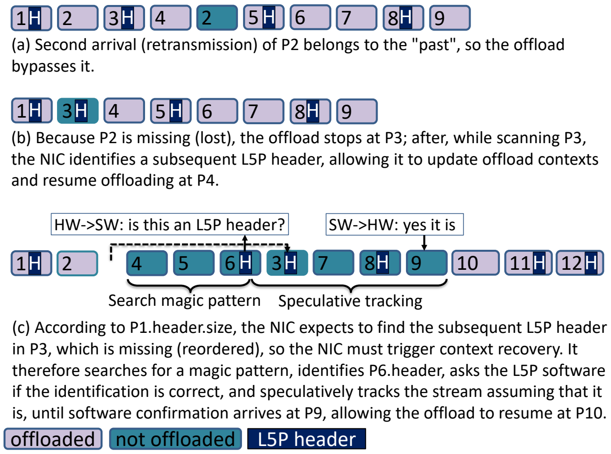
Figure 7: *State-machine for L5P offload recovery from reordering.*

checksum arrives, the NIC looks up its flow's context.[1] If a context is found, the NIC performs the offloaded operation if the TCP sequence numbers of the context and the packet match. Both offloaded and un-offloaded packets are passed to the driver, with an indication (in their descriptors) of whether offloading was performed. The driver passes the packet and the offload result as metadata up to the network stack, which takes care not to coalesce packets with different offload results. The L5P software reads L5P messages handed to it by TCP packet-by-packet and skips computing the offloaded function if all packets are offloaded. Otherwise, the L5P must perform the relevant data manipulation itself.

Out-of-sequence packets The NIC never performs the offload on an OoS packet, but it processes such packets in an attempt to get back in sync with the TCP stream. The NIC cannot wait for the packet Q with the sequence number it expects to arrive, because that would require buffering all the flow's packets that arrive in the meantime. (Without such buffering, packets with sequence numbers higher than Q may reach the OS TCP stack while the NIC waits for Q, leaving it unaware of the next expected sequence number on the wire.) We thus need to resync without waiting for Q to arrive.

It follows from our requirement that offload state depend only on the previous bytes of a message and on the number of previous messages (§3.2) that the NIC can resync itself once it sees the *next* L5P message. Thus, when an OoS packet P arrives, the NIC computes the TCP sequence number of the next L5P message M by using the length of the current message (which is stored in the context). If the TCP sequence number of P, $P.seq$, is before M's sequence number, $M.seq$, then P is ignored. If P contains M's header, the context is updated to M, so that the offloading can resume for the packet following P. Otherwise ($P.seq$ is after $M.seq$), the NIC cannot resync, as it does not know which (if any) L5P messages appeared after M. In this case, the NIC begins a context recovery process in collaboration with the L5P software.

Context recovery A naive *software-driven* approach for context recovery is for L5P software to inform the NIC about the TCP sequences numbers of the messages it receives, thereby allowing the NIC to resync. However, such a scheme is inherently racy: by the time the NIC hears about a message, it may have already started receiving packets of subsequent messages. As a result, the NIC may never be able to successfully recover its context. To avoid this problem, our design employs a *hardware-driven* recovery process, in which the NIC speculatively identifies arriving messages and relies on software to confirm its speculation.



(a) Second arrival (retransmission) of P2 belongs to the "past", so the offload bypasses it.

(b) Because P2 is missing (lost), the offload stops at P3; after, while scanning P3, the NIC identifies a subsequent L5P header, allowing it to update offload contexts and resume offloading at P4.

(c) According to P1.header.size, the NIC expects to find the subsequent L5P header in P3, which is missing (reordered), so the NIC must trigger context recovery. It therefore searches for a magic pattern, identifies P6.header, asks the L5P software if the identification is correct, and speculatively tracks the stream assuming that it is, until software confirmation arrives at P9, allowing the offload to resume at P10.

Figure 8: *NIC processing of various OoS packets received from the wire: (a) retransmission, (b) L5P data reordering, and (c) L5P header reordering which triggers OoS recovery.*

The recovery algorithm is depicted in Figure 7. Initially (transition ⓐ), the NIC enters a *speculative searching state*. In this state, whenever a valid TCP packet arrives, the NIC searches for the protocol's header magic pattern (§3) in the TCP payload. When found, the NIC requests the software L5P to acknowledge the detected message header TCP sequence number (*tcpsn*) via the NIC driver, which calls *l5o_resync_rx_req* to register the request with the L5P. The NIC also transitions to the *tracking state* (ⓑ). The L5P stores *tcpsn* and waits until the corresponding message is received from the OS TCP stack. The L5P then notifies the NIC whether the message's TCP sequence number matches the NIC's "guess," using the *l5o_resync_rx_resp* method (ⓒ).

Meanwhile, the NIC tracks received messages using the message header's length field, verifying that each subsequent message begins with the magic pattern. If an unexpected pattern is encountered or the L5P response indicates that the NIC misidentified a message header, then the NIC moves back to the speculative searching state (d1). If the L5P response indicates success while the NIC is in the tracking state, then the NIC can resume offloading from the next message (d2).

Example Figure 8 depicts the various cases of receive packet processing. If packets arrive in-sequence then all are offloaded. Otherwise, we have the following cases: (a) retransmitted packets (P2) bypass offload and do not affect NIC state; (b) L5P data reordering within the current message (P2). The NIC identifies the next L5P message header (P3), updates its context to expect P4 and continues processing from there. The packet containing the L5P header is not offloaded as it does not match the expected TCP sequence number, but the following packet (P4) does match it; (c) L5P header

[1] Similar hardware functionality already exists for LRO and ARFS.

reordering (P3) causes the NIC to cease offloading. Then, it searches and finds an L5P message header magic pattern (P5) and it requests software to confirm its speculation. We note that it can identify patterns split between packets if they arrive in-sequence. Meanwhile, NIC HW tracks subsequent L5P headers (P8) using header length fields, and verifies their magic pattern. Eventually, L5P software receives the NIC HW request and the corresponding packet after TCP processing, and then L5P software confirms NIC hardware speculation. Finally, offload resumes on the next packet boundary.

5 IMPLEMENTATION

Here, we describe case studies of autonomous offloads targeting in-kernel L5Ps in Linux: NVMe-TCP (§5.1), TLS (§5.2), and their composition (§5.3). The offloads described in this section are (or will be) available in Mellanox ASIC NICs.

5.1 NVMe-Over-TCP Offload

NVMe-TCP [92] is a pipelined L5P which abstracts remote access to a storage controller, providing the host with the illusion of a local block device. In NVMe-TCP, each NVMe [91] submission and completion queue pair maps to a TCP socket. Read/write IO operations use request/response messages called *capsules*, whose header contains a (1) capsule type, (2) data offset, (3) data length, and (4) capsule identifier (CID). The CID field is crucial to correlate between requests and responses, as the controller can service requests in any order. Also, capsules can be protected by a trailing CRC.

Offloaded data manipulations We offload the two dominant data manipulation operations of the protocol: CRC32C [113] data digest computation/verification (on transmit/receive) and capsule data copy from TCP packets to block layer buffers (on receive). Note that this copy cannot be avoided with standard zero-copy techniques, as (1) the OS cannot make the NIC DMA directly into application or page cache buffers, since the OS does not know ahead of time which receive ring entry corresponds to which NVMe-TCP response; and (2) even if that were possible, packets contain capsule headers, which do not belong in block layer buffers.

Copy offload The NIC stores a map from CIDs to the corresponding block layer buffers. The map is updated by the NVMe-TCP before it sends a read request. When a response arrives, the NIC DMA writes the capsule payload to the block layer buffers for each offloaded packet, while placing the packet and capsule headers/-trailers in the NIC's receive ring (see Figure 9). These receive packet descriptors provide all the information necessary to construct a socket buffer (SKB) that points to the received data, including the block layer buffer. When this SKB reaches the NVMe-TCP code responsible for copying capsule payload data to the block layer buffer, the copy is skipped, as the relevant `memcpy` source and destination addresses turn out to be equal. This means that partially- or un-offloaded capsules are handled transparently, with the `memcpy` performed as usual for the remaining un-offloaded parts.

CRC offload On transmit, NVMe-TCP prepares capsules with dummy CRC fields, which the offload fills based on the digest of capsule data of previous in-sequence packets. OoS packets are handled as described in §4.2. On receive, the NIC checks the CRC of all capsules in the TCP payload of in-sequence packets. It reports a single bit to the driver (along with the packet descriptor), which is set if and only if all capsules with the packet pass the CRC check. The driver sets a *crc_ok* bit in the SKB of the received packet[2] according to the NIC's indication, and hands the SKB to the network stack. When NVMe-TCP receives a complete capsule, it skips CRC verification if the *crc_ok* bits of all SKBs in the capsule are set. Otherwise, it falls back to software CRC verification. Partially- or un-offloaded capsules are thus handled easily.

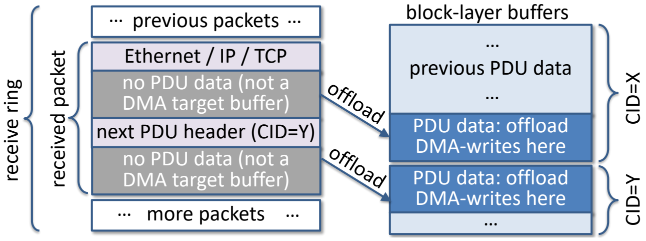


Figure 9: *NVMe-TCP receive offload. The data is written directly to the block layer according to the CID, instead of to the receive ring.*

Magic pattern For speculative searching, we rely on a number of fields from the NVMe-TCP capsule header and trailer to form the pattern and verify it: (1) PDU type: one of only eight valid values (1 byte); (2) header length: well known constant for each PDU type (1 byte); (3) header digest: optional CRC32 digest of the header (4 bytes); and (4) data digest: optional CRC32 digest of the data (4 bytes).

Implementation effort Our patches modifying Linux to support the NVMe-TCP offload are under review by the relevant maintainers. The changes in NVMe-TCP are 418 LoC, and another 1755 LoC in the Mellanox NIC driver.

5.2 TLS Offload

The Transport Layer Security (TLS) protocol is an L5P that protects the confidentiality and integrity of TCP session data [17, 107]. A TLS session starts by exchanging keys via a handshake protocol, after which all data sent/received is protected with a symmetric cipher, such as AES-GCM [50].

Application typically use a library that implements TLS. We modify the popular OpenSSL library to use the Linux kernel's TLS (KTLS) data path, which can leverage our offload. OpenSSL's TLS handshake code remains unmodified.

TLS messages are called *records* and are at most 16 KiB in size. A TLS record consists of a header, data and a trailer. The header holds four fields of interest: (1) record type, (2) version, (3) record length, and (4) initialization vector (IV), used by the cipher. The trailer holds the integrity check value (ICV) of the entire record. For each socket send (receive), KTLS encapsulates (decapsulates) the data into records.

Our offload is motivated by TLS 1.3 [107], which support two symmetric ciphers: AES-GCM and Chacha20/Poly1305. We offload AES-GCM [50], as it is the most common TLS cipher [58, 62, 63, 120, 121].

[2] This requires adding a new bit to the SKB.

static	*dynamic*
TLS protocol version	current record length
encryption key	current record IV
encryption salt	current record sequence no.
software response TCP.seq	offset in current record
	next expected TCP.seq
	offload state (search/track/offload)
	last 16 bytes for ICV calculation
	hardware speculative TCP.seq

Table 4: *Hardware TLS offload contexts. Static state is configured once and dynamic state changes per-packet.*

Crypto offload On transmit, KTLS prepares plaintext records with dummy ICV fields, and the NIC replaces plaintext with ciphertext and fills the ICV. OoS packets are handled as described in §4.2. On receive, the NIC decrypts the payload of each offloaded received packet, and it checks all ICV values within the packet. It reports the result in a single bit to the driver (along with the packet descriptor). The driver sets a *decrypted* bit in the received packet's SKB according to the NIC's indication, and hands the SKB to the network stack. When a complete record is received by KTLS, it skips decryption and authentication if the *decrypted* bits of all SKBs in the record are set. Otherwise, KTLS falls back to software decryption and authentication.

Zero-copy sendfile support KTLS supports the `sendfile` system call. While `sendfile` is typically implemented without copying, KTLS cannot encrypt transmitted page cache buffers in-place, as that would corrupt their content. Instead, standard KTLS `sendfile` encrypts sent data in a separate buffer, allocated for this operation. Our offload *enables skipping this costly allocation*, as KTLS can hand the page cache buffers to the NIC, which encrypts them to the wire instead of in-place. As a result, KTLS with our offload can achieve performance comparable to plain TCP `sendfile` (see §6.3). However, the user becomes responsible for not changing files while they are transmitted.

Partial offload Conceptually, the software fallback for partially-offloaded records is to decrypt the non-offloaded packets and authenticate the record while reusing offload results. However, AES-GCM authentication is computed on the ciphertext data, and so performing authentication in software requires re-encrypting the packets decrypted by the NIC. Consequently, handling partial decryption is costlier than full decryption (see §6.4).

Magic pattern For speculative searching, we rely on a number of fields from the TLS record header to form the pattern and verify it: (1) record type (1 byte): one of only six valid values[3]; (2) record version (2 bytes): the version is constant after the TLS handshake; and (3) record length (2 bytes): this field must be less than 16 KiB.

Software implementation Our OpenSSL changes adding KTLS support consist of 1381 LoC. Offload support in KTLS is 2223 LoC. Offload support in the Mellanox NIC includes 2095 LoC. Our changes have been accepted for inclusion in OpenSSL [96–98] and Linux [8, 43], indicating the relevance of the offload. Others have added KTLS offload support to FreeBSD [6].

Hardware implementation To minimize hardware overheads, the TLS offload extends existing units and interfaces that compute over in-sequence data, such as RDMA T10-DIF [93] and AES-XTS offloads [42]. As a result, the hardware overhead is negligible.

Table 4 presents the main fields in TLS hardware contexts, which reside in NIC memory (SRAM). TLS fields (version, keys, etc.) are also stored in software TLS, while TCP fields (next expected TCP sequence) are also stored in software TCP. The NIC driver stores only the next expected expected TCP sequence to identify when context recovery on transmit is needed.

To offload TLS on transmitted packets, the NIC driver provides a context identifier (ID) alongside packet data to hardware that uses IDs to fetch the corresponding contexts. Then, using static and dynamic contexts, the NIC encrypts and authenticates data in-place, updates the dynamic context data, and sends data onward for TCP checksum calculation and transmission on the wire.

To offload received TLS packets, the NIC hardware looks up the contexts using software defined selectors (e.g., 5-tuple). Then, it verifies the TCP checksum, and compares the packet TCP sequence number against the expected TCP sequence stored in the dynamic context. Then it proceeds to replace data in-place with decrypted data and fills packet completion metadata to indicate if all TLS records within the packet are authenticated successfully.

To recover the context on transmit, the NIC driver queues a number of special transmit descriptors with fences before queueing the packet data descriptor to the same queue. These descriptors: (1) reset the dynamic context according to that of the OoS packet and its TLS record; and (2) feed the TLS AES-GCM engine with the OoS packet's TLS record prefix data obtained from software TLS.

To recover the context on receive, the NIC hardware marks in packet completion that the packet contains a speculatively identified magic pattern. Thereafter, to confirm NIC speculation, the driver updates the static context's "response TCP sequence number" field with the TCP sequence number of the confirmed speculatively identified message.

5.3 NVMe-TLS Offload

Combining NVMe-TCP and TLS offloads is simple, as the layering determines their ordering. NIC HW parsing starts from Ethernet, and proceeds to parse TLS then NVMe-TCP on transmit and receive. In-sequence packet processing remains the same, where each offload is processed independently: on transmit we do NVMe-TCP then TLS; and on receive vice versa. Transmit and receive OoS context recovery are performed independently for each protocol.

6 EVALUATION

Using microbenchmarks, we measure the overhead of the data-intensive operations that our NIC autonomously offloads (§6.1). We then evaluate actual offload performance with macrobenchmarks (§6.3), and we quantify the effect of out-of-sequence TCP packets (§6.4). Finally, we examine the performance of autonomous NIC offload at scale (§6.5).

TLS results are obtained using *real* Mellanox ConnectX6-Dx ASIC NICs. NVMe-TCP results are obtained via *emulation*, as this offload will only become available in Mellanox's next-generation

[3] HW can store an extensible list of these values.

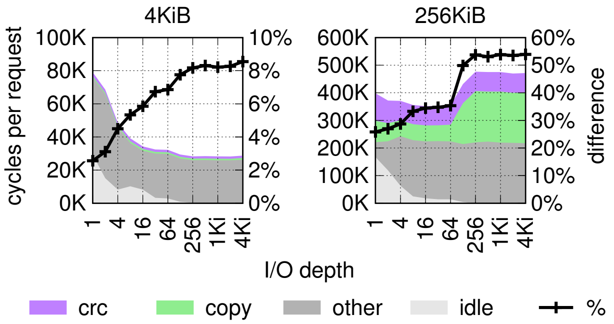


Figure 10: *NVMe-TCP/fio cycles per random read on the server (drive resides on the generator); "%" shows copy+crc out of the total.*

ConnectX-7 NICs. We validate the accuracy of our emulation methodology by comparing the performance of real and emulated TLS offloading (§6.2).

Our setup consists of a Dell PowerEdge R730 server and an R640 workload generator. The server has two 14-core 2.0 GHz Xeon E5-2660 v4 CPUs, and the generator has two 12-core 2.1 GHz Xeon Silver 4116 CPUs. Both have 128 GB (=4x16 GB) memory, and they run Ubuntu 16.04 (Linux 5.6.0) with hyperthreading and Turbo Boost off to avoid nondeterministic effects. For storage, the server utilizes an Optane DC P4800X NVMe SSD that resides remotely, on the generator.

The machines are connected back-to-back via 100 Gbps Mellanox ConnectX6-Dx NICs that implement our TLS AES128-GCM crypto autonomous offload.

All the results presented in this section are trimmed means of ten runs; the minimum and maximum are discarded, and the standard deviation is below 3% unless specified otherwise.

6.1 Cycle Breakdown

NVMe-TCP When NVMe-TCP reads from a remote drive, recall that it accesses the received bytes twice: (1) when copying them from the network buffers to their designated memory locations; and (2) when computing the incoming capsule's CRC. Figure 10 shows how long these two operations take out of the total of an individual I/O request. We use fio [5] to generate random read requests of different sizes (title of subfigures) and to vary the number of outstanding requests (I/O depth along the x axis). The left and right y axis labels show per-request duration in cycles and the relative cost of the copy+CRC overheads out of the total, respectively. The system is limited to using a single core for all of its activity.

We can see that smaller requests have a potential improvement of 2%–8%, and bigger requests have a potential improvement of 25% (lower parallelism, up to depth=64) to 55% (higher parallelism, as of depth=128). In the latter case, the 32 MiB LLC becomes too small to hold the working set (128 requests times 256 KiB per request = 32 MiB). From this point onward, copying becomes the dominant overhead, as every memory access is served by DRAM.

TLS We similarly measure TLS's offloadable overheads: encryption, decryption, and authentication, which we collectively denote as "crypto" operations. For this purpose, we use iperf [123], which measures the maximal TCP bandwidth between two machines, and which we modified to support OpenSSL. We use 256 KiB messages at the sender and ensure that the server's core always operates at 100% CPU. Recall that each message consists of a sequence of TLS records, which can be as big as 16 KiB.

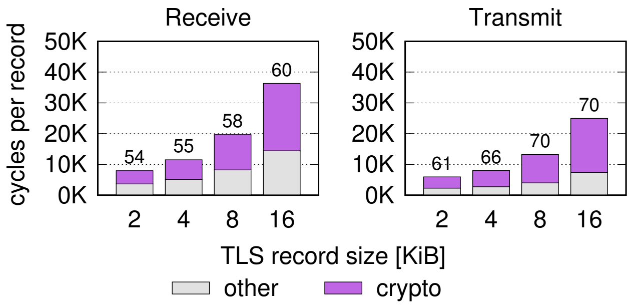


Figure 11: *Kernel-TLS/iperf per-record cycles when using AES-GCM crypto operations (encryption, decryption, and authentication using AES-NI); standard deviation is between 0%–8.2%.*

Figure 11 shows the results. Unsurprisingly, bigger TLS records reduce the weight of network stack processing relative to the crypto operations, making the potential offload benefit more pronounced at the right. This outcome is consistent with the fio results. Typically, network stacks operate more efficiently when sending than when receiving, because batching is easier; the receive side has to work harder. Consequently, the potential benefit of offloading is higher for transmitting (≤74%) than for receiving (≤60%).

With real TLS offloading, we find that iperf's single core throughput improves by 3.3x and 2.2x upon transmit and receive, respectively, relative to the non-offloaded baseline. When saturating the NIC with multiple iperf instances, CPU utilization respectively improves by up to 2.4x and 1.7x.

6.2 NVMe-TCP Offload Emulation

We hypothesize that commenting out the software functionalities to be offloaded (without really implementing them in the NIC) yields similar performance to real offloading. We verify this hypothesis with TLS offloading. We find that the "other" component in Figure 11 is an accurate performance predictor for our new TLS offload capability: at most 7% difference between the real and predicted improvements in all cases.

We use this finding to emulate NVMe-TCP offloading by: (1) setting the value of all stored data to be a repetitive sequence of an 8-byte "magic" word (0xCC...CC); (2) modifying NVMe-TCP receive-side to *refrain* from copying incoming "magic capsules" (that start with the magic word) to their target buffers, and also; (3) skipping CRC computation and verification for magic capsules. Clearly, a block device driver that fails to copy device content to the designated target buffers seems problematic. We next describe how the integrity of our experiments is preserved despite this problematic behavior.

Nginx The subsequent evaluation uses two macrobenchmarks. The first is the nginx http web server [104], configured to serve files from an ext4 filesystem mounted on our NVMe-TCP block device. (Recall that the drive resides remotely, on the workload generator machine.) We pre-populate ext4 with "magic files", which exclusively contain magic word sequences. We set the size of magic files to be an integral multiple of 4 KiB, and we configure nginx clients to only request these files. We also set ext4 read-ahead to the file size, such that there are no block requests that exceed this size.

Neither the kernel of the server machine, nor nginx and its clients care about the content of the files that they send/receive. Ext4 does not collocate metadata within the 4KiB blocks of magic files, so it is indifferent to whether their content is copied to the server's page cache; nginx, which sends this page cache content to its clients, is likewise indifferent to the content; and the clients do not actually use the content either.

Redis-on-Flash The second macrobenchmark we use is Redis-on-Flash (RoF), a key-value store [45, 114] that uses RocksDB [110] as storage backend. RocksDB is incompatible with our emulation. It runs with RoF on the server and uses the NVMe-TCP block device to read and write its internal data structures, which interleave metadata and data in nontrivial ways. Consequently, magic capsules do not exist in this setup, even if the values we store exclusively consist of magic words.

To overcome this problem, with some help from Redis Labs [103] engineers, we implemented OffloadDB, a simple alternative storage backend for RoF, which does separate between keys, values, and metadata (568 LoC). Coupling RoF with an OffloadDB storage backend makes our emulation approach applicable to RoF as well.

6.3 Macrobenchmarks

As noted in §6.2, we use the nginx and RoF macrobenchmarks to evaluate the performance of our two autonomous NVMe-TCP and TLS offloads, individually and together. We begin with nginx and drive it with the wrk [30] http benchmarking tool. Wrk connects to nginx using 16 threads, which together maintain 1024 open connections that repeatedly request files of a specified size and then wait for a response. We utilize two configurations: C_1 and C_2. In C_1, none of the drive's relevant data is found in the server's page cache. C_1 stresses NVMe-TCP offloading, with a maximal possible rate of the drive's optimal read bandwidth: 2.67 GB/s (≈21.38 Gbps). In C_2, all of the drive's relevant data already resides in the server's page cache, and so it is not read from the remote drive while nginx is operational. C_2 stresses TLS offloading, with a maximal possible rate of 100 Gbps, our NIC's line rate.

Individual Offloads Figure 12a shows the http throughput of nginx in C_1, with and without the NVMe-TCP offload. We replicate the microbenchmark methodology and limit system activity at the server to one core (which becomes 100% busy as a result). The outcome turns out qualitatively similar: throughput improvements range between 4%–44% and are correlated with the size of the requested files. In Figure 12b and Figure 12c, we allow server activity to utilize up to eight cores, which is enough computational power for nginx to be able to fully utilize the remote drive's bandwidth. When maximal bandwidth is reached, NVMe-TCP offload improvements manifest in up to 27% reduced CPU consumption.

We repeat the above experiment in C_2 (data in page cache) using four different setups: (1) "https" employs the baseline, KTLS sendfile with AES-NI crypto operations without any offloads; (2) "offload" improves the baseline by adding TLS offload; (3) "offload+zc" further improves it by instructing TLS to refrain from making copies and instead send files directly from the page cache in a zero-copy ("zc") manner (making it the responsibility of users to avoid changing files while transmitted); and (4) "http" sends unencrypted text and thus serves as an upper bound on improvements.

size	*base*	*+TLS*	*+copy*	*+CRC*
4K	169 $_{\pm 0.6}$	167 $_{\pm 0.4}$ (0.99)	165 $_{\pm 0.3}$ (0.98)	165 $_{\pm 0.5}$ (0.98)
16K	221 $_{\pm 0.5}$	210 $_{\pm 0.5}$ (0.95)	204 $_{\pm 0.3}$ (0.92)	200 $_{\pm 0.4}$ (0.90)
64K	466 $_{\pm 0.5}$	396 $_{\pm 0.5}$ (0.85)	376 $_{\pm 0.3}$ (0.81)	365 $_{\pm 0.2}$ (0.78)
256K	1321 $_{\pm 5.1}$	1056 $_{\pm 0.5}$ (0.80)	980 $_{\pm 0.0}$ (0.74)	941 $_{\pm 0.4}$ (0.71)

Table 5: *Average latency in µsec for a single, synchronous request when cumulatively adding our L5P autonomous offloads. Values in parentheses show relative latency as compared to the baseline. Values to the right of the ± sign show standard deviation in percentages.*

Figure 13 shows the results. With one core, offload and offload+zc deliver 7%–70% and 11%–2.7x higher throughput as compared to https, respectively. With eight cores, they reduce CPU consumption by 0%–2% and 0%–23%, respectively. Offload+zc delivers 88% higher throughput when reaching the NIC line rate. Offload+zc throughput is within 25%–28% of http throughput with one core, and it consumes 23% more CPU cycles with eight cores when using 256 KiB files. Interestingly, for smaller files, offload+zc consumes 3% *less* CPU than http. This happens due to TCP batching effects, which cause http to utilize more, smaller packets for sending.

Overall, offloading eliminates the per-byte cost of the data manipulation, leaving only per-packet costs. This can be seen in the smaller files (size between 128B–1024B), where per-byte costs are small and so offloading yields only a small improvement of 0%–10% and 0%–4% in throughput and CPU consumption, respectively.

Combined Offloads To combine the NVMe-TCP and TLS offloads (together denoted "NVMe-TLS"), we add TLS support in NVMe-TCP Linux subsystem (210 LoC, not yet upstreamed). We evaluate nginx and RoF in the C_1 configuration. In the RoF experiment, we run one RoF instance per core and use the memtier [65] "get" workload to drive the instances with 8 concurrent request-response connections per instance.

Figure 14 shows the outcome for nginx. It is qualitatively consistent with the previous results that were bounded by the drive's bandwidth (Figure 12, which was dedicated to the NVMe-TCP offload). But the quantitative improvement of the offload combination is more substantial. For example, with a single core and an I/O size of 256 KiB, the improvement provided by the NVMe-TLS offload is 4.0x bigger than that of the NVMe-TCP offload (44% vs. 180%≡2.0x).

Figure 15 shows the benefit of NVMe-TLS offloading for RoF. When comparing it to the corresponding single-offload RoF experiment (not shown), we find that the improvement is 4.6x bigger (28% vs. 130%≡2.3x), similarly to the aforementioned nginx ratio.

So far, our workloads have been throughput-oriented. In Table 5, we show the average latency of a single http GET request (single connection) for multiple offload combinations. In particular, we cumulatively add to the baseline configuration the TLS offload, then the NVMe-TCP copy offload, and then the NVMe-TCP CRC offload. TLS symmetric crypto is much costlier than copying and

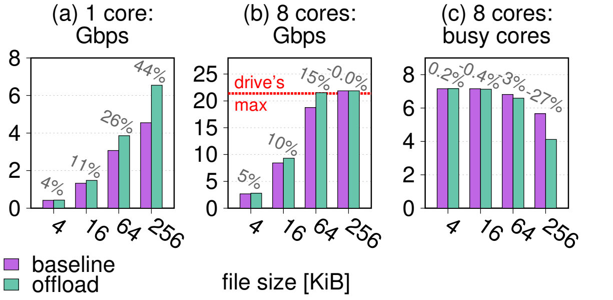


Figure 12: *Nginx improvements with the NVMe-TCP offload. None of the files reside in the server's page cache (configuration C_1), so the throughput is bounded by the drive's maximal bandwidth. Bar labels show offload improvement over the baseline.*

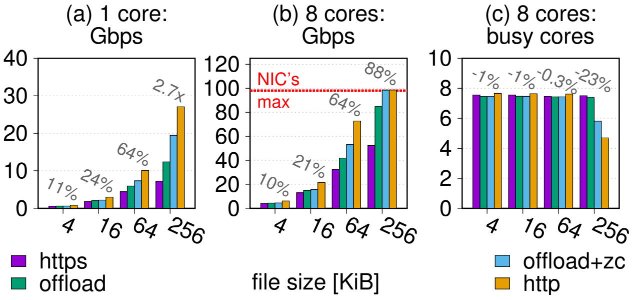


Figure 13: *Nginx improvements with two TLS offload variants compared to https (baseline encryption) and http (no encryption). All files reside in the page cache (C_2), so throughput is bounded by the NIC line rate. Bar labels show offload+zc improvement over https.*

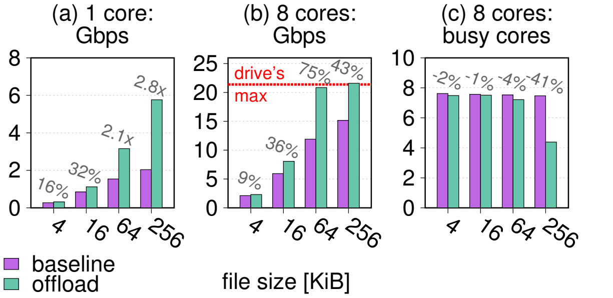


Figure 14: *Nginx improvements obtained with the NVMe-TLS combined offload.*

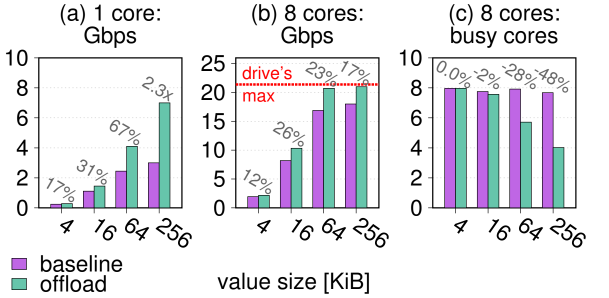


Figure 15: *Redis-on-Flash improvements obtained with the NVMe-TLS combined offload.*

CRC-ing, so the corresponding offload unsurprisingly achieves the majority of the benefit: a 1%–19% latency reduction. The NVMe-TCP offloads then reduces the latency further by 1%–9% percentage points. As before, bigger requests benefit more.

6.4 Reordering and Loss

Out-of-sequence TCP packets (caused by reordering and loss) make our autonomous offloads less effective and imply that NICs and/or CPUs must work harder. Figure 16a depicts the effect of gradually increasing packet loss rate on a single sender core transmitting through 128 iperf streams at 100% CPU utilization. We use loss rates between 0%–5% because on the internet, loss rate is typically ≤2% [20] and reordering is likewise ≤2% [94]. (In datacenters, loss can largely be avoided with DCTCP [3]). On transmit, we can see that TLS offload performance is close to regular TCP performance, delivering throughput that is within 8%–11% of plain TCP. The benefit of offloading compared to software TLS becomes smaller as loss increases, but it nevertheless remains non-negligible, with a minimal 33% improvement at 5% loss. Figure 16b reports the internal interconnect bandwidth that the NIC consumes when reading data from memory to reconstruct its contexts (in percents out of the total gen3 x16 PCIe available bandwidth). The figure shows that even with 5% loss, context recovery costs no more than 2.5% of total PCIe bandwidth.

Figure 17a and Figure 18a show the results of conducting the same experiment but with a receiver using loss and reordering, respectively. Loss and reordering are much costlier at the receiving end when offloading, which is why the corresponding curves rapidly get closer to the software TLS curve. Recall that each out-of-sequence (reordered or retransmitted) packet implies that the encapsulating TLS record will not be offloaded. Figure 17b and Figure 18b classify TLS records into three: offloaded (no packet in the record was out-of-sequence), partially offloaded (some were out-of-sequence), and not offloaded. Even with 5% loss, we see that more than half of the records are fully offloaded, which highlights the effectiveness of the NIC's context recovery. Figure 17a indeed shows that offloading still yields a non-negligible throughput improvement of 19% with the highest packet loss rate. With reordering of even 2% , unlike loss, we see that only 24% of the records are fully offloaded, and with 5% almost no TLS record is offloaded(≤ 2%). Nevertheless, Figure 18a shows that offloading yields a 9% improvement in throughput with 2% of packet reordering, and in the worst case (5%), performance is still as good as software tls.

6.5 Scalability

Autonomous NIC offloads use per-flow state stored in NIC caches to perform well. But NIC caches are inherently limited. They can be exhausted when serving a few thousands of flows, triggering flow state eviction into main memory, which later incurs costly DMA operations over PCIe upon state reuse. For this reason, previous studies indicated that RDMA (which also uses per-flow state) does not scale well [19, 54]. The question is: do autonomous offloads

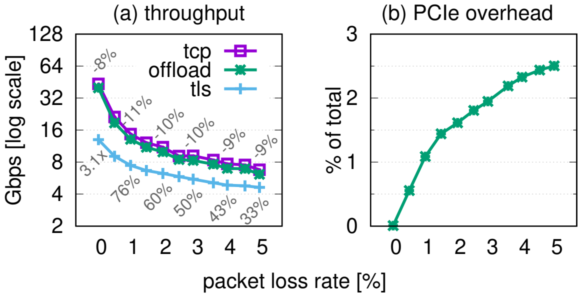


Figure 16: *Loss effect at sender (top and bottom labels show how offload relates to no encryption and software TLS, respectively). Throughput standard deviation is below 1.6%.*

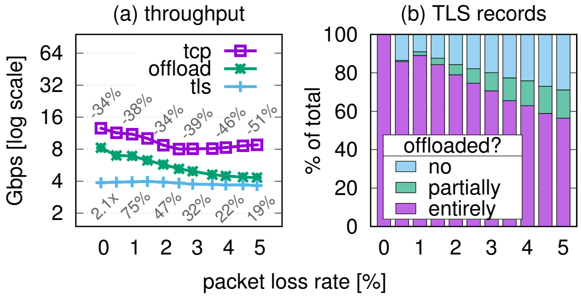


Figure 17: *Loss effect at receiver (top and bottom labels show how offload relates to no encryption and software TLS, respectively). Throughput standard deviation is below 2.61%.*

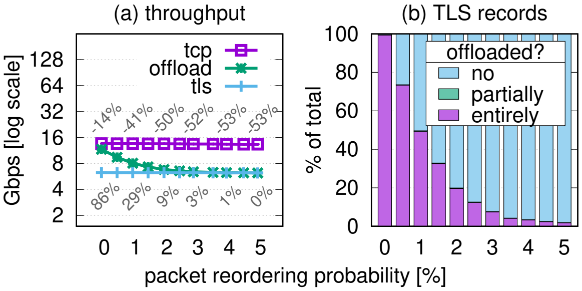


Figure 18: *Reordering effect at receiver (top and bottom labels show how offload relates to no encryption and software TLS, respectively).*

suffer from the same problem as the number of connections exceeds the capacity of the NIC caches?

To answer this question, we add another generator machine and connect it to the server using its ConnectX6-Dx second port. (As it happens, using two ports allows the throughput to exceed 100 Gbps somewhat.) We repeat the nginx experiment involving TLS offloading with eight cores and 256 KiB files in C_2 (data in page cache). But this time, we increase the number of connections, exponentially, from 64 to 128 K. With 4 MiB of on-NIC memory and 208 B per-flow state, the NIC can store at most 20 K flows, ignoring memory used for other resources such as packet queues.

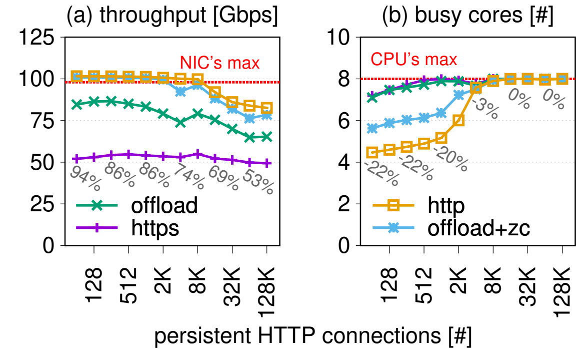


Figure 19: *Nginx scalability with two TLS offload variants compared to https (baseline encryption) and http (no encryption). All files reside in the page cache (C_2). The labels show offload+zc improvement over https.*

Figure 19 shows the results. As the number of connections increases, CPU utilization likewise increases until the CPU becomes the bottleneck. Contributing to the increased utilization is the fact that TCP packet batching becomes less effective with more connections: from 48 packets per batch with 128 connections, to only 8 packets per batch with 128 K connections.

Observe that for low connection counts, https and offload performance is not visibly bottlenecked on neither the CPU nor the NIC. This anomaly happens due to imbalanced request spreading that results in some underutilized cores.

In all measurements, offload+zc throughput is within 10% of http throughput, and it consumes at most 1.25x more CPU; offload and offload+zc deliver 32%–63% and 53%–94% higher throughput as compared to https, respectively.

Overall, our workload scales reasonably despite the inherent cache contention problem at the NIC caused by the growing aggregated size of per-flow state. These scalability results disagree with that of certain previous studies [19, 54]. We find that the reason for this disagreement is packet batching, which is dominant in our workloads (at least 8 packets per batch), as they involve bigger message sizes. In contract, the cited previous studies focus on smaller messages. More specifically, as the flow state size exceeds the NIC's cache capacity, each newly serviced packet might in principle trigger a cache miss and a costly memory access. But only the first packet in the batch incurs this cost, whereas subsequent packets do enjoy temporal locality. (We remark that batching might not be dominant if nginx is made to serve only small files. In this case, however, the workload ceases to be data-intensive and is thus outside the scope of our work.)

When comparing our work to the aforementioned previous studies [19, 54], it should also be noted that we use more recent NICs and thus benefit from their improved cache management and increased parallelism, which, similarly to batching, help hide cache miss latencies as demonstrated by a more recent NIC scalability study [90].

7 APPLICABILITY

Next, we further discuss the applicability of autonomous offloads to additional computations and protocols.

Decompression and deserialization As discussed in §3.1, non-size-preserving operations preclude offloading when sending, but not when receiving. A non-sized-preserving operation can be performed on receive by having the NIC write the offloaded operation's results to pre-allocated buffers (set up by the L5P) while also writing the original packet data as-is to the driver's receive ring. The driver will pass packets with offload results as metadata up to the network stack. Later, L5P software will skip performing the offloaded operation if all the packets in the message were offloaded; otherwise, it will fall back to software using the original packet data.

As mentioned above, we require pre-allocated buffers to offload non-size-preserving computation. To pre-allocate these buffers, we need to have either (1) predetermined response sizes, as in the NVMe-TCP protocol, or (2) maximum message size limits enforced by the implementations, such as in HTTP servers that limit request headers to 16 KB.

We note that in contrast to copy, encryption, and digest offloads, which pass packet data through PCIe and memory only once, non-size-preserving offloads will pass packet data through PCIe and memory twice: (1) offload results and (2) original packet data that is needed only for software fallback processing. Nevertheless, this is still better than off-CPU accelerators that pass data three times: (1) from the network; (2) to the off-CPU accelerator; and (3) from the off-CPU accelerator.

Pattern matching Deep packet inspection (DPI) software looks for known patterns in packet payloads using either fixed-length string pattern matching or regular expression matching. Patterns are matched only within L5P messages and never across messages. Thus, these computations fit our offload properties and we can autonomously offloaded them as follows: for each packet, check if some pattern match completes within it using the per-flow context to track pattern matches across packets. If yes, report the match with metadata to indicate the pattern; otherwise, report that the packet contains no match. Later, DPI software inspects packets in-order and if all packets of an L5P message are marked by the NIC, then report results according to offload metadata. Otherwise, if some packet bypassed NIC offload, perform DPI in software.

Not restricted to TCP This paper focuses on L5Ps built on top of TCP. But autonomous offloading is, in fact, orthogonal to the specific layer-4 protocol that is being used. Namely, an L5P is autonomously offloadable if it has the properties defined in §4, regardless of the specific underlying layer-4 that it is built upon. The reason we have chosen to focus on TCP (in addition to its popularity) is because its properties make it the most challenging to autonomously offload. All the other layer-4 protocols that we are aware of can be either similarly offloaded or are easier to offload.

Consider, e.g., a simple L5P that is built on top of UDP and directly mirrors its properties. A message of this L5P is therefore a datagram that is entirely contained in a UDP packet; the message might get lost or be handed to the receiving end out-of-order. For example, DTLS (Datagram Transport Layer Security [108]) is such an L5P, as it only encrypts and decrypts UDP packets. Autonomously offloading this type of L5Ps is trivial and does not merit an academic publication (we indeed do not consider it part of our contribution). Because the NIC operates on individual, self-contained datagrams, it never has to worry about such issues as losing and having to reconstruct its position in the sequence due to packet reordering and loss. Falling back on L5P software processing is likewise never needed: the NIC always knows what to do next, since all the information required for acceleration is encapsulated inside the currently-processed incoming or outgoing datagram.

The main contribution of this work is coming up with a way to autonomously offload a more sophisticated type of protocols—those that provide some *stream abstraction* for their users. The challenging aspect in autonomously offloading such protocols is that an L5P message can be spread across multiple packets in the stream with no alignment between L5P messages and packets, making it challenging for the NIC to identify L5P message boundaries in the face of packet reordering and loss.

SCTP (Stream Control Transmission Protocol [118]) can be viewed as an L5P that uses UDP to provide reliable, in-sequence delivery of a stream of messages with congestion control. SCTP divides messages into "chunks," such that each chunk is entirely contained in a UDP packet along with its own header. A chunk header indicates, in particular, whether the associated data starts a new SCTP message. Therefore, autonomously offloading SCTP is similar to, but easier than TCP-based offloads, because the NIC can identify message beginnings within packets in a deterministic manner, ridding it from the need to speculate using magic patterns.

QUIC [49] is an emerging protocol that provides a stream abstraction. It is capable of multiplexing multiple byte streams on top of encrypted UDP packets. Each packet contains one or more "frames" that corresponds to some byte stream. A QUIC autonomous offload must be able to encrypt and decrypt the packets. (Simpler than TLS offloading, as it is done per UDP packet.) Then, given access to the frames' content, all autonomously offloadable operations become relevant: copy to avoid L5P message reassembly, decompression (e.g., QPACK [61]), pattern matching, etc.

8 CONCLUSIONS

Autonomous NIC offloading is a new way to accelerate layer-5 protocols. The approach is appealing because it is nonintrusive, allowing system designers to keep the existing TCP/IP stack intact. We speculate that the non-intrusiveness of the design would give rise to additional layer-5 offloads in the future and perhaps even influence protocol design.

ACKNOWLEDGMENTS

This research was funded in part by the Israel Science Foundation (grant 2005/17) and the Blavatnik Family Foundation. We thank Mellanox architecture, chip-design, and software development teams for their contributions, notably Ilya Lesokhin, Adi Menachem, Miriam Menes, Uria Basher, Ariel Shachar, Tariq Tokun, Ben Ben-Ishay, Gal Shalom, Hans Peter Selasky, Or Gerlitz, and Noam Cohen. We also thank Mark Silberstein, Gail Weiss, the anonymous reviewers, and our shepherd Mike Marty for their valuable feedback. Finally, we thank Oran Agra, Guy Korland, and Yiftach Shoolman from Redis Labs for providing Redis on Flash and helping us adopt it for our purposes.

REFERENCES

[1] D. Eastlake 3rd and P. Jones. 2001. *US Secure Hash Algorithm 1 (SHA1).* RFC 3174. Internet Engineering Task Force. 22 pages.

http://www.rfc-editor.org/rfc/rfc3174.txt.

[2] Aizman Alex and Yusupov Dmitry. 2005. Open-iSCSI High-Performance Initiator for Linux. https://lwn.net/Articles/126530/. Accessed: 2020-03-24.

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 63–74. https://doi.org/10.1145/1851275.1851192.

[4] Muhammad Shoaib Bin Altaf and David A. Wood. 2017. LogCA: A High-Level Performance Model for Hardware Accelerators. In *ACM International Symposium on Computer Architecture (ISCA)*. 375–388. https://doi.org/10.1145/3079856.3080216.

[5] Jens Axboe. 2014. Fio - Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html.

[6] John Baldwin. 2019. TLS in the kernel. https://reviews.freebsd.org/D21277. FreeBSD Kernel patches. Accessed: 2020-03-24.

[7] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*. 49–65. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-belay.pdf.

[8] Pismenny Boris and Lesokhin Ilya. 2018. TLS offload rx, netdev & mlx5. https://lwn.net/Articles/759052/. Accessed: 2019-08-27.

[9] Pismenny Boris and Kuperman Yossi. 2018. UDP GSO Offload. https://netdevconf.info/0x12/session.html?udp-segmentation-offload. Accessed: 2020-05-23.

[10] Peter Breuer, Andrés Marín-López, and Arturo Ares. 2000. The network block device. *Linux Journal* 73 (05 2000). https://www.linuxjournal.com/article/3778.

[11] Chelsio Communications. 2018. Chelsio Cryptographic Offload and Acceleration Solution Overview. https://www.chelsio.com/crypto-solution/. Accessed: 2018-12-13.

[12] Mosharaf Chowdhury and Ion Stoica. 2012. Coflow: A Networking Abstraction for Cluster Applications. In *ACM Workshop on Hot Topics in Networks (HotNets)*. 31—-36. https://doi.org/10.1145/2390231.2390237.

[13] D. D. Clark and D. L. Tennenhouse. 1990. Architectural Considerations for a New Generation of Protocols. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 200–208. https://doi.org/10.1145/99508.99553.

[14] Edward Cree. 2016. Checksum Offloads. https://www.kernel.org/doc/html/latest/networking/checksum-offloads.html. Accessed: 2020-03-24.

[15] Joan Daemen and Vincent Rijmen. 2013. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media.

[16] Watson Dave, Pismenny Boris, Lesokhin Ilya, and Yehezkel Aviad. 2017. Kernel TLS. https://lwn.net/Articles/725721/. Accessed: 2020-03-24.

[17] Tim Dierks and Eric Rescorla. 2008. *The transport layer security (TLS) protocol version 1.2*. RFC. Internet Engineering Task Force. 104 pages. https://rfc-editor.org/rfc/rfc5246.txt.

[18] DPDK VLAN 2014. VLAN Offload Tests. https://dpdk-test-plans.readthedocs.io/en/latest/vlan_test_plan.html. Accessed: 2019-08-30.

[19] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}.

[20] Nandita Dukkipati, Tiziana Refice, Yuchung Cheng, Jerry Chu, Tom Herbert, Amit Agarwal, Arvind Jain, and Natalia Sutin. 2010. An Argument for Increasing TCP's Initial Congestion Window. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 26–33. https://doi.org/10.1145/1823844.1823848.

[21] Alexander Duyck. 2016. Segmentation Offloads. https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html. Accessed: 2020-03-24.

[22] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *USENIX Annual Technical Conference (ATC)*. 345–362. https://www.usenix.org/conference/atc19/presentation/eran.

[23] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone.

[24] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux Journal* 2004, 124 (Aug 2004), 5. http://dl.acm.org/citation.cfm?id=1012889.1012894.

[25] Linux Foundation. 2016. Why Linux engineers currently feel that TOE has little merit. https://wiki.linuxfoundation.org/networking/toe. Accessed: 2018-11-06.

[26] Eitan Frachtenberg. 2012. Holistic datacenter design in the open compute project. *Computer* 45, 7 (2012), 83–85. https://doi.ieeecomputersociety.org/10.1109/MC.2012.235.

[27] H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson. 2010. Introduction to the wire-speed processor and architecture. *IBM Journal of Research and Development* 54, 1 (2010), 3:1–3:11. https://doi.org/10.1147/JRD.2009.2036980.

[28] Steve French. 2007. CIFS VFS - Advanced Common Internet File System for Linux. https://linux-cifs.samba.org/. Accessed: 2020-03-24.

[29] Drew Gallatin. 2020. Netflix view on TOE. Private email communication with a Netflix engineer; quote approved by Netflix and used with permission.

[30] Will Glozer. 2012. Wrk - a HTTP benchmarking tool. https://github.com/wg/wrk.git. Accessed: 2019-08-06.

[31] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. 2017. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 83–96. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/go.

[32] Google. 2015. gRPC: a high-performance, open source universal RPC framework. https://grpc.io/. Accessed: 2020-03-05.

[33] Vinodh Gopal, J Guilford, E Ozturk, G Wolrich, W Feghali, J Dixon, and D Karakoyunlu. 2011. Fast CRC computation for iSCSI Polynomial using CRC32 instruction. *Intel Corporation* (2011). https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/crc-iscsi-polynomial-crc32-instruction-paper.pdf.

[34] Vinodh Gopal, Sean Gulley, Wajdi Feghali, Dan Zimmerman, and Ilya Albrekht. 2015. *Improving openssl performance*. Technical Report. Intel Corporation. https://software.intel.com/en-us/articles/improving-openssl-performance.

[35] Sagi Grimberg. 2018. TCP transport binding for NVMe over Fabrics. https://lwn.net/Articles/772556/. Accessed: 2020-03-24.

[36] Shay Gueron. 2010. Intel advanced encryption standard instructions (AES-NI). *Intel White Paper* (2010). https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf.

[37] Sean Gulley, Vinodh Gopal, Kirk Yap, Wajdi Feghali, J Guilford, and Gil Wolrich. 2013. Intel sha extensions. *Intel White Paper* (2013). https://software.intel.com/content/dam/develop/external/us/en/documents/intel-sha-extensions-white-paper-402097.pdf.

[38] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: A GPU-Accelerated Software Router. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 195–206. https://doi.org/10.1145/1851182.1851207.

[39] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*. USENIX, Hollywood, CA, 135–148. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han.

[40] Xiaokang Hu, Changzheng Wei, Jian Li, Brian Will, Ping Yu, Lu Gong, and Haibing Guan. 2019. QTLS: High-performance TLS Asynchronous Offload Framework with Intel QuickAssist Technology. In *ACM Symposium on Principals and Practice of Parallel Programming (PPoPP)*. 158–172. http://doi.acm.org/10.1145/3293883.3295705.

[41] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 127–140. https://www.usenix.org/conference/nsdi20/presentation/hwang.

[42] Burstein Idan. 2019. Enabling Remote Persistent Memory. https://www.snia.org/educational-library/enabling-remote-persistent-memory-2019. Persistent memory summit. Accessed: 2020-05-23.

[43] Lesokhin Ilya, Pismenny Boris, and Yehezkel Aviad. 2017. tls: Add generic NIC offload infrastructure. https://lwn.net/Articles/738847/. Accessed: 2019-08-27.

[44] Intel. 2015. Intel QuickAssist Adapter 8950 Product Brief. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/quickassist-adapter-8950-brief.pdf. Accessed: 2018-12-13.

[45] Intel. 2019. Accelerating Redis with Intel DC persistent memory. https://ci.spdk.io/download/2019-summit-prc/02_Presentation_13_Accelerating_Redis_with_Intel_Optane_DC_Persistent_Memory_Dennis.pdf. Accessed: 2019-08-06.

[46] Intel Corporation. 2010. DPDK: Data Plane Development Kit. http://dpdk.org. (Accessed: May 2016).

[47] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent Distributed Storage. *Proceedings of the VLDB Endowment* (2017), 1202–1213. https://doi.org/10.14778/3137628.3137632.

[48] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. 2016. Consensus in a Box: Inexpensive Coordination in Hardware. In *USENIX*

Symposium on Networked Systems Design and Implementation (NSDI). 425–438. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/istvan.

[49] J. Iyengar and M. Thomson. 2020. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC Draft. Internet Engineering Task Force. 206 pages. https://tools.ietf.org/html/draft-ietf-quic-transport-34.

[50] A. Choudhury J. Salowey and D. McGrew. 2008. *AES Galois Counter Mode (GCM) Cipher Suites for TLS*. RFC. Internet Engineering Task Force. 8 pages. https://tools.ietf.org/html/rfc5288.

[51] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 489–502. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

[52] Rick A. Jones. 2009. MongoDB: The database for modern applications. https://www.mongodb.com/. Accessed: August, 2020.

[53] JSOF research lab. 2019. Ripple20: 19 Zero-Day Vulnerabilities Amplified by the Supply Chain. https://www.jsof-tech.com/ripple20/. Accessed: 2020-08-07.

[54] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*. 185–201. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia.

[55] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. In *ACM International Symposium on Computer Architecture (ISCA)*. 158—-169. https://doi.org/10.1145/2872887.2750392.

[56] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 67–81. http://dx.doi.org/10.1145/2872362.2872367.

[57] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *ACM Eurosys* (Dresden, Germany). 1–16. https://doi.org/10.1145/3302424.3303985.

[58] Franziskus Kiefer. 2017. Improving AES-GCM Performance. https://blog.mozilla.org/security/2017/09/29/improving-aes-gcm-performance/. Mozilla Security Blog. Accessed: 2020-03-05.

[59] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 297–312. https://doi.org/10.1145/3230543.3230572.

[60] E Kohler, M Handley, and S Floyd. 2006. *Datagram Congestion Control Protocol (DCCP)*. RFC. Internet Engineering Task Force. 129 pages. https://rfc-editor.org/rfc/rfc4340.txt.

[61] C Krasic, M. Bishop, and Ed. A. Frindell. 2020. *QPACK: Header Compression for HTTP/3*. RFC Draft. Internet Engineering Task Force. 49 pages. https://tools.ietf.org/html/draft-ietf-quic-qpack-20.

[62] Vlad Krasnov. 2016. It takes two to ChaCha (Poly). https://blog.cloudflare.com/it-takes-two-to-chacha-poly/. The Cloudflare Blog. Accessed: 2020-03-05.

[63] Vlad Krasnov. 2017. How "expensive" is crypto anyway? https://blog.cloudflare.com/how-expensive-is-crypto-anyway/. The Cloudflare Blog. Accessed: 2020-03-05.

[64] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. 2016. Paravirtual remote i/o. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 49–65. http://dx.doi.org/10.1145/2872362.2872378.

[65] Redis Labs. 2013. Memtier benchmark. https://github.com/RedisLabs/memtier_benchmark. Accessed: 2029-03-05.

[66] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. 2017. Page fault support for network controllers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 449–466. https://doi.org/10.1145/3037697.3037710.

[67] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK Ports, Irene Zhang, Ricardo Bianchini, Haryadi S Gunawi, and Anirudh Badam. 2020. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 591–605. https://doi.org/10.1145/3373376.3378531.

[68] ARM Limited. 2011. *ARMv8 Instruction Set Overview*. Technical Report. ARM. https://www.element14.com/community/servlet/JiveServlet/previewBody/41836-102-1-229511/ARM.Reference_Manual.pdf.

[69] Linux and TOE 2005. Linux and TCP offload engines. https://lwn.net/Articles/148697/. Accessed: 2018-11-06.

[70] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 175–186. http://doi.acm.org/10.1145/2619239.2626311.

[71] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. 2017. Disk|Crypt|Net: Rethinking the Stack for High-performance Video Streaming. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 211–224. https://doi.org/10.1145/3098822.3098844.

[72] Iskra Mark, Dibbiny Majd, and Tragler Anita. 2019. Deployable OVS hardware offloading for 5G telco clouds. https://www.openvswitch.org/support/ovscon2019/day2/1125-dibbiny-tragler-iskra-shern-efraim.pdf. Accessed: 2020-05-23.

[73] W. Pepper Marts, Matthew G. F. Dosanjh, Whit Schonbein, Ryan E. Grant, and Patrick G. Bridges. 2019. MPI Tag Matching Performance on ConnectX and ARM. In *Proceedings of the 26th European MPI Users' Group Meeting (EuroMPI '19)*. Article 13, 10 pages. https://doi.org/10.1145/3343211.3343224.

[74] Mellanox. 2011. ConnectX®-3 Pro Product Brief. https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_ConnectX-3_Pro_Card_EN.pdf. Accessed: 2019-08-06.

[75] Mellanox. 2013. Messaging Accelerator (VMA). https://www.mellanox.com/products/software/accelerator-software/vma. Accessed: 2020-02-05.

[76] Mellanox. 2014. ConnectX®-4 En Card Product Brief. https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_ConnectX-4_EN_Card.pdf. Accessed: 2019-08-06.

[77] Mellanox. 2017. ConnectX®-5 En Card Product Brief. https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_ConnectX-5_EN_Card.pdf. Accessed: 2019-08-06.

[78] Mellanox. 2018. ConnectX®-6 En Card Product Brief. https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_ConnectX-6_EN_Card.pdf. Accessed: 2019-08-06.

[79] Mellanox. 2020. ConnectX®-6 Dx En Card Product Brief. https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_ConnectX-6_Dx_EN_Card.pdf. Accessed: 2020-07-06.

[80] Mellanox. 2020. Mellanox company timeline. https://www.mellanox.com/company/timeline. Accessed: 2019-08-06.

[81] Mellanox. 2020. Mellanox NIC pricing list effective March 2020. https://store.mellanox.com/. Accessed: 2020-03-24.

[82] Microsoft 2017. Introduction to Receive Side Scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling. Accessed: January 2020.

[83] Microsoft 2017. Overview of Receive Segment Coalescing. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/overview-of-receive-segment-coalescing. Accessed: January 2020.

[84] Microsoft. 2017. Why Are We Deprecating Network Performance Features (KB4014193)? https://techcommunity.microsoft.com/t5/Core-Infrastructure-and-Security/Why-Are-We-Deprecating-Network-Performance-Features-KB4014193/ba-p/259053. Accessed: 2019-08-30.

[85] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting network support for RDMA. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 313–326. https://doi.org/10.1145/3230543.3230557.

[86] Jeffrey C Mogul. 2003. TCP Offload Is a Dumb Idea Whose Time Has Come. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. 25–30. https://www.usenix.org/conference/hotos-ix/tcp-offload-dumb-idea-whose-time-has-come.

[87] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 77–92. https://www.usenix.org/conference/nsdi20/presentation/moon.

[88] Energy Sciences Network. 2012. NIC Tuning - Chelsio 10Gig NIC, Linux and FreeBSD. https://fasterdata.es.net/host-tuning/nic-tuning/. Accessed: 2020-03-24.

[89] TechTarget Network. 2013. TCP offload's promises and limitations for enterprise networks. https://searchdatacenter.techtarget.com/tip/TCP-offloads-promises-and-limitations-for-enterprise-networks. Accessed: 2020-03-24.

[90] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. 2019. Storm: A Fast Transactional Dataplane for Remote Data

Structures. In *ACM International Systems and Storage Conference (SYSTOR)*. Association for Computing Machinery, New York, NY, USA, 97—-108. https://doi.org/10.1145/3319647.3325827.

[91] NVM Express Workgroup. 2014. NVM Express (NVMe) Specification – Revision 1.2. http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_2-Gold-20141209.pdf. Accessed: Jan 2015.

[92] NVM Express Workgroup. 2018. NVMe/TCP Transport Binding specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-over-Fabrics-1.0-Ratified-TPs.zip. Accessed: Jan 2020.

[93] Tzahi Oved. 2018. T10-DIF offload. https://www.openfabrics.org/images/2018workshop/presentations/307_TOved_T10-DIFOffload.pdf. OpenFabrics alliance workshop. Accessed: 2020-05-23.

[94] Vern Paxson. 1997. End-to-end Internet packet dynamics. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 139–152. https://doi.org/10.1145/263109.263155.

[95] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*. 1–16. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-peter_simon.pdf.

[96] Boris Pismenny. 2018. Kernel TLS Receive Side. https://github.com/openssl/openssl/pull/7848. Accessed: 2019-08-27.

[97] Boris Pismenny. 2018. Kernel TLS socket API. https://github.com/openssl/openssl/pull/5253. Accessed: 2019-08-27.

[98] Boris Pismenny. 2019. KTLS Sendfile. https://github.com/openssl/openssl/pull/8727. Accessed: 2019-08-27.

[99] Steven Pope and David Riddoch. 2007. 10Gb/s Ethernet Performance and Retrospective. *SIGCOMM Comput. Commun. Rev.* 37, 2 (March 2007), 89–92. https://doi.org/10.1145/1232919.1232930 https://doi.org/10.1145/1232919.1232930.

[100] Steve Pope and David Riddoch. 2011. *Introduction to OpenOnload*. Technical Report. Solarflare Communication. https://www.openonload.org/.

[101] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. 2019. Unikernels: The Next Stage of Linux's Dominance. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. 7—-13. https://doi.org/10.1145/3317550.3321445.

[102] RedHat. 2019. SegmentSmack and FragmentSmack: IP fragments and TCP segments with random offsets may cause a remote denial of service. https://access.redhat.com/articles/3553061. Accessed: 2020-08-07.

[103] Redis 2011. Redis Labs. https://redislabs.com. (Accessed: May 2020.).

[104] Will Reese. 2008. Nginx: The High-performance Web Server and Reverse Proxy. http://dl.acm.org/citation.cfm?id=1412202.1412204. *Linux J.* 2008, 173, Article 2 (Sept. 2008).

[105] Philipp Reisner. 2009. The Distributed Replicated Block Device (DRBD) driver. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b411b3637fa71fce9cf2acf0639009500f5892fe. Accessed: 2020-03-24.

[106] E. Rescorla. 2000. *HTTP Over TLS*. RFC 2818. Internet Engineering Task Force. 7 pages. http://www.rfc-editor.org/rfc/rfc2818.txt.

[107] Eric Rescorla. 2018. *The transport layer security (TLS) protocol version 1.3*. RFC. Internet Engineering Task Force. https://rfc-editor.org/rfc/rfc8446.txt.

[108] Eric Rescorla and Nagendra Modadugu. 2012. *Datagram Transport Layer Security Version 1.2*. RFC. Internet Engineering Task Force. https://rfc-editor.org/rfc/rfc6347.txt.

[109] Luigi Rizzo. 2012. Netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo.

[110] RocksDB 2012. RocksDB: A persistent key-value store. https://rocksdb.org. (Accessed: May 2020.).

[111] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. 2004. *Internet Small Computer Systems Interface (iSCSI)*. RFC 3720. Internet Engineering Task Force. http://www.rfc-editor.org/rfc/rfc3720.txt.

[112] Manish Shah, Robert Golla, Gregory Grohoski, Paul Jordan, Jama Barreh, Jeffrey Brooks, Mark Greenberg, Gideon Levinsky, Mark Luttrell, Christopher Olson, et al. 2012. Sparc T4: A dynamically threaded server-on-a-chip. *IEEE/ACM International Symposium on Microarchitecture (MICRO)* 32, 2 (2012), 8–19. https://doi.org/10.1109/MM.2012.1.

[113] D. Sheinwald, J. Satran, P. Thaler, and V. Cavanna. 2002. *Internet Protocol Small Computer System Interface (iSCSI) Cyclic Redundancy Check (CRC)/Checksum Considerations*. RFC 3385. Internet Engineering Task Force. 23 pages. http://www.rfc-editor.org/rfc/rfc3385.txt.

[114] Kevin Shu. 2018. Optimize Redis with NextGen NVM. https://www.snia.org/sites/default/files/SDC/2018/presentations/PM/Shu_Kevin_Optimize_Redis_with_NextGen_NVM.pdf. Intel. Accessed: 2019-08-06.

[115] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 2020. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 708—-721. https://doi.org/10.1145/3387514.3405897.

[116] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable cross-language services implementation. *Facebook White Paper* 5, 8 (2007). https://thrift.apache.org/static/files/thrift-20070401.pdf.

[117] Igor Smolyar, Alex Markuze, Boris Pismenny, Haggai Eran, Gerd Zellweger, Austin Bolen, Liran Liss, Adam Morrison, and Dan Tsafrir. 2020. IOctopus: Outsmarting Nonuniform DMA. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 101–115. https://doi.org/10.1145/3373376.3378509.

[118] R. J. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, and M. Kalla. 2000. *Stream Control Transmission Protocol*. RFC 2960. Internet Engineering Task Force. 134 pages. http://www.rfc-editor.org/rfc/rfc2960.txt.

[119] Jeff Stuecheli. 2013. POWER8. In *Hot Chips*. 1–20. https://doi.org/10.1109/HOTCHIPS.2013.7478303.

[120] Nick Sullivan. 2015. Do the ChaCha: better mobile performance with cryptography. https://blog.cloudflare.com/do-the-chacha-better-mobile-performance-with-cryptography/. The Cloudflare Blog. Accessed: 2020-03-05.

[121] Nick Sullivan. 2016. Padding oracles and the decline of CBC-mode cipher suites. https://blog.cloudflare.com/padding-oracles-and-the-decline-of-cbc-mode-ciphersuites/. The Cloudflare Blog. Accessed: 2020-03-05.

[122] Veritas Support. 2015. How to Disable TCP Chimney, TCP/IP Offload Engine and/or TCP Segmentation Offload. https://www.veritas.com/content/support/en_US/article.100031033. Accessed: 2020-03-24.

[123] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. 2005. Iperf: The tcp/udp bandwidth measurement tool. *dast. nlanr. net/Projects* (2005), 38. https://iperf.fr/.

[124] Herbert Tom and de Bruijn Willem. 2011. Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt. Accessed: 2020-03-05.

[125] WolfSSL. 2016. WolfSSL/Wolfcrypt async with Intel QuickAssist. https://www.wolfssl.com/docs/intel-quickassist/. Accessed: 2020-02-05.

[126] WolfSSL. 2018. WolfSSL ARMv8 support. https://www.wolfssl.com/wolfssl-on-armv8-lemaker-2/. Accessed: 2020-04-05.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact provides modified Linux kernel sources and configurations along with modified nginx, fio, and iperf benchmarks, and with further scripts to use these to reproduce the experiments and plot the graphs in the paper. This will allow evaluation of the cycle breakdown results and NVMe-TCP result on any x86 system. Results involving TLS offload require Mellanox ConnectX6-Dx crypto enabled NICs. We also provide the raw data we collected to plot the figures in the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Autonomous inline offload for TLS and NVMe-TCP.
- **Program:** Linux kernel, nginx, fio, iperf. All sources included.
- **Compilation:** gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1 16.04)
- **Transformations:** Linux kernel: (1) support for kernel TLS offload in mlx5_core.ko; (2) support for NVMe-TCP offload emulation in tcp.ko; (3) support for measuring TLS AES-GCM crypto cycles in tls.ko; and (4) support for measuring NVMe-TCP copy and CRC cycles in tcp.ko.
  Nginx: Support for using sendfile with kernel TLS.
  Iperf: Support for using kernel TLS.
  We note that some of the modifications are already upstream and open-source, for instance, we have modified OpenSSL and our patches are available in the official OpenSSL Github page.
- **Run-time environment:** Tested on Ubuntu 16.04.
- **Hardware:** ConnectX6-Dx crypto enabled NICs.
- **Experiments:** As described in the evaluation section.

- **How much time is needed to prepare workflow (approximately)?:** About an hour or two; installing the Linux kernel is probably most time consuming.
- **How much time is needed to complete experiments (approximately)?:** About 15 hours for 5 repetitions of all tests = 1h for TLS iperf, 10 hours for fio, and 4 hours for nginx TLS. You can reduce this significantly by running less repetitions (replace REPEAT=5 with the desired number).
- **Publicly available?:** https://github.com/BorisPis/autonomous-asplos21-artifact.git
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.4319415

A.3 Description

A.3.1 Hardware dependencies

All experiments depend on a pair of machines connected back-to-back: device under test (DUT) and load generator machines. Only the experiment in Figure 13 requires access to Mellanox ConnectX6-Dx crypto enabled NICs. NVMe-TCP experiments require the load generator to expose a disk over NVMe-TCP to the DUT. If no disk is available, then scripts are provided to setup a null device or a ramdisk instead.

A.3.2 Software dependencies

See Github link above.

A.4 Installation

Obtain the code and sub-modules from github and run `make` to compile all. Follow instructions in the "Setup configuration" section in the Github repository to configure your machine parameters (IPs/MACs/CPU cores and their affinity/etc.) for example configurations, see `TestSuite/Conf/config_*.sh`.

A.5 Experiment workflow and expected result

Use shell scripts (`scripts/run_*.sh`) to reproduce results and plot them using gnuplot scripts (`scripts/plot_*.sh`) that generate Figure 10, Figure 11, and Figure 13 as in the paper.