TLS performance characterization on modern x86 CPUs

Pawel Szymanski, Manasi Deval

Intel Inc.

Abstract

The volume of network traffic protected by using various encryption and authentication methods is growing year by year. One of the most popular method used for this purpose is Transport Layer Security (TLS) protocol. This trend generates growing interest in off-loading TLS data record encryption and authentication from CPU to network controller.

This talk presents results of performance measurements for the user and kernel processing of TLS. It characterizes and compares user mode TLS, KTLS with write system call and KTLS with Sendfile system call to contrast various bottlenecks in each one with regards to encryption and authentication, cost of system calls and the memory bandwidth. The experiments further compare application behavior in simple web-serving model versus a Video-on-Demand (VoD).

The detailed characterization can help a deployer / user to decide which of the three software-based encryption methods are most suitable for a given traffic profile.

Keywords

TLS, AES, performance, networking

Introduction

TLS [1] is a protocol widely used in the Internet to secure network traffic based on TCP. In particular it is used by HTTP servers serving variety of content such as simple web pages or Video-on-Demand streams.

There are two major implementation options available for applications running under Linux as shown in Figure 1. The first option is implementation of entire TLS protocol functionality as a user space library (for example openssl). It is called User Space TLS in this paper. The second option is an implementation split between user space library and Kernel TLS module. Its is called KTLS in this paper. In this option the user space library is responsible for establishing TLS connection, while Kernel TLS module divides the application data into TLS records and runs the data through cryptographic functions such as encryption and authentication.

One of the major advantage for KTLS option is the ability to use sendfile() system call. The call allows to transfer content of a file via network without copying content of the file into user space. Another advantage of KTLS is the fact that Kernel component running above TLS module have access to plain text of the messages. This enables implementation of BPF programs that can act upon the data. Similarly Kernel Connection Multiplexer (KCM) can be used to make intelligent scheduling choices based on content of data received from TLS connection.

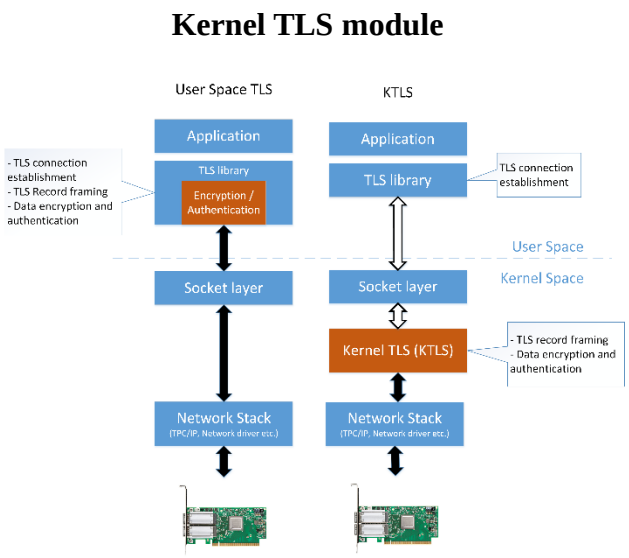

Figure 1. TLS implementation options

As shown in Figure 1, on TX path Kernel TLS module receives application data from socket layer, divides the data into TLS records, encrypts them and passes them to the reset of network stack. The module does not implement cryptographic primitives on its own but uses kernel cryptographic modules. KTLS module supports two cipher options: AES_GCM_128 and AES_GCM_256.

In the simplest case when an application wants to send data via TLS connection, it passes the data to a TLS library. Next the library invokes write() system call to send the data into the TLS module. When the cryptographic module is about to encrypt the data, it reads the plain text data from buffers located in user space and puts the cipher text into buffers located in Kernel space. In this paper such flow is called KTLS Write.

If the data to be sent via TLS connection resides in a file, the application must first read the data from the file to a user space buffer and next use write() system call to pass the data to TLS and network stack. This inefficient due to two reasons. First, such flow requires two data copy operations - the data must be moved from page cache to user space buffer(s) and from user space to network buffers in the kernel. The latter copy is done as part of encryption process. Secondly the application must allocate memory for the user

space buffers. The application could either allocate a buffer of size equal to the file size or use a smaller buffer to reduce memory consumption. If the latter option is applied, the application must invoke write() syscall multiple times per file, which brings an additional overhead.

The flow can be optimized by using sendfile() system call. The call requests kernel to transfer a file via TLS connection without copying the data into user space. The kernel reads the data from storage device to the page cache and passes the pages to TLS module, which deals with encrypting the data and sending it to network stack. In this paper such flow is called KTLS Sendfile. Note that during the encryption process the data is copied from page cache to network buffers, so it avoids one memory copy operation comparing to KTLS Write case.

Test Setup

Hardware Configuration

The machines used to perform measurements are dual-socket Intel® Xeon® Gold 6142 CPU (2.60 GHz). They have 2 NUMA nodes with 16 cores each. The systems have 376GB of RAM (188GB per NUMA node). In this experiment, we used Intel Ethernet Controller E810 100 GbE adapters plugged into each system and connected back-to-back.

Table 1 lists hardware settings configured using UEFI menu. The settings were applied to ensure maximum performance and repeatability of results.

Table 1.UEFI Hardware Settings

Hardware Setting	Value
Hyper-threading	Disabled
C-states	Disabled
P-states (aka EIST)	Disabled
Turbo	Disabled
CPU Power & Performance Policy	Performance
Enable CPU HWPM	Native Mode

Software Configuration

The machines used for these experiments run Ubuntu 18.04.2 with 5.1.0 Linux Kernel with KTLS support enabled.

The machine playing server role run NGINX [2] version 1.5.11 as HTTP server with OpenSSL 3.0.0-dev library [3]. Both Kernel crypto module and OpenSSL library are using AES-NI instruction set available in the latest Intel x86 CPUs. The implementations are different (i.e. they do not share common code base), but both are based on algorithms and guidelines specified in [4] and [5]. Table 2 lists values of NGINX configuration directives important to achieve maximum performance of the HTTP server.

Table 2. Key NGINX Configuration Directives

NGINX Directive	Value
ssl_session_cache	shared:SSL:10m
ssl_session_timeout	10m
access_log	Off
aio	Threads
aio_write	On
read_ahead	0
tcp_nopush	On
tcp_nodelay	Off
keepalive_requests	1000000
keepalive_timeout	10000
ssl_ciphers	AES128-GCM-SHA256

The machine playing client role run WRK 4.1.0 [6]. The tool was selected, because it allows the user to modify timing of generated request by using LUA scripts. The tool was configured to use HTTP persistent connections to eliminate the overhead of establishing TCP/TLS connections.

Intel® VTune™ was used to characterize the software behavior and analyze the cycles spent in various parts of the stack.

Test Scenarios and Results

The experiments consisted of sending HTTP Get requests from client to server machine via TLS connection and sending back content of files in HTTP Get response messages. In order to avoid the overhead of TLS and TCP connection setup, the tests were conducted with HTTP persistent connection enabled. The capability allows to transfer multiple HTTP requests using single TCP/TLS connection.

Each time we run the same test scenario with three TLS implementation options: User Space TLS, KTLS Write and KTLS Sendfile. The experiments were conducted with files of various sizes (between 1KB and 10MB). During the experiments we measured file transfer throughput and CPU utilization for various software functions and components on server machine.

Simple Web Server scenario

The first group of experiments utilized 100 HTTP connections and each connection sent subsequent HTTP requests back to back (i.e. without any delay in-between). The goal for this group of experiments was to simulate simple web server scenario.

Figure 2 shows comparison of throughput for User Space, KTLS Write and KTLS Sendfile options. Performance of User Space implementation is higher than KTLS for smaller file sizes – 1 and 4 KB. KTLS Sendfile outperforms User Space and KTLS Write starting with 64KB files. The difference between KTLS Sendfile and User Space is 30% for 10 MB files.

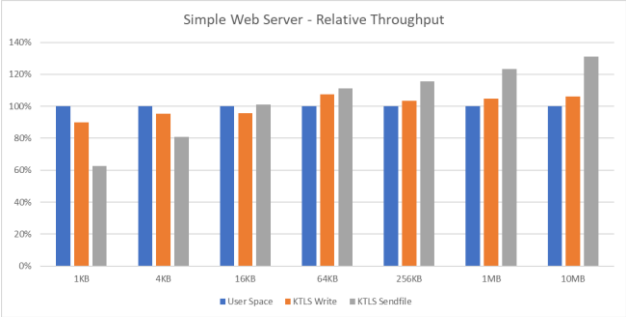

Figure 2. Simple Web Server scenario results

We expected that KTLS Sendfile will outperform the other two options. However, we did not expect that User Space performance for small files will be higher than KTLS performance. After some additional investigation performed using BPF-based utilities and profiling code using Intel VTune, we concluded that there are 2 reasons for the above results.

First, when the HTTP server sends an HTTP Get response message, it must prepare the response header in a buffer located in user space. Therefore when the server uses KTL Sendfile method, it must execute 2 syscalls: write() syscall to send the response header to the socket and sendfile() syscall to send the file as the response body. When the HTTP server uses User Space or KTLS Write, it can prepare the entire HTTP response in a single buffer and send the buffer content using single write() syscall or it can prepare the response in multiple buffers and use single writev() syscall to send multiple buffers at once. The overhead of the additional sendfile() syscall in same regardless of the file size, so it has relatively bigger impact on KTLS Sendfile performance for smaller files.

The second cause of lower KTLS Sendfile performance for smaller files sizes is the way how TLS kernel module utilizes AESNI Crypto module. Both the openssl library and the AESNI crypto driver implement AES-GCM algorithm using AESNI instructions available in Intel CPUs. Both use Karatsuba algorithm that allows to parallelize encryption process. The algorithm and high-level idea of its implementation is described in []. At the beginning of the encryption process the algorithm precomputes some frequently used values such as exponents of the hash key. The values can be reused for entire TLS connection lifetime and Openssl implementation of the AES-GCM method, which is utilized in User Space TLS option, takes advantage of the fact. In case of KTLS options, AESNI crypto module recalculates the values for each new TLS record passed for encryption from TLS module. So, it uses the values only within single TLS record. There is no mechanism that allows to keep the precalculated values between subsequent encryption requests sent from TLS module to AESNI crypto module. Again this behavior has bigger impact on performance, when the sizes of TLS records are smaller than maximum 16KB value.

VoD-like Server scenario

The second group of experiments utilized higher number of connections (between 5000 and 50000) and each connection sent HTTP requests with a delay proportional to the number of connections. The goal for this group of experiments was to simulate HTTP server delivering VoD-like content similarly to MPEG-DASH protocol.

The experiments were run with constant network throughput:

- 70 Gb/s – when files are kept in TMPFS
- 30 Gb/s – when files are kept in NVMe disk

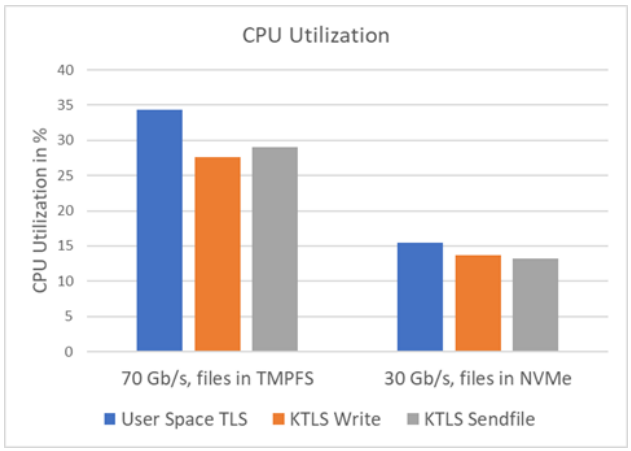

Figure 3. VoD-like Server CPU Utilization

Figure 3 shows CPU utilization measured for the two scenarios. In both cases CPU utilization for KTLS Write and KTLS Sendfile are close, while the metric for User Space TLS is higher. This is in-line with Simple Web Server results, where User Space TLS throughput is lower than KTLS.

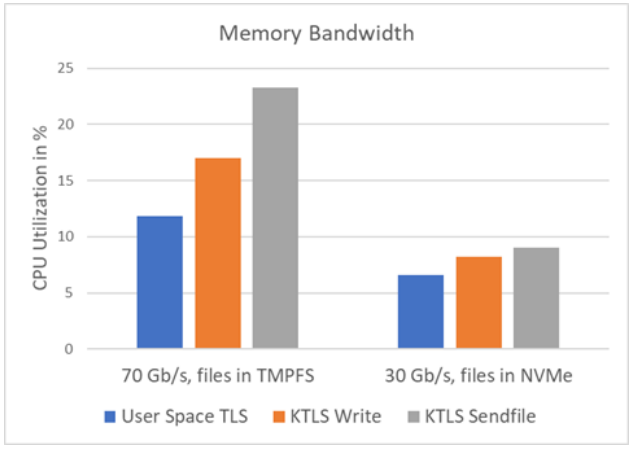

Figure 4. VoD-like Server Memory Bandwidth utilization

Figure 4 shows Memory Bandwidth utilization measured for the two scenarios. KTLS Sendfile exhibits much higher Memory Bandwidth utilization than User Space or KTLS

Write. Such results seem to be counter intuitive, because KTLS Sendfile shall generate less memory copy operations and in consequence shall generate lower traffic to system memory.

Unfortunately, we were not able to investigate the reason for such behavior. We suspect that the behavior is caused by the fact that KTLS implementation is scattered among multiple kernel modules each keeping its own control data structures and access to these data structures generates the extra traffic to memory.

Conclusion

Throughout our TLS performance characterization measurements, we found that KTLS Sendfile provides better performance both for Simple Web Server and VoD-like scenarios. For Simple Web Server scenario KTLS Sendfile outperforms User Space and KTLS Write options only in case of files equal or bigger than 64 KB.

We also found that for VoD-like scenario, KTLS options utilize less CPU cycles, but consume significantly more Memory Bandwidth.

References

[1] Eric Rescorla, et al, RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3, https://tools.ietf.org/html/rfc8446

[2] NGINX - HTTP and reverse proxy server, a mail proxy server, and a generic TCP/UDP proxy server, https://nginx.org/

[3] OpenSSL – Cryptography and SSL/TLS Toolkit, https://www.openssl.org/

[4] Vinodh Gopal, Erdinc Ozturk, Wajdi Feghali, Jim Guilford, Gil Wolrich, Martin Dixon. Optimized Galois-Counter-Mode Implementation on Intel Architecture Processors, Intel White Paper, August 2010, https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communications-ia-galois-counter-mode-paper.pdf

[5] Erdinc Ozturk, Vinodh Gopal. Enabling High-Performance Galois-Counter-Mode on Intel Architecture Processors, Intel White Paper, October 2012, https://www.intel.com/content/dam/www/public/us/en/documents/software-support/enabling-high-performance-gcm.pdf

[6] Wrk - a HTTP benchmarking tool, https://github.com/wg/wrk