kTLS Offload Performance Enhancements for Real-life Applications

Bar Tuaf, Tal Gilboa, Tariq Toukan

Nvidia
Tel-Hai/Yokneam, Israel
{bartu, talgi, tariqt}@nvidia.com

Abstract

Information security is a continuously growing concern for all internet services. As of today, more than 70% of Internet traffic is encrypted using Transport Layer Security (TLS). kTLS (kernel TLS) serves as a kernel layer unit that offers TLS operations support to secure TCP connections. First introduced in kernel v4.13 as a SW offload for user space libraries, NVIDIA Mellanox ConnectX-6Dx kTLS TX offload support was introduced by mlx5e driver in kernel v5.6 and the RX ability was added in kernel v5.9.

In this paper, we will review the life cycle of a HW offloaded kTLS connection and the driver-HW interaction required to support it. We will demonstrate and analyze the significant performance speed-up gained by offloading kTLS operations to the network device. It will be showcased with the well-known Nginx web server, using mlx5e driver on top of an NVIDIA Mellanox ConnectX-6Dx NIC.

Keywords

TLS Offload, TX Offload, RX offload, NIC, Network Devices, TLS, SW kTLS, kTLS device, Crypto, TCP, Nginx, Wrk.

Introduction

With today's ever-increasing link speeds, we see an increase in CPU usage, for general packet processing. Even when using known offloads like TSO, there might be CPU limitations for running 100GbE at a high packet rate. This problem tenfold when introducing the TLS protocol processing overhead.

Offloading crypto processing from the application to kernel and from kernel to NIC led the way for easy to maintain, high-performing implementation for secure internet traffic. It allows relying on the resiliency and robustness of the networking stack while eliminating the need for expensive crypto operations by offloading them to the NIC. Transmitted TLS offload packets traversing the stack unencrypted leave behind computational tasks for the device. In the same manner, received packets are decrypted by the device before being handed over to the stack. By reducing this CPU consuming operations on the software side, we can speed-up any application which uses a secure network. This way more CPU resources can be assigned for packet processing and 100GbE line-rate may be achieved.

Motivation

Running applications in a secure environment is necessary to ensure that private data is kept private. With the constant increase in networking line rates, offloading work from the CPU to the networking device is essential in order to keep up the pace free as much CPU as possible for network processing.

TLS TX Offload path

This section describes transmit-side TLS offload. Leaving encryption costly assignments to the NIC requires the coordination of software. Framing TLS packets as well as TLS headers and trailers are examples of software responsibilities. In addition, software should mark the socket as offloaded and place those packets into the dedicated socket. This combined with offloading the keys used by TLS sockets for encryption to the device compel only single send operation which helps to reduce PCIe latency and bandwidth.

TLS protocol encrypts each record independently while other well-known security protocols do so per packet. Each of these TLS records can be located on several TCP segments, while TCP segments can store multiple TLS records. Therefore, a driver must update the hardware in case of drops/retransmission.

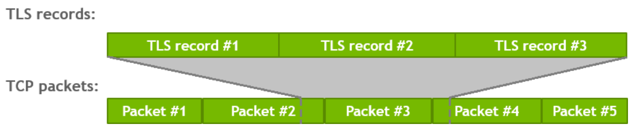

Figure 1- several TLS records spread on multiple TCP packets.

Data path

The TLS TX data path diverges right after calling sock_sendmsg() (and the subsequent call to inet_sendmsg()). kTLS SW implementation initiates encryption by calling tls_sw_sendmsg(), which allows parallel encryption/decryption (for systems with support for AES instructions set). But there's more to be done; the message to transmit is iterated, and on each iteration an encrypted message is allocated over the socket. The message is then processed for the wanted socket operation – in this case SK_PASS, which continues processing by calling tls_push_record(). while offloaded sockets go directly to tls_device_sendmsg() and tls_push_data(). Next, both flows travel toward the driver transmit functions. While the software works to encrypt the data, the TLS offload skips and sends plaintext to the marked socket.

Mlx5e kTLS TX flow examines each packet for the following conditions:

1. Does the packet belong to a TLS offloaded socket? if it does not, regular transmit packet processing will handle it.
2. Does the packet TCP sequence number equal the expected TCP sequence number that the driver/HW maintain? If it does not, trigger the resync flow (sync HW with the crypto context for the checked packet).

Subsequently software will send the packet for authentication and encryption on-the-fly by the NIC.

Resynchronization

From the previous sections we understand that in order to successfully process TCP packets on an offload socket, hardware must track the relevant connection crypto context. The driver is responsible for verifying that each SKB designated to TLS offload must have the expected TCP sequence number; otherwise it will be cast as a retransmission and transmit resync flow will be activated to restore the hardware state: Driver will post the packet, preceded with fast path communication to the device, syncing it with the upcoming TLS record state and TCP sequence number, doing the necessary fencing between the control communication and the packet data to guarantee proper ordering. No TLS stack procedure is involved in this process. This enforces the software to track TLS records and release them only after the record last SKB is acknowledged.

TLS RX Offload path

On the receive side, establishing a TLS offload connection requires the TLS kernel stack to provide the relevant TLS context to the device driver. This TLS context includes keys, IV and TCP-related context including 5-tuple and a sequence number, in addition to the TLS record sequence number. Once the NVIDIA Mellanox device is in offload state and as long as all TCP packets are obtained in-order, the NIC will decrypt the packets and mark them to notify the software to skip decryption. However, in a congested environment reordering/drops may occur; in this case our device will stop decrypting packets for this connection and will request the driver to resynchronize it in order to return to offload state.

Data path

The RX descriptor indicates whether the incoming packet was decrypted by the device. In this case the driver mark skb->decrypted and hands the packet to the stack. Similar to the transmit side path, TLS RX data paths are split in the expected sock_recvmsg() (also here after a call to inet_recvmsg()). While TLS offload continues to tls_device_recvmsg(), kTLS will call tls_sw_ recvmsg() to invoke decryption methods. A call chain generates the crypto function to decrypt packets by the novel OpenSSL libraries, while packets decrypted by hardware will skip those and go directly to the kTLS process and validation stages. Each SKB handled by the TLS RX offload path must hold metadata for driver usage. Packet that indicates an issue/error must not be aggregated by any layer besides kTLS. For each record received by the socket the TLS will perform the following checks:

1. Are all packets in the TLS record are decrypted?
   a. If true, copy the decrypted data to the user space.
2. Otherwise, check if part of the record is decrypted.
   a. If true, decrypt the remaining part using SW kTLS and copy plaintext data to user space.
3. Otherwise, fully encrypted record been received.
   a. Inform the driver to trigger a HW resynchronization flow and copy the decrypted data to the user space.

Partial decryption

Out of order can lead to handing software with TLS records that contains both cipher and plain text packets. In such cases kTLS validates the authentication and decrypts ciphertext packets within the mixed TLS record. The AES-GCM algorithm uses XOR operation on the date with the suitable keystream, which is generated using the TLS record IV (Initialization Vector). In order to get ciphertext or plaintext for a specific packet, XOR should be performed once with the keystream which leads to a single trip throw the TLS record.

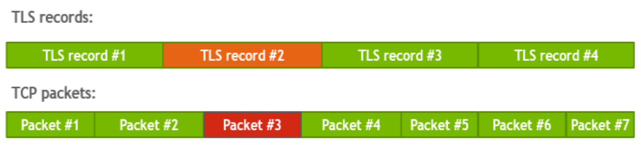

Figure 2- packet 3 received decrypted will all other is plaintext, SW will finish the job.

Resynchronization

The above flow shows that when the TLS offload engine encounters drops or an out-of-order packets, it might lose the TLS record framing within the TCP stream. In this case our device will indicate a resynchronization request in the RX descriptor and will stop decrypting packets for this connection until resynchronization procedure is completed successfully. Our driver uses the new async resync API in which it asks the TLS stack for a resync but provides the TCP sequence asynchronously at a later point after querying the device for it. Once the TLS stack finds a matching packet for that TCP sequence it informs the driver that configures the device back to offload state.

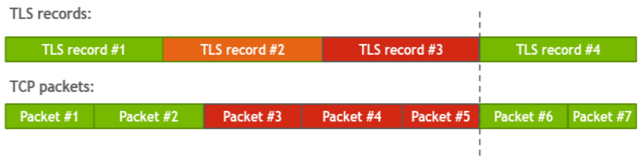

Figure 3- packets 3,4 and 5 contains cipher text, resync flow was completed successfully towards packet #6.

TX offload Real-life Application CPU saving

We demonstrate kTLS device TX offload benefits over Nginx server and Wrk benchmarking tool, using AMD EPYC 7742 connected through switch with NVIDIA Mellanox ConnectX-6Dx NICs. Wrk opens various amounts of connections over 64 threads, while repeatedly requesting 1MB files. Nginx and Wrk responds with either SW kTLS using OpenSSL TLSv1.2 and AES128-GCM-256 cipher suite, kTLS device offload and plain HTTP traffic. We used modified Nginx which uses sendfile() operations for HTTPS traffic to a better comparison as HTTP support it by default. This sendfile() implementation still copies the data to protect against changes in the page cache.

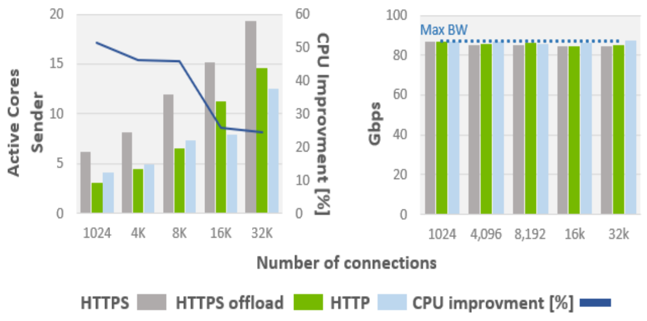

Figure 4- HTTPS kTLS device TX offload CPU utilization. Right table shows throughput reported on Wrk (Client). Left table show the number of active cores used in each case on server side (Nginx) and the CPU improvement between HTTPS using SW kTLS and kTLS device TX offload.

Hundreds of connections are enough in order to get full wire speed (100G) regardless the type of traffic, in the range between 1024 to 32k connections all cases show ~85Gbps application layer throughput (Goodput – excluding networking headers). CPU improvement was calculated by the following formula:

$$\frac{\textit{https active cores} - \textit{https kTLS device TX active cores}}{\textit{https active cores}} = \textit{CPU Improvment}$$

kTLS device TX offload saves 25% of the CPU used by SW kTLS with 32k connections and %51 of the core used to achieve line-rate over 1024 connections. In addition, we can notice a lower CPU utilization on server side with HTTPS using kTLS device TX offload in comparison to HTTP, this is a result of higher packet rate transmitted when running HTTP traffic.

Expected Improvement

In order to get a feel for the expected performance impact of full unidirectional kTLS device offload, we ran a patched version of Iperf, with OpenSSL support. This simulated a small-scale client-server application and would allow us to observe both an improvement in bandwidth and a reduction in CPU utilization.

Comparing the performance of SW kTLS implementation and full undir HW kTLS offload (TX offload on server side and RX offload on client side), we noticed a significant improvement in bandwidth, due to the reduced CPU utilization. With a single TCP stream, we achieved more than 2.5x speedup. For 8 streams, where SW kTLS struggled around 50Gbps, the HW offload already achieved 100GbE line-rate. In all cases, with 16 and above streams, where both implementations achieved a 100GbE line-rate, the HW offload case had much lower CPU utilization.

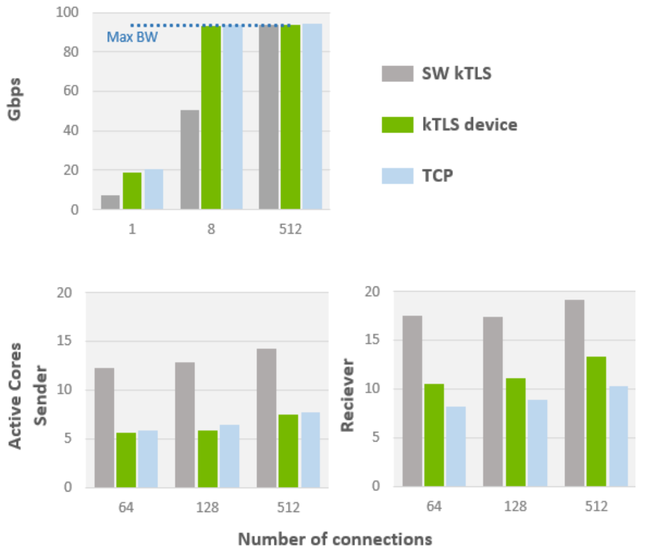

Figure 5- TLS offload throughput and CPU utilization. Top table shows better bandwidth as number of connections increase, bottom tables shows how CPU recover from TLS protocol overhead in both sender and receiver.

Once again sender active cores show better CPU utilization on transmit side for kTLS device in comparison to TCP traffic, reflecting higher packet rate for TCP. The results leave us with two main expectations for the full unidirectional kTLS device offload real-life cases:

1. Where SW kTLS can’t achieve line-rate – HW offload is expected to provide around 2.5x

bandwidth speed-up or get to line-rate while lowering CPU utilization.

2. Where SW kTLS achieves line-rate – HW offload should achieve line-rate as well, with lower CPU utilization (50-60% less).

References

1. "TLS Offload to Network Devices", Boris Pismenny, Ilya Lesokhin, Liran Liss and Haggai Eran (paper based on a talk presented at Netdev 1.2, Tokyo, Japan, September 1, 2016). *The Technical Conference on Linux Networking*. https://netdevconf.info/1.2/papers/netdevconf-TLS.pdf.

2. "TLS Offload to Network Devices - Rx offload", Boris Pismenny, Ilya Lesokhin and Liran Liss (paper based on a talk presented at Netdev 2.2, Seoul, Korea, November 1, 2017). *The Technical Conference on Linux Networking*. https://netdevconf.info/2.2/papers/pismenny-tlscrypto-talk.pdf

3. "Crypto kernel TLS socket", Dave Watson, accessed September 22, 2016, https://tools.ietf.org/html/rfc5246.

4. RFC 6347: Datagram Transport Layer Security Version 1.2, E. Rescola, accessed November 1, 2016. https://tools.ietf.org/html/rfc6347.

5. RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2, T. Dierks, University of Bern, Switzerland, accessed August 1, 2008. https://tools.ietf.org/html/rfc5246.

6. Nginx, Admin Guide. Nginx SSL Termination. https://docs.nginx.com/nginx/admin-guide/security-controls/terminating-ssl-http/

7. Nginx, Documentation. Module ngx_stream_core_module. http://nginx.org/en/docs/stream/ngx_stream_core_module.html

8. Nvidia Mellanox, Product Documentation. Kernel Transport Layer Security (kTLS) offloads. https://docs.mellanox.com/display/OFEDv502180/Kernel+Transport+Layer+Security+%28kTLS%29+Offloads