TLS Offload to Network Devices - Rx Offload

Boris Pismenny, Ilya Lesokhin, Liran Liss

Mellanox
Yokneam, Israel
{borisp, ilyal, liranl}@mellanox.com

Abstract

Encrypted Internet traffic is becoming the norm, spearheaded by the use Transport Layer Socket (TLS) to secure TCP connections. This trend introduces a great challenge to data center servers, as the symmetric encryption and authentication of TLS records adds significant CPU overhead. New CPU capabilities, such as the x86 AES-NI instruction set, alleviate the problem, yet encryption overhead remains high. Alternatively, cryptographic accelerators require dedicated hardware, consume significant memory bandwidth, and increase latency. We propose to offload TLS symmetric crypto processing to the network device. Our solution does not require a TCP Offload Engine (TOE). Rather, crypto processing is moved to a kernel TLS module (kTLS [5, 6]), which may leverage inline TLS acceleration offered by network devices. Transmitted packets of offloaded TLS connections pass through the stack unencrypted, and are processed on the fly by the device. Similarly, received packets are decrypted by the device before being handed off to the stack. We will describe the roles and requirements of the kTLS module, specify the device offload APIs, and detail the TLS processing flows. Finally, we will demonstrate the potential performance benefits of network device TLS offloads.

Keywords

TLS Offload, Rx Offload, Network Devices, TLS, Crypto, TCP.

Introduction

In today's networks, Transport Layer Security (TLS) is widely used to securely connect endpoints both inside data centers [1] and on the internet. TLS encrypts, decrypts, and authenticates its data, but these operations incur a significant overhead on the server.

Fixed function hardware accelerators are known to give improved performance and greater power-efficiency when compared to running a software implementation on a general purpose CPU. Cryptographic operation such as those used in TLS are very suitable for such hardware accelerators but they are not widely used in the context of networking. We believe that the reason is that the offload model is not good enough.

Existing solutions fall into four categories:

- **TLS Proxy** – A middlebox [2] is used to decrypt/encrypt all incoming/outgoing traffic. The middlebox is running a TCP connection against trusted machines and a TLS connection against untrusted machines, reducing the load on the trusted machine. However, if applied inside the data center, some traffic remains unprotected.

- **TOE** – TCP offload engines have been around for a while [3]. A TOE could run a full TLS offload as well reducing PCI traffic and freeing CPU cycles even further. However, the TCP stack of TOE devices is inflexible, hard to debug and fix when compared to a software TCP implementation. Moreover, with full TLS offload, security vulnerabilities could remain unfixed for a long time.

- **Crypto offload PCIe card** – A dedicated PCIe card to accelerate cryptographic operations, such as [4]. In the case of a PCIe card performing encryption/decryption operation, the data is sent towards the card over PCIe. It is then modified and sent back for further processing. This solution trades computational overhead for higher stress on the memory subsystem, leaving less memory bandwidth for other tasks, while also consuming more power.

- **TLS in the kernel** – Kernel TLS [5, 6] is kernel module for performing the bulk symmetric encryption of TLS records by the kernel instead of using a user space library. It facilities using sendfile for TLS connections. Moreover, where previously data was copied once during encryption and once again to be sent by TCP, using this approach encryption and data copy from user-space to the kernel become a single operation. This approach can leverage the x86 AES-NI instruction set for accelerating AES operations.

Motivation

In our previous paper we presented the transmit path offload. We use Iperf with OpenSSL support to show the speedup obtained by using this offload. Our setup consists of two Xeon E5-2620 v3 machines connected back-to-back with Innova-TLS NICs (ConnectX4-Lx + Xilinx FPGA). We show the speedup gained by the transmitting machine in terms of throughput per CPU cycle using TLS1.2 and the AES-GCM-256 ciphersuite. We compare the following:

- **openssl version 1.0.1e** with no offload
- **openssl version 1.0.1e with offload support** but using the OpenSSL read/write API
- **tls syscall** which uses OpenSSL for the TLS handshake and then calls the kernel's TLS read/write system calls directly
- **tcp** as an upper bound for potential speedup

We use the bandwidth/cycles measurement because for all models the bottleneck is the receive side packet processing. In this work, we offload the crypto receive side processing. Using this offload improves end-to-end bandwidth with TLS.

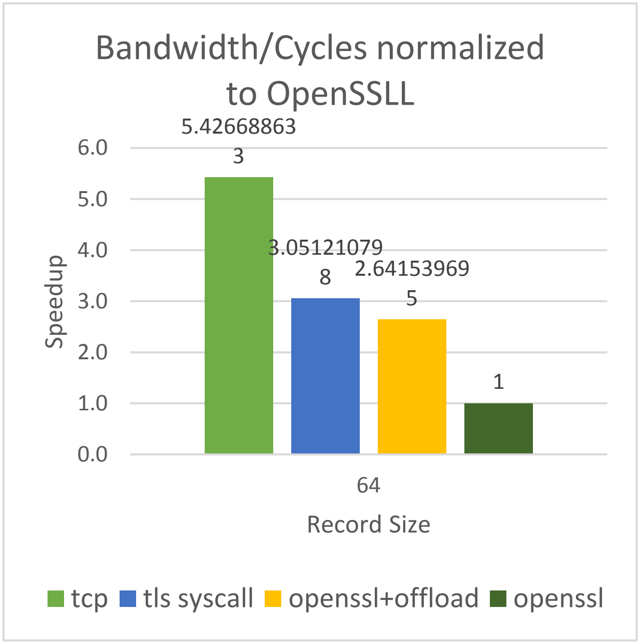

Figure 1. Speedup in terms of bandwidth per CPU cycle. Normalized to OpenSSL version 1.0.1e. Offload provides rovement for maximum sized records (16K) on the transmitting machine.

Model and Software Stack

In this paper, we propose a model and a software stack (see Figure 2) where the payload of network packets is transformed in-place by the network device. This model retains all the benefits of using a robust software network stack while offloading the crypto data crunching to the device. Since the data needs to reach the network device regardless of the offload. This model doesn't add any memory traffic or IO.

We focus on the AES-GCM ciphersuite, and the TLS1.2 protcol. It is possible to extend this model to other ciphersuites and TLS1.3 with some additional effort.

In the proposed model, the keys used by the TLS layer are offloaded to the NIC to which the connected socket is routed. The socket is marked as offloaded. From this moment onward packets of this TCP socket will be opportunistically decrypted by the device.

Upon receiving a packet, the device identifies it for offload according to the 5-tuple and TCP sequence number. The NIC will offload matching packets producing packets with the same headers, while replacing the payload with plaintext. Out of order packets are not processed by hardware and these are unmodified by hardware. Plaintext or ciphertext indication is maintained per SKB and the software stack must prevent coalescing of plaintext and ciphertext SKBs. TCP congestion control, memory management, retransmission, and other enhancements remain unchanged. Finally, the TLS layer does the required crypto operation to make sure the user gets authenticated plaintext. Typically the entire record is received as plaintext and already authenticated by the HW, so no cryptographic operation needs to be performed.

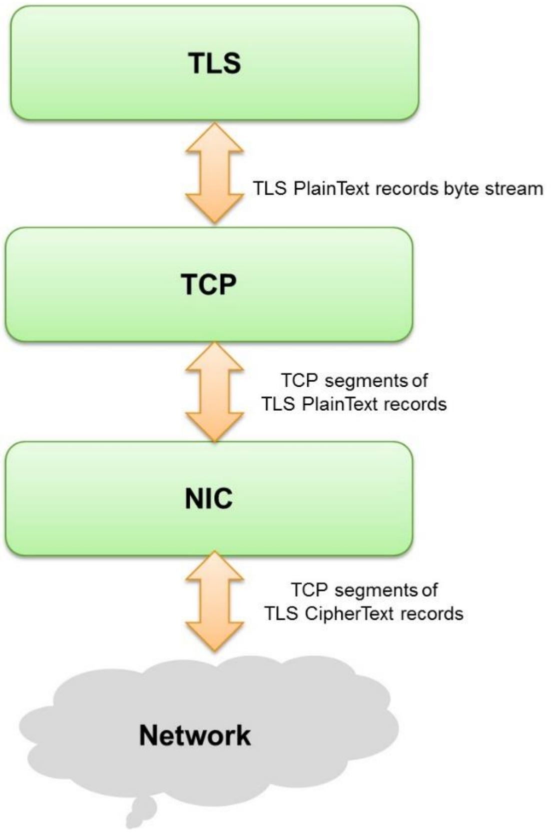
TLS
TLS PlainText records byte stream
TCP
TCP segments of TLS PlainText records
NIC
TCP segments of TLS CipherText records
Network

Figure 2. Kernel software stack for TLS offload for the fastpath flow. The NIC decrypts incoming TCP segments that carry TLS ciphertext. These packets go through the normal TCP/IP to the TLS layer where the TLS records are removed, but no crypto processing is required.

TLS Rx Offload Challenges

When the TLS offload is initiated, the TLS provides the relevent TLS context to the NIC. The TLS context including keys and IV, TCP context including the 5-tuple and an expected sequence number, and the TLS record sequence number are provided to the NIC. The good flow assumes that matching TCP packets are received in-order. The NIC will decrypt all these packets and provide an indication to software, which could skip decryption. The problem is with packet drops/reordering.

Upon packet drops/reordering, the NIC loses the state required to perform additional packet offload. For example, in TLS after reordering, the NIC might lose track of the TLS record format and TLS record sequence numbers, which are necessary and sufficient for inline TLS packet processing.

If software would attempt to assign a new TLS context, then it would need to provide the TCP sequence of the next expected TLS record. However, software processes packets when hardware processing future packets. As a result, while there is traffic being received, software could not provide the TCP sequence number of the next expected TLS record. This is what we call the “race between software and hardware”.

In Figure 3 we show an example of this race. Assume records R1-4 are sent on the wire. First, packets P1-3 arrive in one burst, and packest P4-6 arrive at a later time. While software processes P1-3, hardware receives and processes P4-6. Software only knows about R1 and R2, while hardware already processed some of R3 and it knows the location of record R4. Therefore, software does not have sufficient information to update the state of hardware to process the next record, e.g. R4.

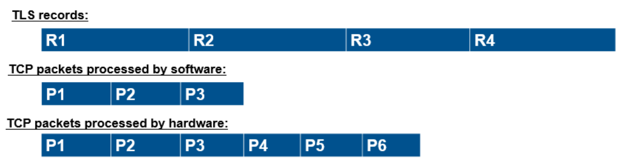
TLS records:
R1
R2
R3
R4
TCP packets processed by software:
P1
P2
P3
TCP packets processed by hardware:
P1
P2
P3
P4
P5
P6

Figure 3: Race condition between software and hardware. While software is starting to process TLS record R2, hardware is already in the middle of reading TLS record R3. Therfore, software cannot resynchronize hardware without some hardware assistance.

Control Path

The control path is based on an extension of the kTLS[5][6] control plane.

In response to a user offload request, kTLS calls tls_dev_add, a new NDO, for the netdevice used by that socket. kTLS provides the following parameters to the tls_dev_add NDO:

- The socket
- The crypto parameters
- The TCP sequence of the start of the next expected TLS record to be received.

If the device can offload this TLS session, the function returns success. From this moment onwards, any packet received over that socket can be plaintext. The device will track TCP sequence numbers, decrypt and authenticate all packets received from this socket.

The sk_destruct function of the TCP socket is replaced to free resources related to TLS in the socket layer. Similarly, kTLS goes on to call another new NDO called ktls_dev_del, in order to free device driver and hardware resources.

TLS also supports key renegotiation during a session. The renegotiation in TLS1.2 is based on an encrypted TLS handshake where cryptographic material is exchanged. Eventually, the change cipher spec message is sent by each party to mark that the next packet will be encrypted using the new keys.

During renegotiation, the NIC might not identify the CCS record type. As a result a single record after the CCS, which is encrypted using the new key, is decrypted using the old key and its authentication check fails. We fix this in kTLS when new keys are added by the userspace handling the renegotation. kTLS will remove the old offload and go over all socket buffers of the TLS record after CCS in the receive queue reversing decryption offload.

Data Path

Each SKB that participates in Rx TLS offload must provide two additional bits of metadata to the kTLS layer:

- tls_processed: Was this packet processed by the TLS accelerator?
- tls_success: Was this packet processed successfully?

Packets with mismatching metadata bits must not be coalesced at any layer except kTLS.

The data path consists of a fast path and a slow path. In the fast path all packet are decrypted by hardware, and decryption is skipped entirely.

The following pseudo code is performed by the kTLS layer for each record received:

```
1. Initialize:
    a. partial_decrypt = 0; resync = 1;
2. Go over all socket buffers in the TLS record:
    a. If skb is not tls_processed:
        i. partial_decrypt = 1;
    b. If skb is tls_processed and not
       tls_success:
        i. Return authentication error.
    c. If skb is tls_processed and tls_success:
        i. resync = 0;
3. Else If resync: //fully encrypted record received
```

a. Call dev->tls_rx_resync(..)
b. tls_sw_decrypt_and_auth_record(..)

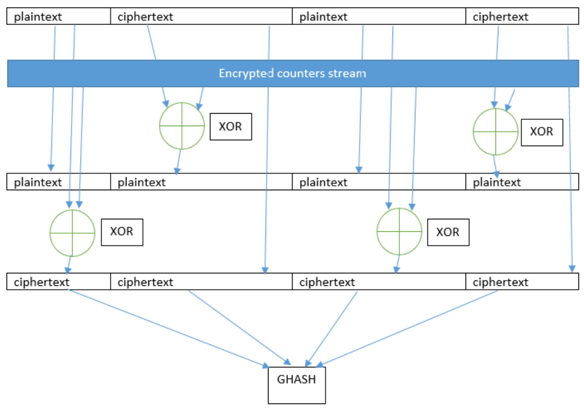

Figure 5: Partial decryption of a TLS record which consists of plaintext and ciphertext packets. Both ciphertext and plaintext is obtained. The former is used for authentication and the latter is provided to the user.

4. Else If partial_decrypt: // Mixed plaintext and ciphertext
    a. tls_partial_decrypt_and_auth(..)
5. Copy plaintext to userspace.

Resynchronization

When the TLS offload accelerator experiences significant out-of-order it might lose the TLS record framing inside the TCP stream. This will prevent further offload until the context is resynchronized (resync). The kTLS layer could identify this by receiving a fully encrypted TLS record header while using TLS Rx crypto offload. Resync requires software to provide hardware with a new expected TCP sequence number of a TLS record and the corresponding TLS record sequence number.

Partial Decrypt

Due to reordering some packets are unmodified while others are decrypted. As a result, kTLS must validate the authentication and decrypt TLS records that consist of some ciphertext and some plaintext packets.

In AES-GCM we need to obtain the ciphertext to authenticate the record. We do this by encrypting the payload of each decrypted packet. AES-GCM encryption is performed via a XOR of the data with the keystream, which is generated using a counter starting from the TLS record IV. The ciphertext is processed by the GMAC algorithm to produce the ICV which is compared to the authentication tag on the wire. Also, packets that are received encrypted need to be decrypted to get a plaintext TLS record. Overall, this partial decryption operation requires only a single pass over the TLS record, because each packet is XORed with the keystream once to get either the ciphertext or the plaintext, and all the ciphertext goes through the GMAC algorithm.

In Figure 5, we show and example of a TLS record that consists of 4 packets. The ciphertext and plaintext packets are interleaved. Partial decryption will authenticate and decrypt the record in a single pass over the data.

Note1: The authentication tag on the wire is never modified.

Note2: We fallback to software decryption when the entire record is ciphertext.

Note3: These ideas can be adjusted to CBC with some modifications.

SKB Coalescing

SKBs with mismatching tls_processed/tls_success bits cannot be merged. This might be a problem when TCP attempts to prune its receive queues. A possible solution is to call kTLS to perform partial decryption on these SKBs. After partial decryption these bits could be reset and SKBs could be coalesced.

Conclusion

We suggest a kernel API for TLS receive side offload, and provide an initial performance evaluation. TLS offload improves performance by at least 3x over current state-of-the-art kernel implementation, reducing per packet CPU overhead and enabling the use of encryption in high throughput. Receive-side crypto offload will improve the throughput of TLS connections. Further improvements gained by receive side offload is yet to be evaluated.

References

1. “The Fully Encrypted Data Center”, Oracle Technical White Paper, accessed September 22, 2016, http://www.oracle.com/technetwork/server-storage/hardware-solutions/fully-encrypted-datacenter-2715841.pdf

2. “Server Farm Security in the Business Ready Data Center Architecture”, Cisco design guide, Chapter 6 “Catalyst SSL Services Module Deployment in the Data Center with Back-End Encryption” http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/ServerFarmSec_2-1/ServSecDC/DC_Pref.html

3. “Why TOE is bad?”, accessed September 22, 2016. https://wiki.linuxfoundation.org/networking/toe

4. “Intel QuickAssist”, accessed September 22, 2016. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/quickassist-adapter-8950-brief.pdf

5. Optimizing TLS for “High-Bandwidth” Applications in FreeBSD, R. Stewart, et. al. accessed September 22, 2016. https://people.freebsd.org/~rrs/asiabsd_2015_tls.pdf

6. Crypto Kernel TLS socket, D. Watson, accessed September 22, 2016.
https://lwn.net/Articles/665602/

7. RFC 4303: IP Encapsulation Security Payload (ESP), S. Kent, accessed November 1, 2016,
https://www.ietf.org/rfc/rfc4303.txt

8. QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2 draft-hamilton-early-deployment-quic-00, J. Iyengar, et. al. accessed November 1, 2016.
https://tools.ietf.org/html/draft-hamilton-early-deployment-quic-00

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

10. TLS Offload to Network Devices, B. Pismenny, et. al., accessed October 19, 2017.
https://www.netdevconf.org/1.2/papers/netdevconf-TLS.pdf