To TLS or not?

That's not the question…

Pedro Tammela, Nabil Bitar, Jamal Hadi Salim

Work being done at Bloomberg

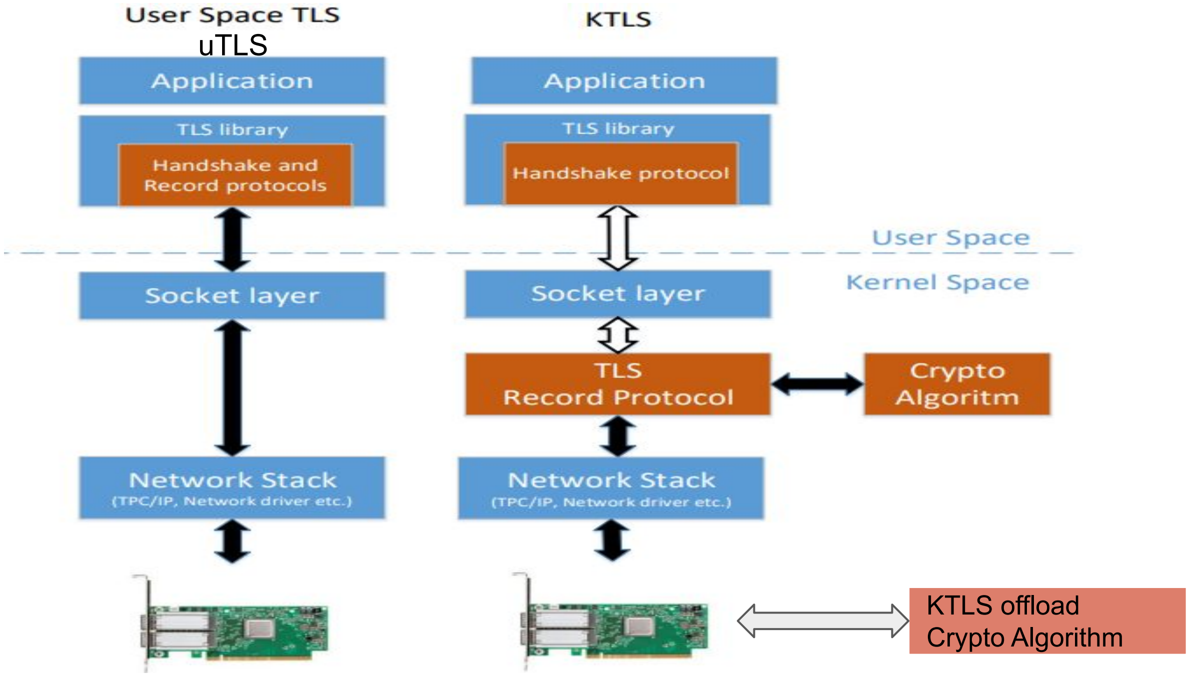
Agenda

1. TLS/kTLS overview
2. Test Setup
3. Results
4. Debugging issues
5. Summary

Our work builds on previous presentations

1. TLS performance characterization on modern x86 CPUs
   - Pawel Szymanski, Manasi Deval
   - https://legacy.netdevconf.info/0x14/session.html?talk-TLS-performance-characterization-on-modern-x86-CPUs
2. kTLS HW offload - implementation and performance gains
   - Tariq Toukan, Bar Tuaf, Tal Gilboa
   - https://legacy.netdevconf.info/0x14/session.html?talk-kTLS-HW-offload-implementation-and-performance-gain
3. Performance study of kernel TLS handshakes
   - Alexander Krizhanovsky, Ivan Koveshnikov
   - https://legacy.netdevconf.info/0x14/pub/papers/35/0x14-paper35-talk-paper.pdf

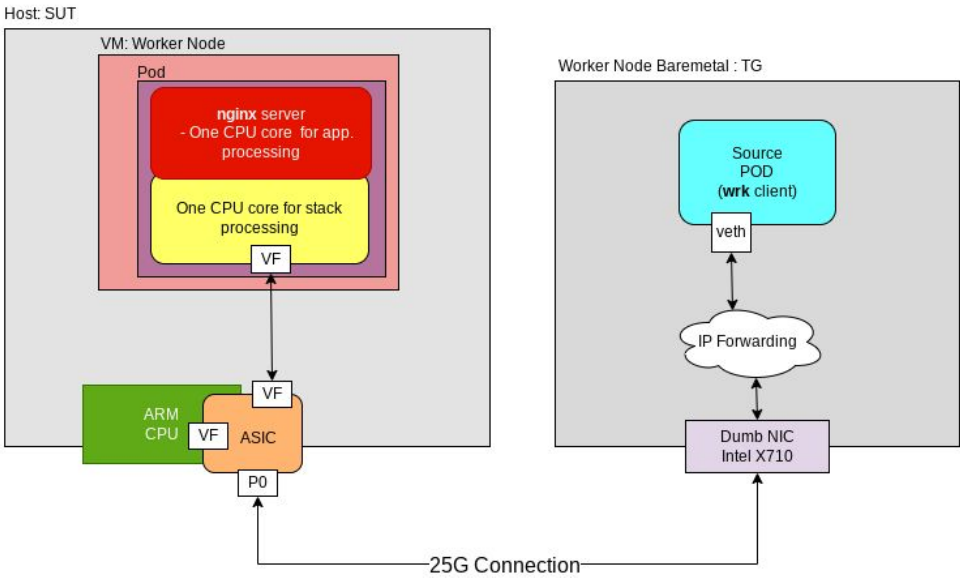
TLS Overview: User Space, KTLS, And KTLS offload



See Ref[1] and Ref[2]For credit of this illustration

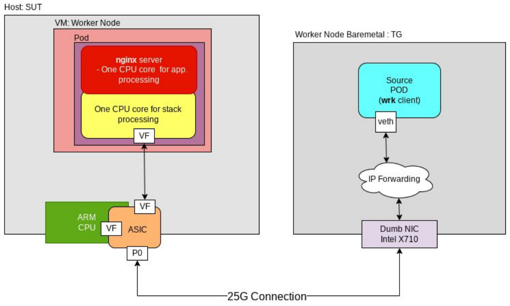
TLS Performance Testing Goals

- Test datapath crypto offload (record protocol) performance
- Nvidia Bluefield 2: only available options are **TLS1.2** and **AES 128** for kTLS offload
  - Our testing is specific to those parameters
  - We also tested disabling the CPU AES acceleration
  - We consider a TLS record size of 16KB
- Nginx was used as it supports all 3 scenarios
- Wrk is our client for HTTPS connections



TLS Performance Testing Setup

- System under test: VM with a single k8s POD running nginx server
- Client: POD with *wrk* traffic generator
  - Open two https connection
  - Request files of different sizes (for each test)
    - 1K, 16K, 32K, 64K, 128K, 1M, 1G
  - Reuse the same socket up to 1000 requests complete for each test
    - Close/open again and again (as long as the 25 seconds has not expired)
- 3 test runs, each 25s, to measure
  - Throughput
    - Measure transfer bytes/sec over the 25 seconds
  - Transactional Testing
    - Count http requests/sec accumulated over 25s
- Request RTT latency
  - How long each http request took
  - Calculate the percentiles



Test Setup

Host	Value
CPU	Xeon Gold 6230R
Hyper-Threading	N
Turbo Boost	N
RAM	192GB 2993Mhz

Hardware Setting	Value
Hyper Threading	Disabled
Turbo boost	Disabled
CPU Power & Performance Policy	Performance
KVM CPU Affinity pinning	on
GSO	on
GRO	on
TCP Segmentation Offload	on

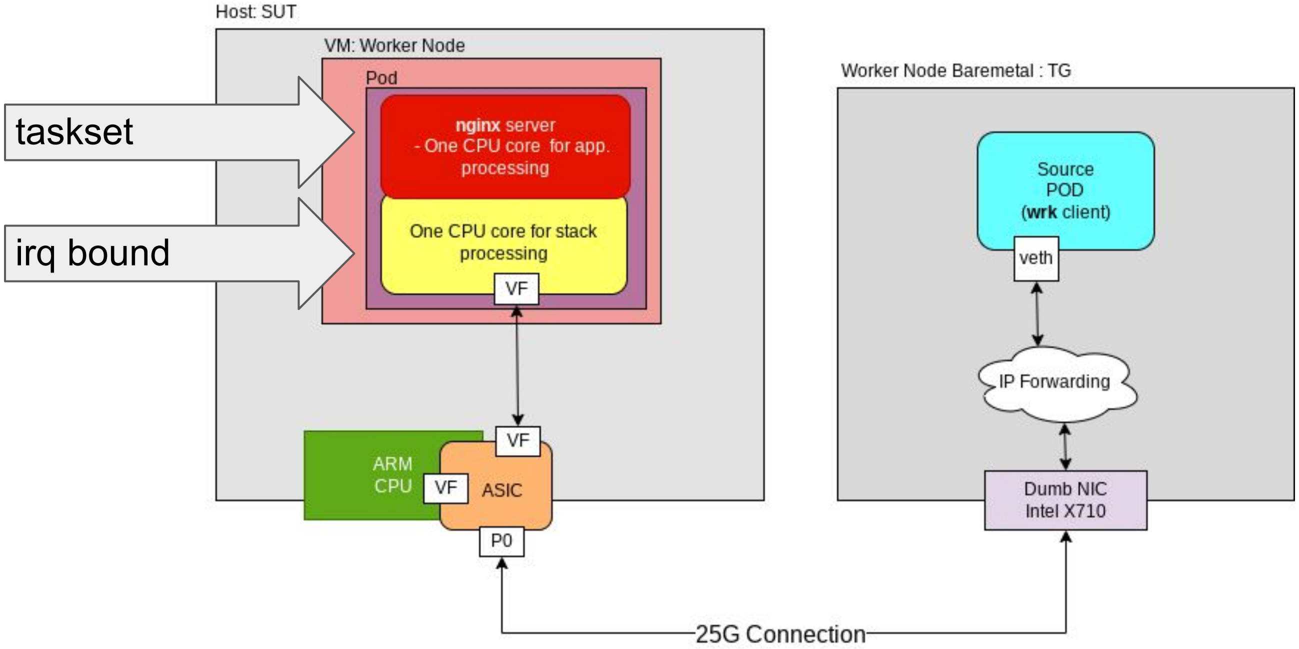
Virtual Machine	Value
Processor	Host bypass
CPUs	6
RAM	16Gb
SRIOV	on
RSS	off
RX/TX Descriptors	combined 1
rmem_max	16777216
wmem_max	16777216
rmem_default	16777216
wmem_default	16777216
tcp_rmem	4096 87380 16777216
tcp_wmem	4096 87380 16777216
tcp_mem	1638400

Test Setup

NGINX Directive	Value
worker_processes	1
sendfile	on
ssl_protocols	TLSv1.2
ssl_ciphers	AES128
ssl_conf_commands	Options KTLS
keepalive_requests	1000

Program	Version
Host kernel	5.15.10
Kubernetes	1.21.3
nginx	1.21.6
wrk	debian/4.1.0-3build1 [epoll]
VM kernel	5.17.5
OS	ubuntu 20.04

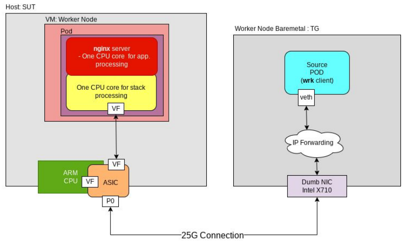
Reproducible Results: Network Vs Application CPU

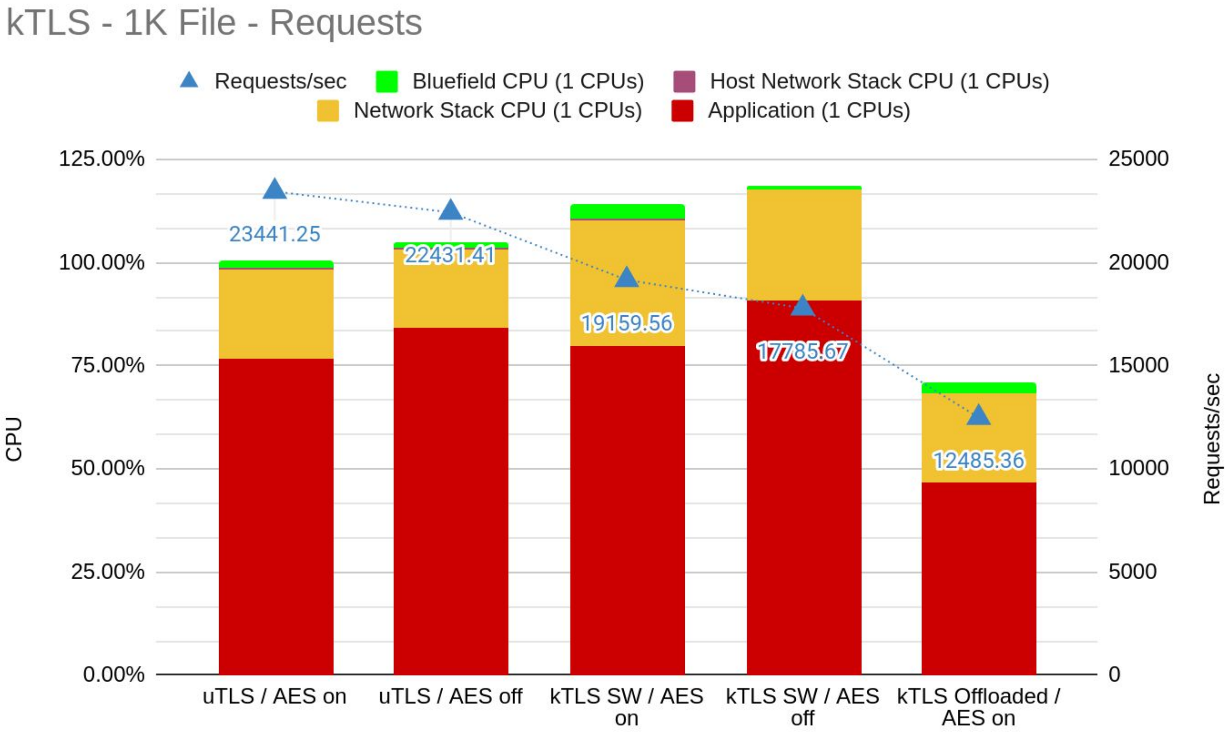


Results

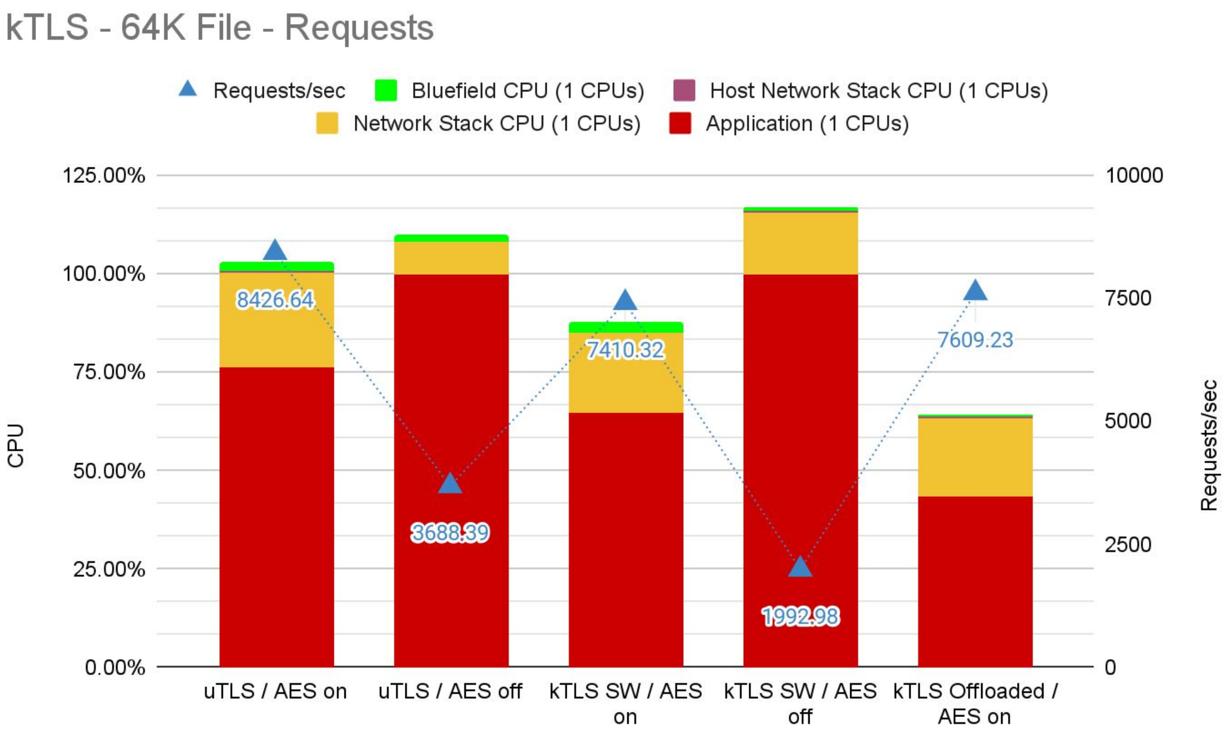
Transactional Testing

Transactional Testing: 1K files





Transactional Testing: 64K files



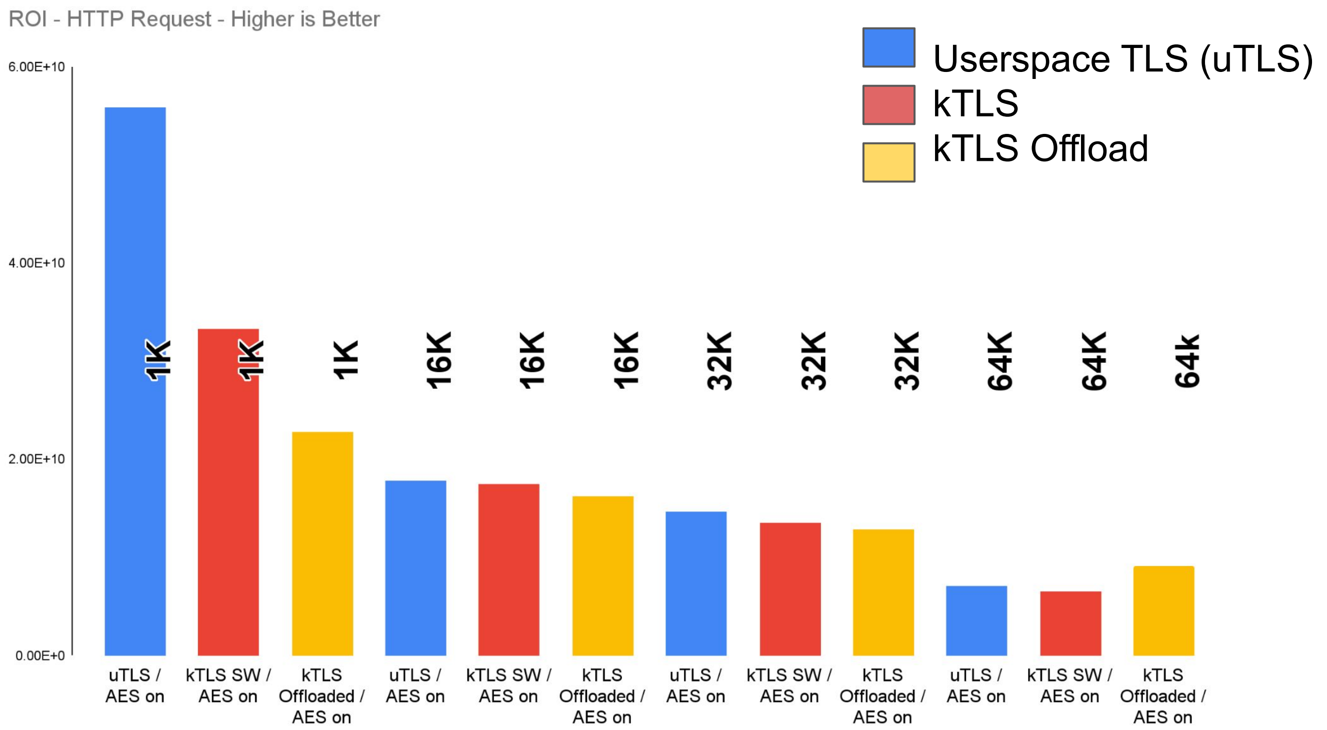
Visualising the Transactional results

- The ideal implementation consumes the least amount of CPU while producing the highest amount of transactions
  - Transactions should always strive for full link capacity
- We had to visualise the results in a way it’s obvious which implementation has the best ROI.

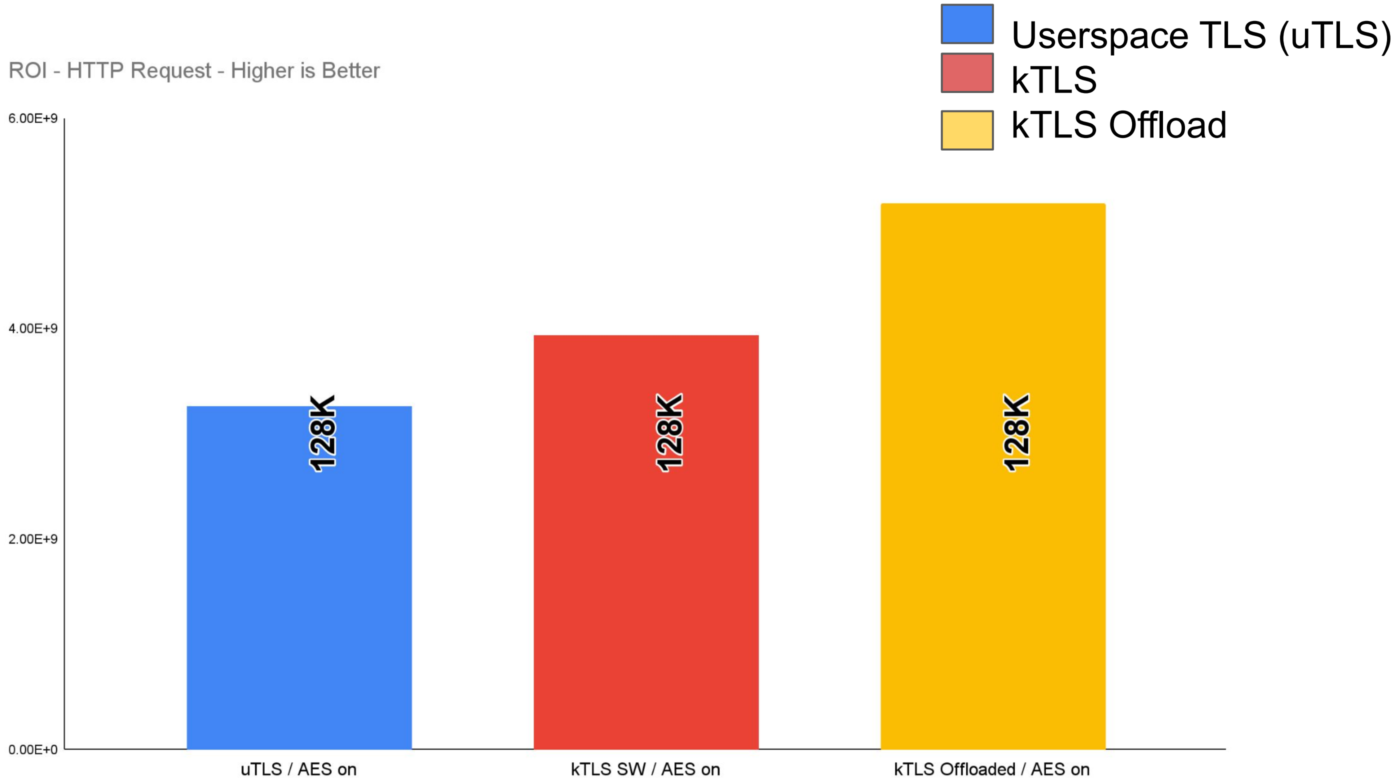
$$\frac{T^n}{\sum C_i}$$

T = Throughput (https req / sec)
n = throughput weight factor (set to 2)
Ci = CPU “i” utilization (e.g., application CPU, IO CPU)

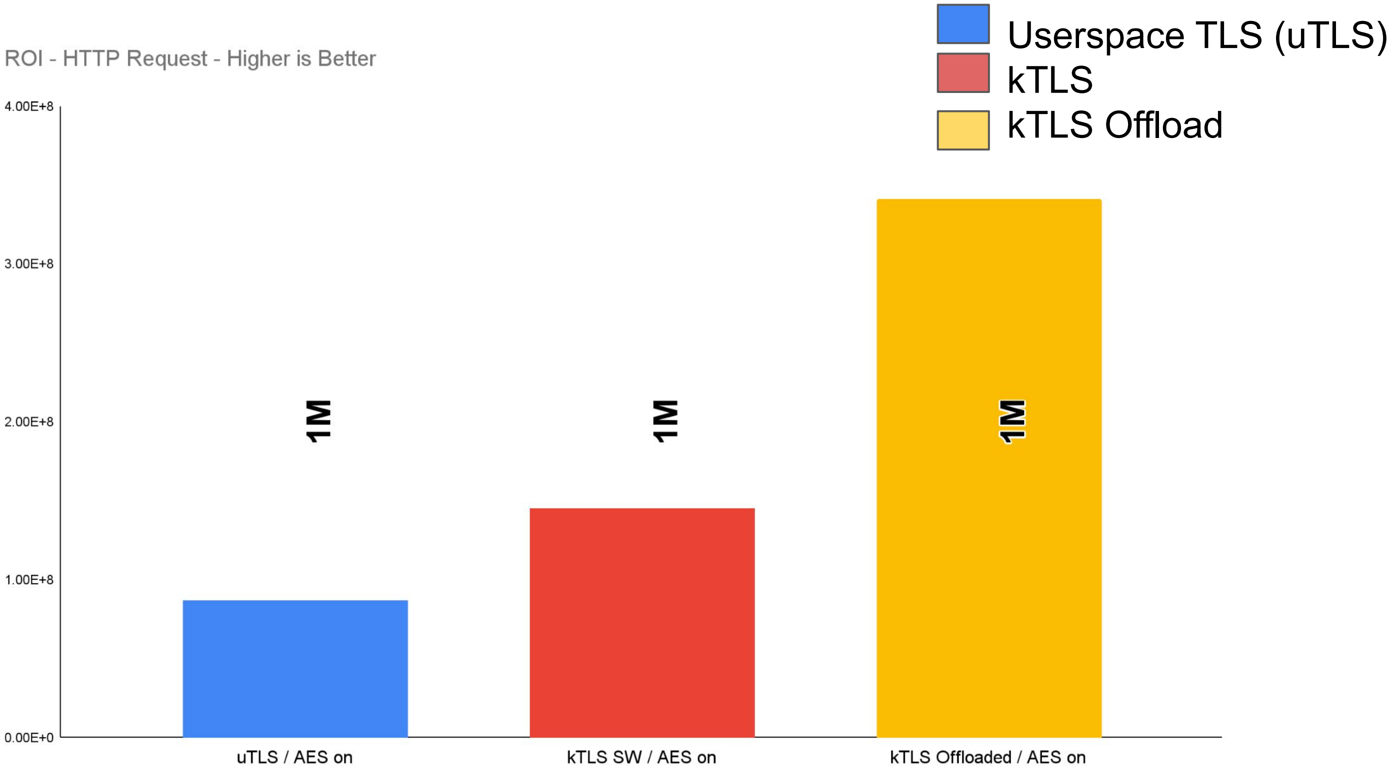
ROI - Requests (1/4) with up to 64KB size file



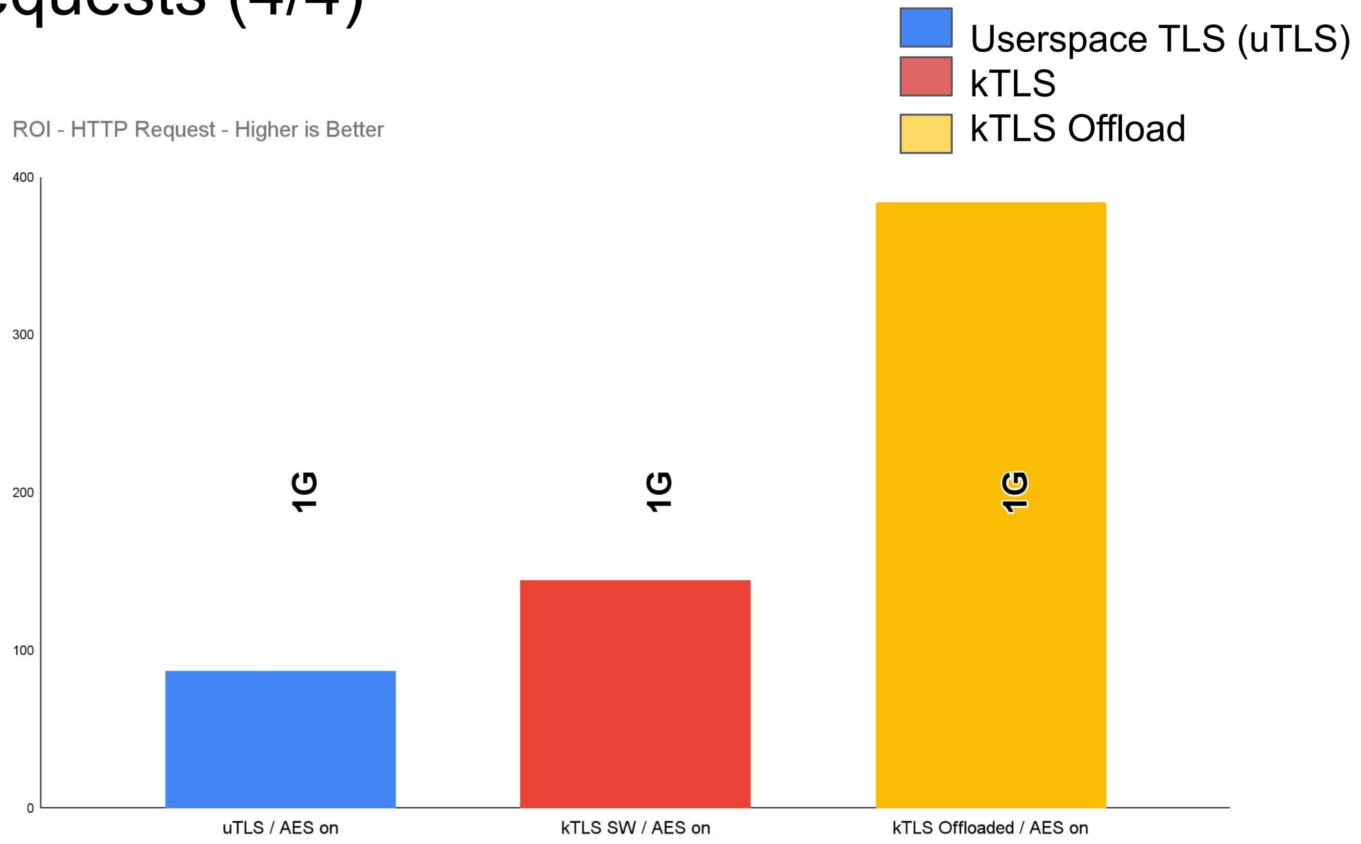
ROI - Requests (2/4) with 128KB file size



ROI - Requests (3/4) with 1MB file size



ROI - Requests (4/4)



Summary For Transactional Tests

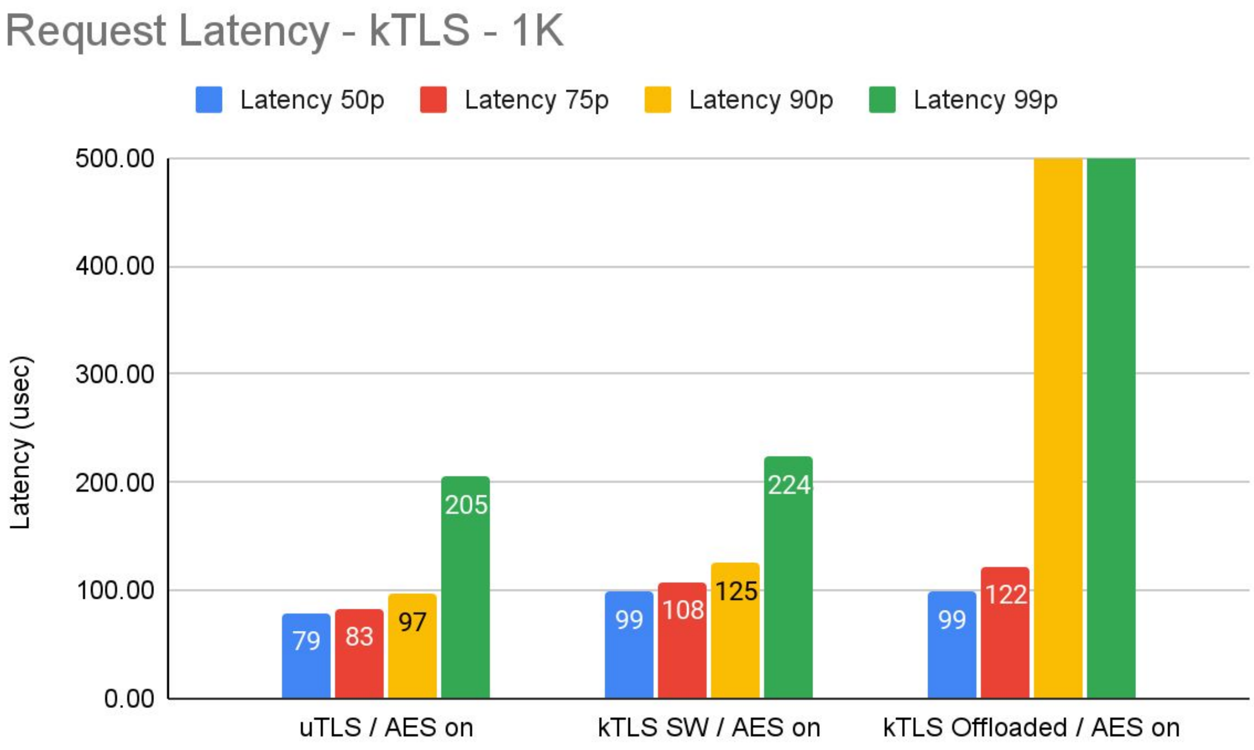
- User space TLS (uTLS) is the best implementation on short flows
- kTLS starts to show promising results after 128KB file size
- kTLS offload starts to show promising results after 64KB file size
- The CPU consumption for kTLS offload stays relatively constant across file sizes while number of handled requests improves in comparison to other implementations as file size increases
    - We saw a 35% reduction in CPU utilization accounted for the application in case of kTLS offload, when compared to other implementations
- CPU crypto acceleration in case of uTLS and kTLS provide value

Latency Testing

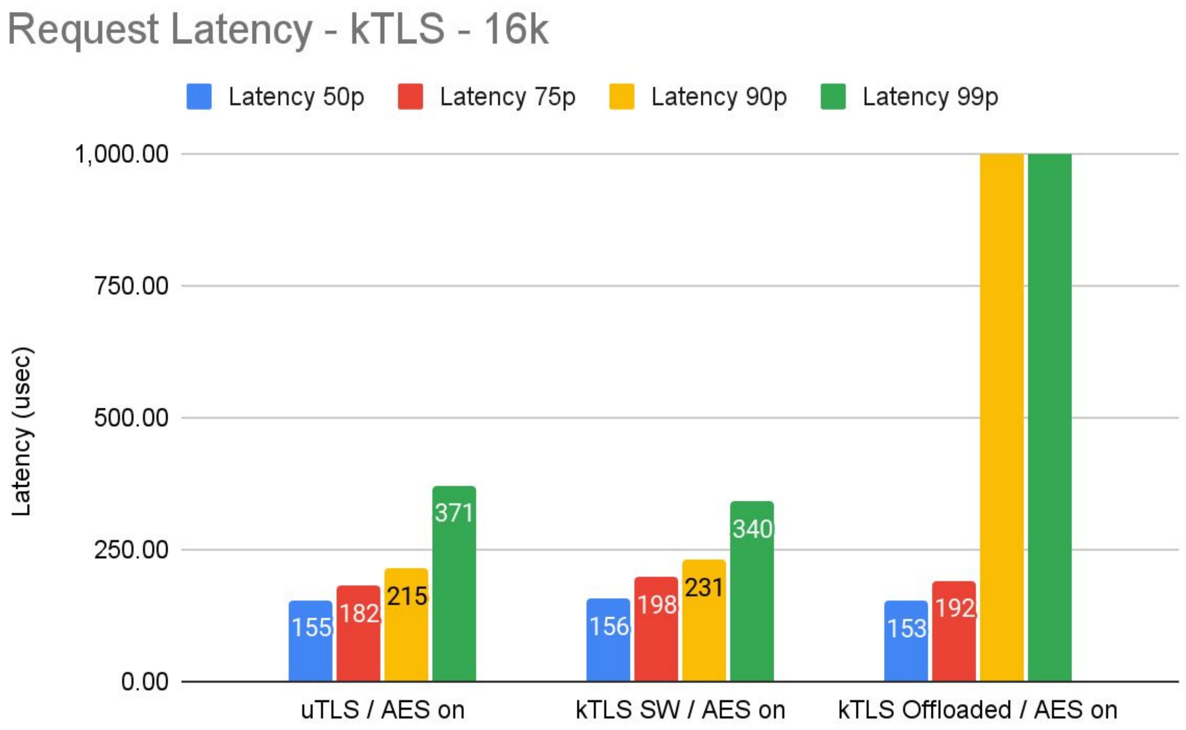
Latency: The forgotten part

- Previous Netdev conf presentations [1], [2], [3] showed similar results for throughput as we did
  - As file size increases, kTLS performs better or equal than uTLS
- None of the previous presentations discussed latency
  - **Latency matters!**
- Reminder Latency measurement comes from wrk
  - It’s the RTT of a HTTP request

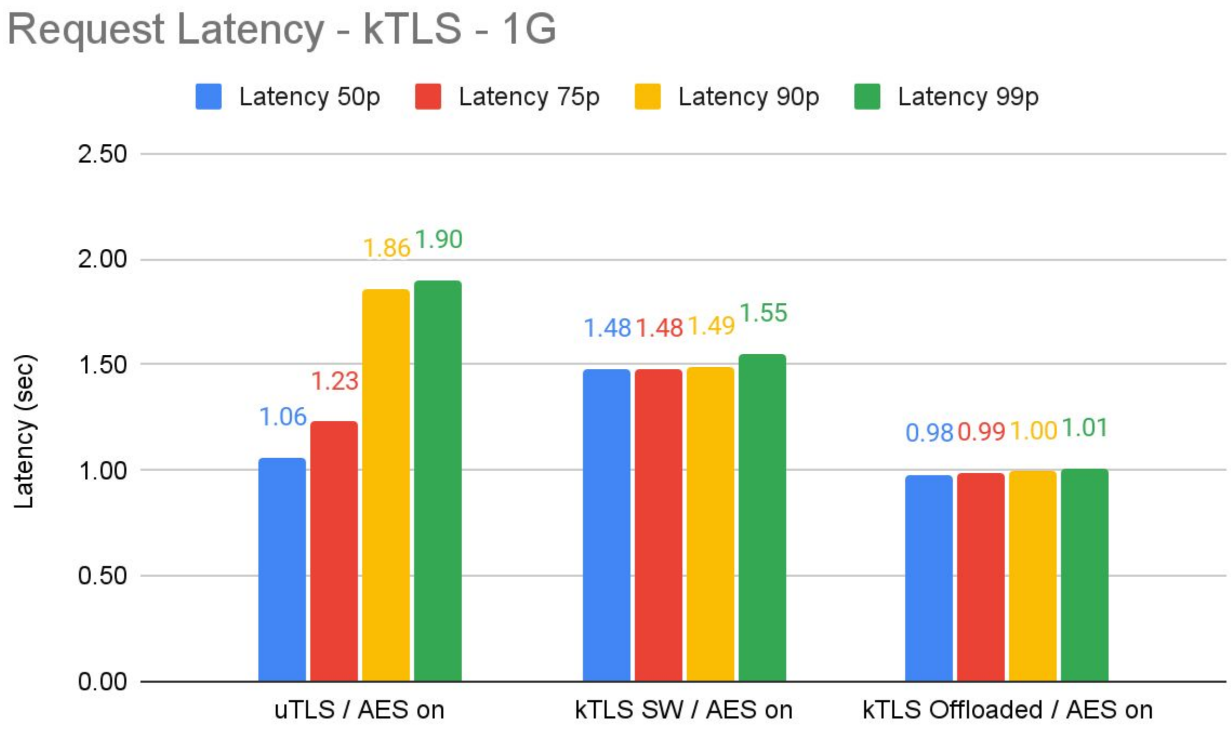
Request Latency Testing (Lower Better)



Request Latency Testing (Lower Better)



Request Latency Testing (Lower Better)



Latency Results

- Where is this 90/99p latency coming from in kTLS Offload?
  - Theory: Crypto engine setup
  - Theory: Network noise
  - Theory: Some obscure misconfiguration
  - Theory: VM Overhead

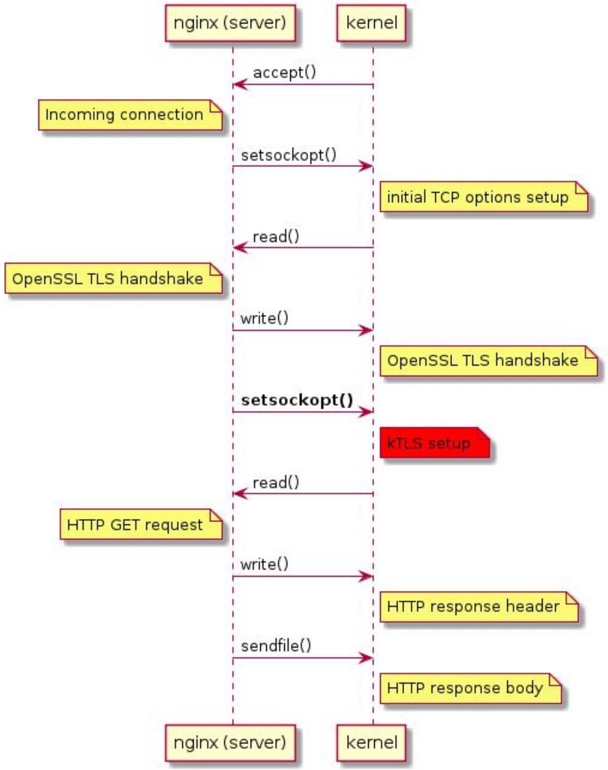
Theory: Crypto Engine Setup

- Handshake estimation:
  - Disregarding tricks like Session Resumption

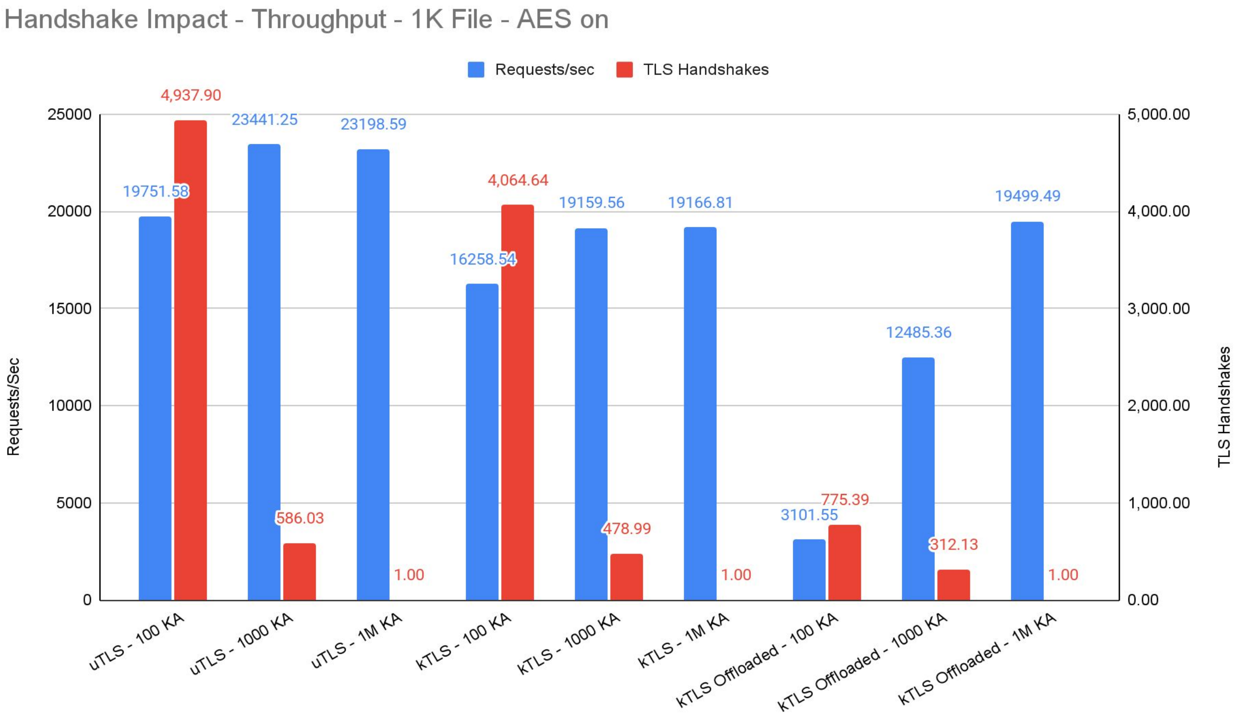
$$\approx \frac{DR}{K}$$

D = Test duration
R = Total http requests
K = Nginx keep alive parameter

- What happens if we increase K?
  - Expect to see better latency
  - Expect to see better throughput

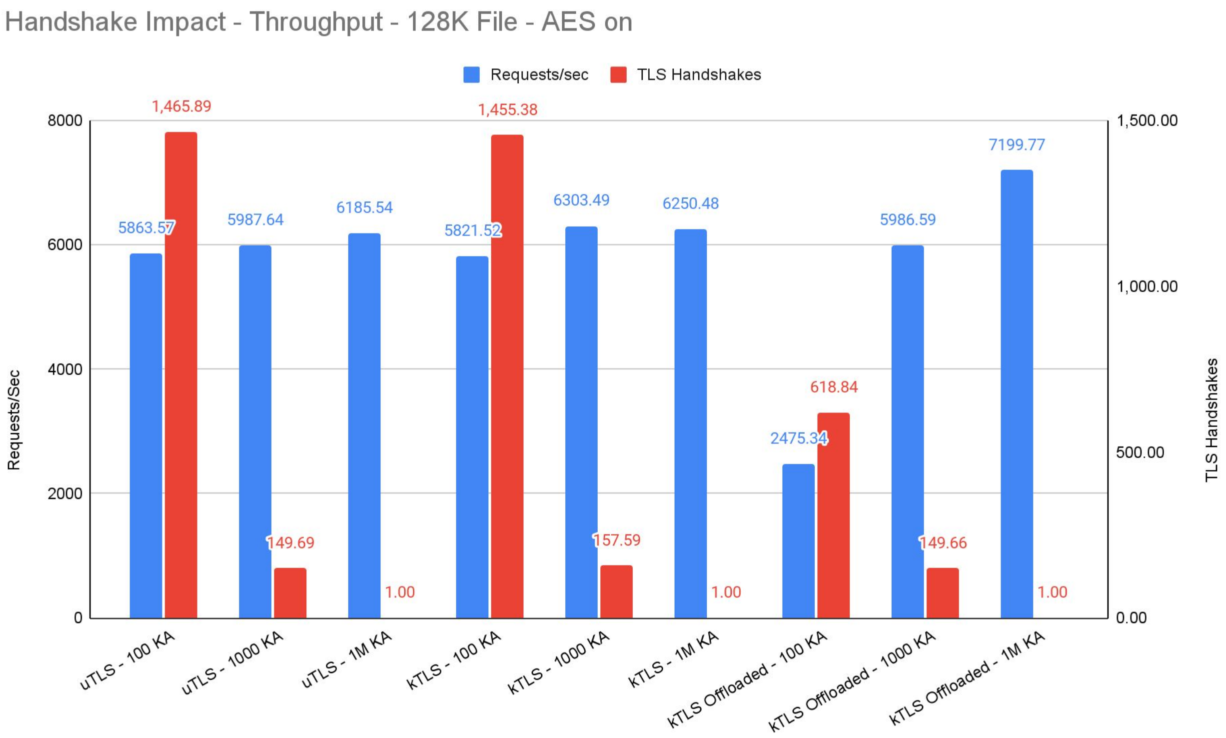


Handshake Impact: Throughput

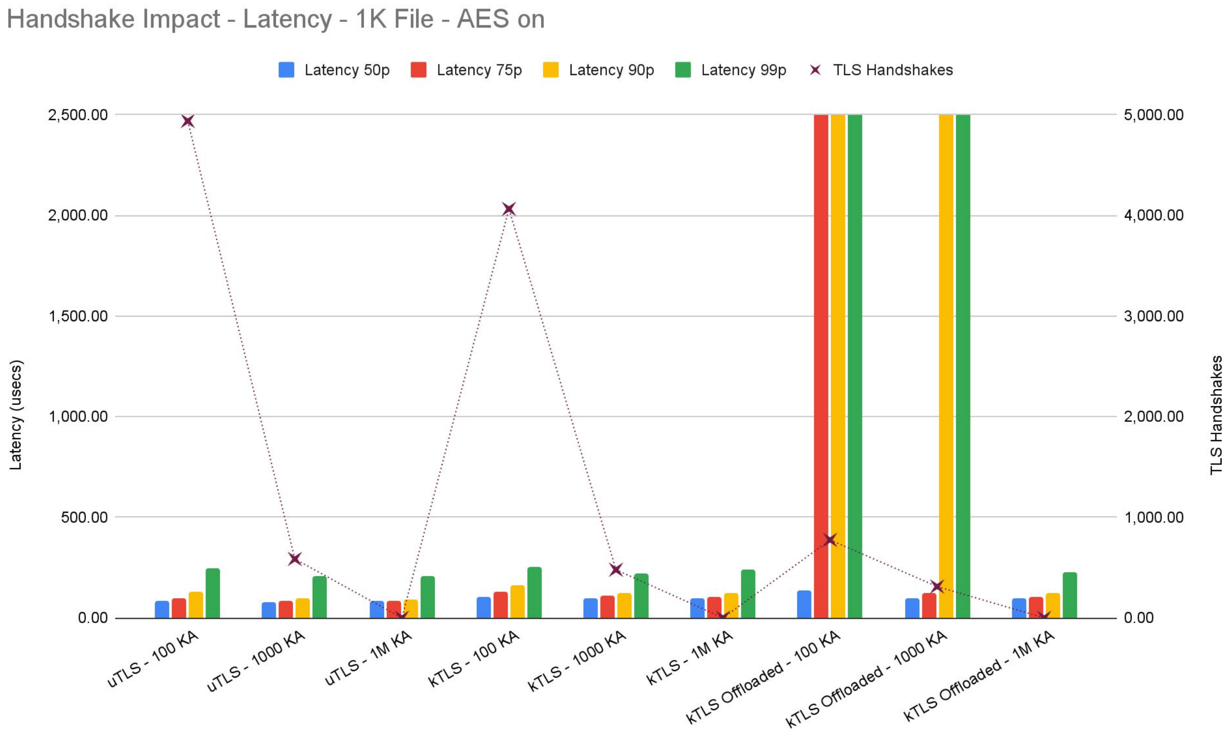


KA (Keep alive)

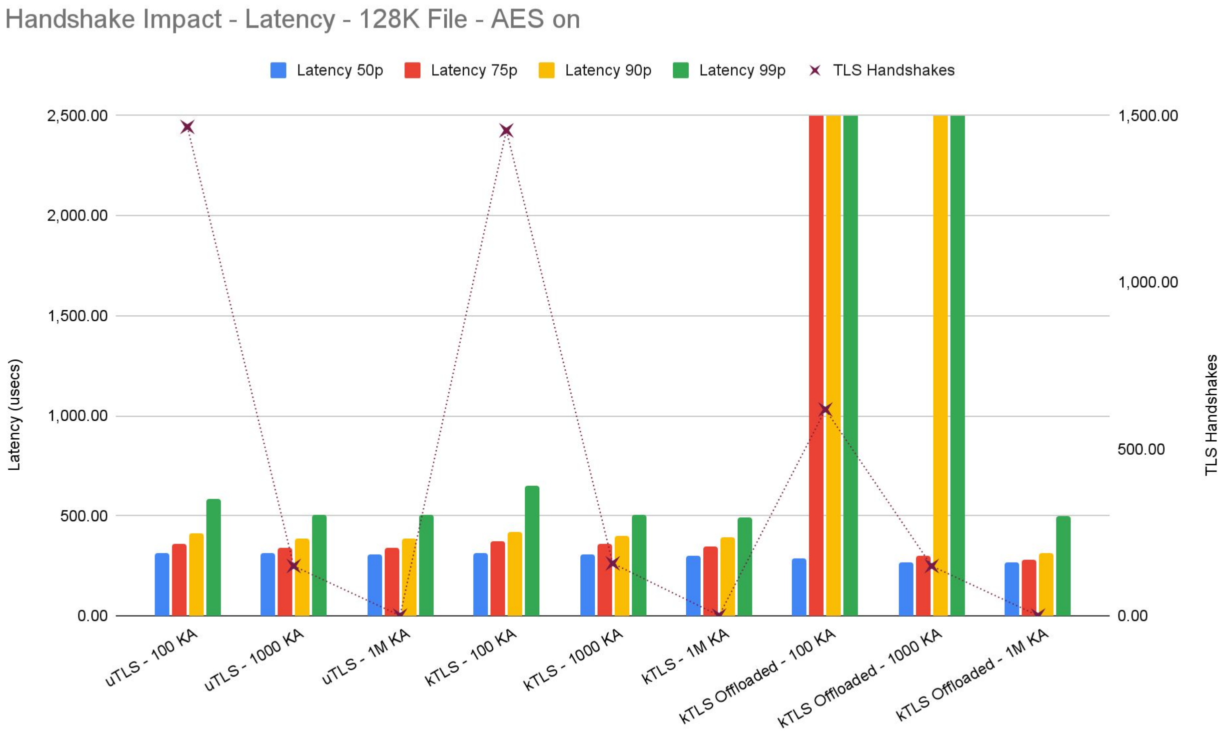
Handshake Impact: Throughput



Handshake Impact: Latency



Handshake Impact: Latency

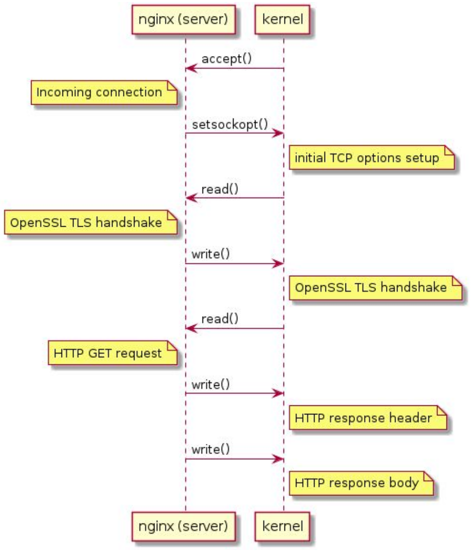


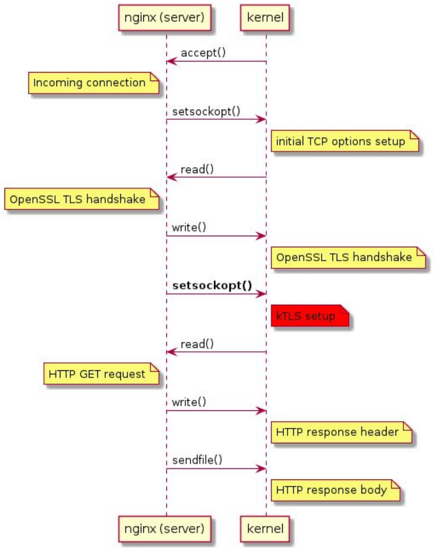
kTLS offload has a handshake setup problem!

- Clearly it influences the latency percentiles
- Quick Solution => Just run a huge keep alive constant!
  - Very dependent on application and deployment
  - Not really a satisfactory solution
- Hardware Offload brings **a lot of** savings!
  - More energy efficiency
  - More CPU for the application itself

Handshake Impact: Going deep with tracing

- By tracing nginx we can see what’s happening under the hood
  - Usually tools will collect syscall latency for convenience
- What is the cost of the socket setup for each setup?
  - `perf trace` should have the answer!
  - Remember: kTLS requires an **additional** setsockopt setup per connection





Handshake Impact: Going deep with tracing

kTLS:

…

17.884 (**0.037 ms**): setsockopt(fd: 3<socket:[21253011]>, level: TLS, optname: 2, optval: 0x7ffd1b474980, optlen: 40) = 0

…

18.005 (**0.010 ms**): setsockopt(fd: 3<socket:[21253011]>, level: TLS, optname: 1, optval: 0x7ffd1b474990, optlen: 40) = 0

…

kTLS with offload:

…

18.684 (**3.857 ms**): setsockopt(fd: 3<socket:[21233724]>, level: TLS, optname: 2, optval: 0x7ffd1b474980, optlen: 40) = 0

…

22.747 (**1.207 ms**): setsockopt(fd: 3<socket:[21233724]>, level: TLS, optname: 1, optval: 0x7ffd1b474990, optlen: 40) = 0

…

Handshake Impact: Going deep with tracing

- setsockopt() in kTLS offload is a direct call into the driver.
  - ftrace gives us this call graph:

```
=> mlx5e_ktls_add
=> tls_set_device_offload_rx
=> tls_setsockopt
=> sock_common_setsockopt
=> __sys_setsockopt
=> __x64_sys_setsockopt
=> do_syscall_64
=> entry_SYSCALL_64_after_hwframe
```

Handshake Impact: Going deep with tracing

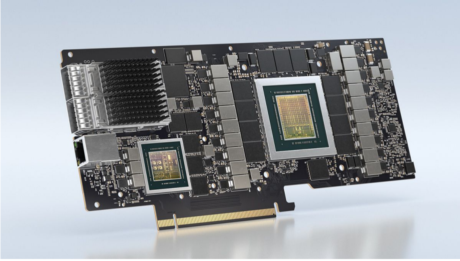
- `ftrace` tells us the culprits lies deep in the mlx5e_ktls_add_rx:

```
1485647.176662 |   4) nginx-1575808  | ...1. |                |  mlx5e_ktls_add_rx [mlx5_core]() {
1485647.176667 |   4) nginx-1575808  | d..2. |   1.136 us     |    irq_enter_rcu();
1485647.176669 |   4) nginx-1575808  | d.h2. | + 19.062 us    |    __sysvec_irq_work();
1485647.176688 |   4) nginx-1575808  | d.h2. |   0.826 us     |    irq_exit_rcu();
1485647.176690 |   4) nginx-1575808  | ...1. |   0.824 us     |    kmem_cache_alloc_trace();
1485647.176692 |   4) nginx-1575808  | ...1. |                |    mlx5_ktls_create_key [mlx5_core]() {
1485647.178577 |   4) nginx-1575808  | ..... | # 1885.080 us  |    }
1485647.178580 |   4) nginx-1575808  | d..1. |   0.563 us     |    irq_enter_rcu();
1485647.178581 |   4) nginx-1575808  | d.h1. | + 13.491 us    |    __sysvec_irq_work();
1485647.178595 |   4) nginx-1575808  | d.h1. |   0.620 us     |    irq_exit_rcu();
1485647.178597 |   4) nginx-1575808  | ...1. |                |    mlx5e_rx_res_tls_tir_create [mlx5_core]() {
1485647.180749 |   4) nginx-1575808  | ..... | # 2151.880 us  |    }
1485647.180754 |   4) nginx-1575808  | d..1. |   0.996 us     |    irq_enter_rcu();
1485647.180755 |   4) nginx-1575808  | d.h1. | + 17.549 us    |    __sysvec_irq_work();
1485647.180773 |   4) nginx-1575808  | d.h1. |   0.815 us     |    irq_exit_rcu();
1485647.180775 |   4) nginx-1575808  | ...1. |   0.471 us     |    __init_swait_queue_head();
1485647.180776 |   4) nginx-1575808  | ...1. |   1.320 us     |    _raw_spin_lock_bh();
1485647.180778 |   4) nginx-1575808  | b..2. |   1.677 us     |    post_static_params [mlx5_core]();
1485647.180780 |   4) nginx-1575808  | b..2. |   0.353 us     |    mlx5e_ktls_build_progress_params [mlx5_core]();
1485647.180781 |   4) nginx-1575808  | b..2. |   1.042 us     |    _raw_spin_unlock_bh();
1485647.180782 |   4) nginx-1575808  | ..... | # 4121.243 us  |  }
```

Summary

- The cost of the socket setup is **much higher** with hardware offload
  - Visible in the first packets of the connections which show up in the 90/99th percentile
  - As the flow size increases, hardware offload becomes more viable
    - Socket lifetime is longer
    - Resource savings are visible and show significant potential gains
  - Short flows are still problematic for either kTLS implementations

Why kTLS offload is desired?



- Reduce resource usage in host machines
  - Offload to crypto ASICs whenever supported by underlying hardware
  - Free up CPU resources for other tasks
- Leverage the sendfile() syscall for transparent encryption when possible
  - Avoid memory copies to user space
  - “Transparent” encryption when combined with kTLS

↑ Throughput
Free CPU time

↓ CPU Usage
Energy consumption

What's Next for kTLS?

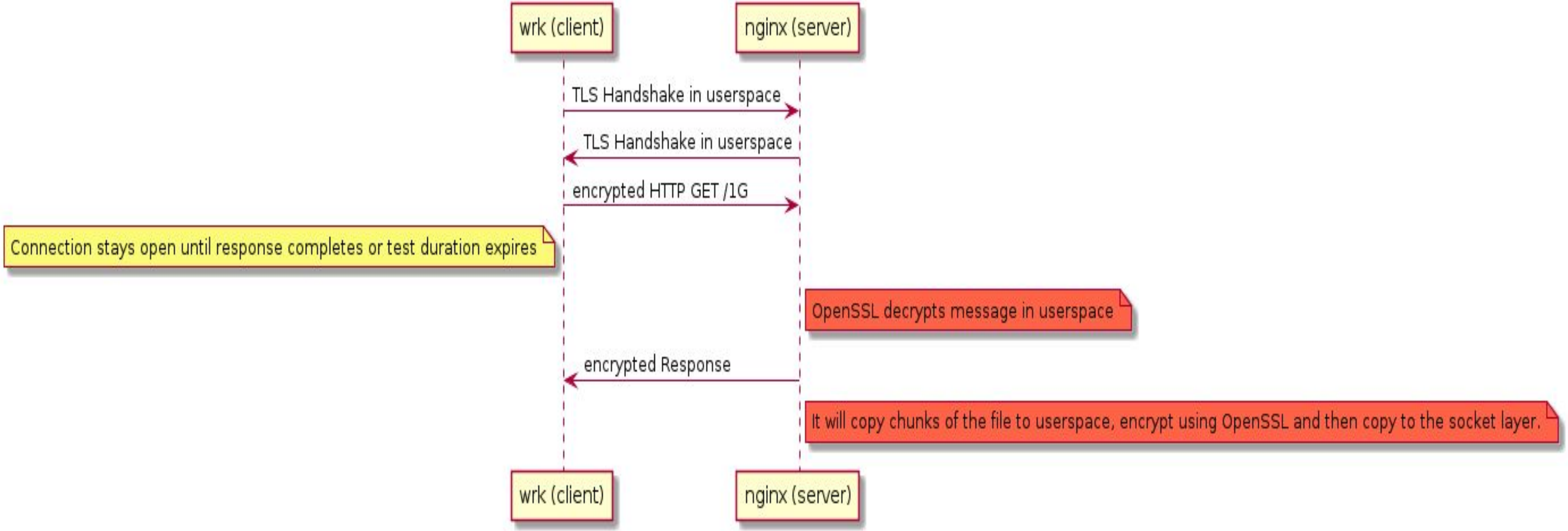
- Can we rethink the kTLS offload in order to expand beyond elephant flows?
  - Challenge: Minimize the cost of the crypto engine setup
    - Results directly in more throughput and less latency as shown in the tests
    - TLS Handshakes in the kernel?
      - Presented in Netdev 0x14
- kTLS is still not competitive with uTLS on short flows
  - Perhaps upper layer protocol was not the best approach?
    - Connection setup cost is still visible in the tests
  - More code optimizations are needed?
    - Some interesting patches popping up in the mailing list

Questions

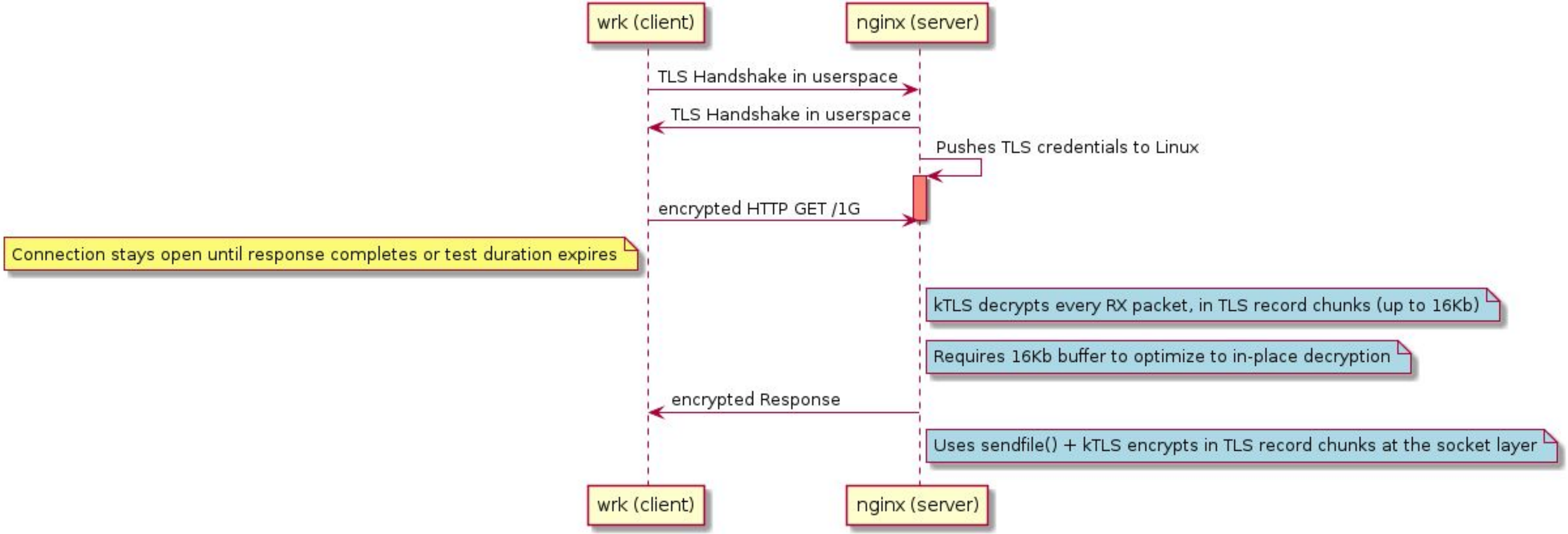
Backup slides

Implementations

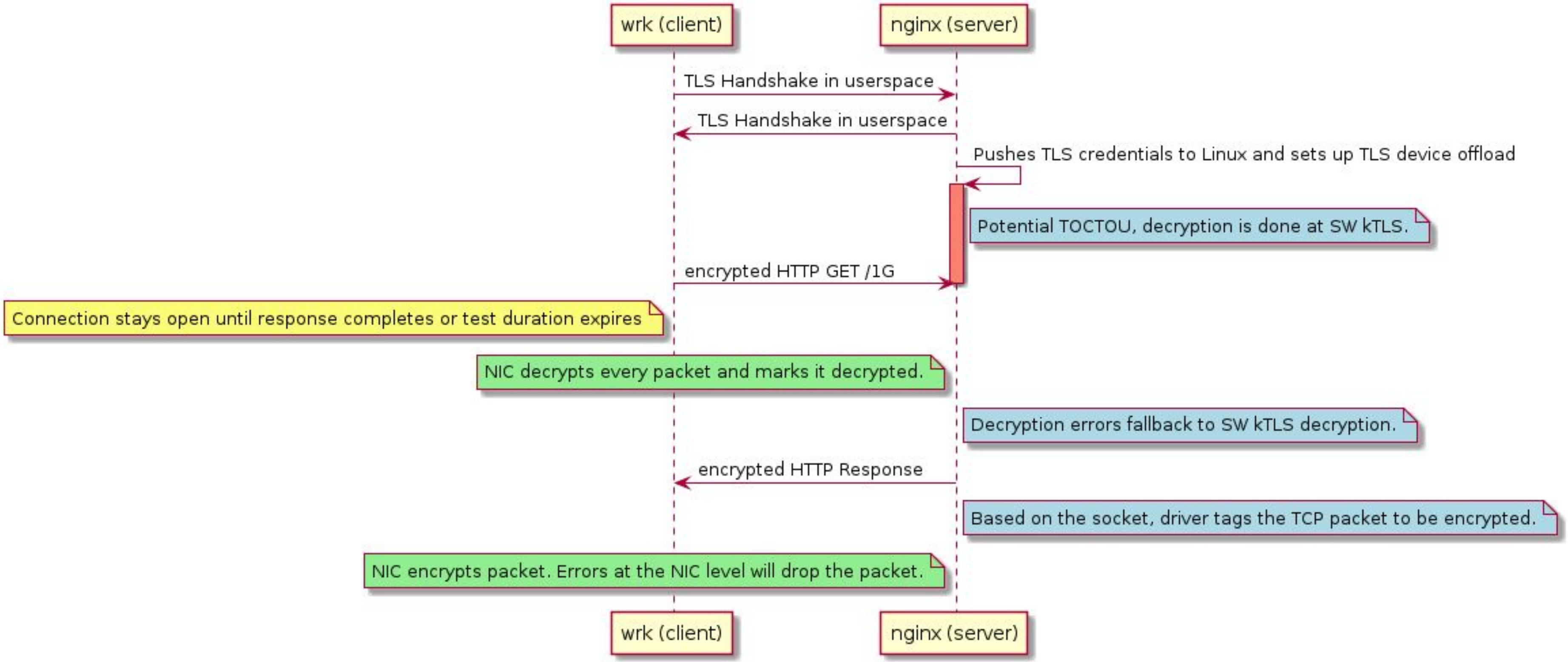
User Space TLS



Kernel TLS (KTLS)

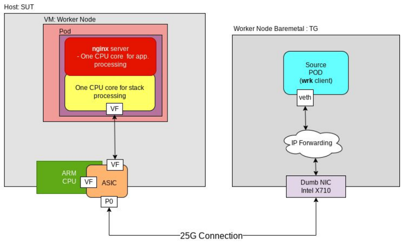


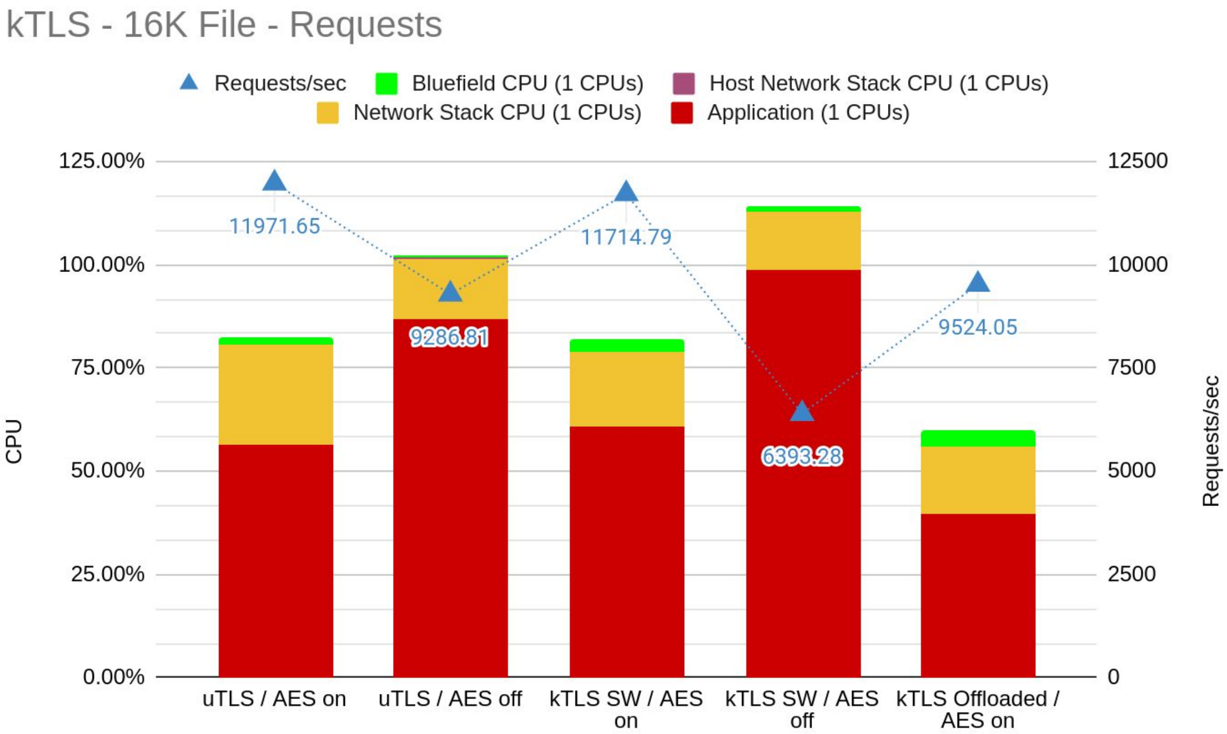
KTLS + Offload



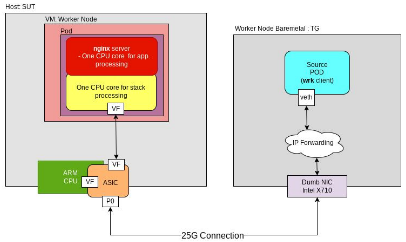
Transactional tests

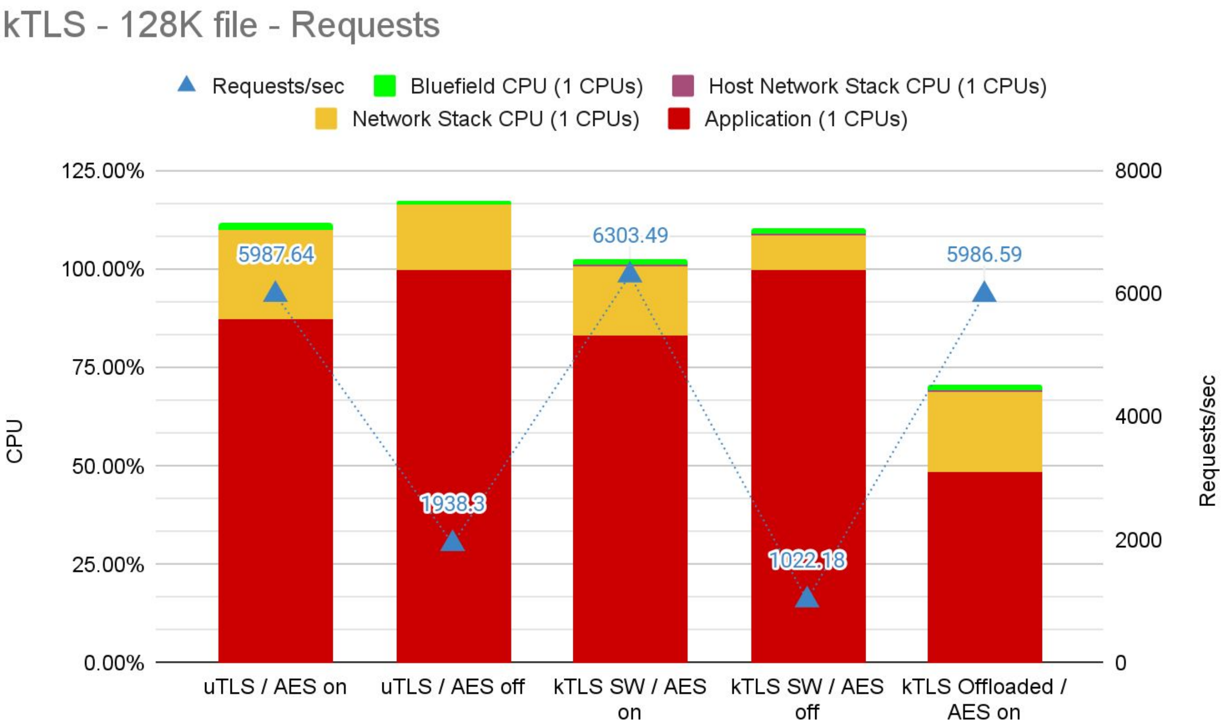
Transactional Testing: 16K files





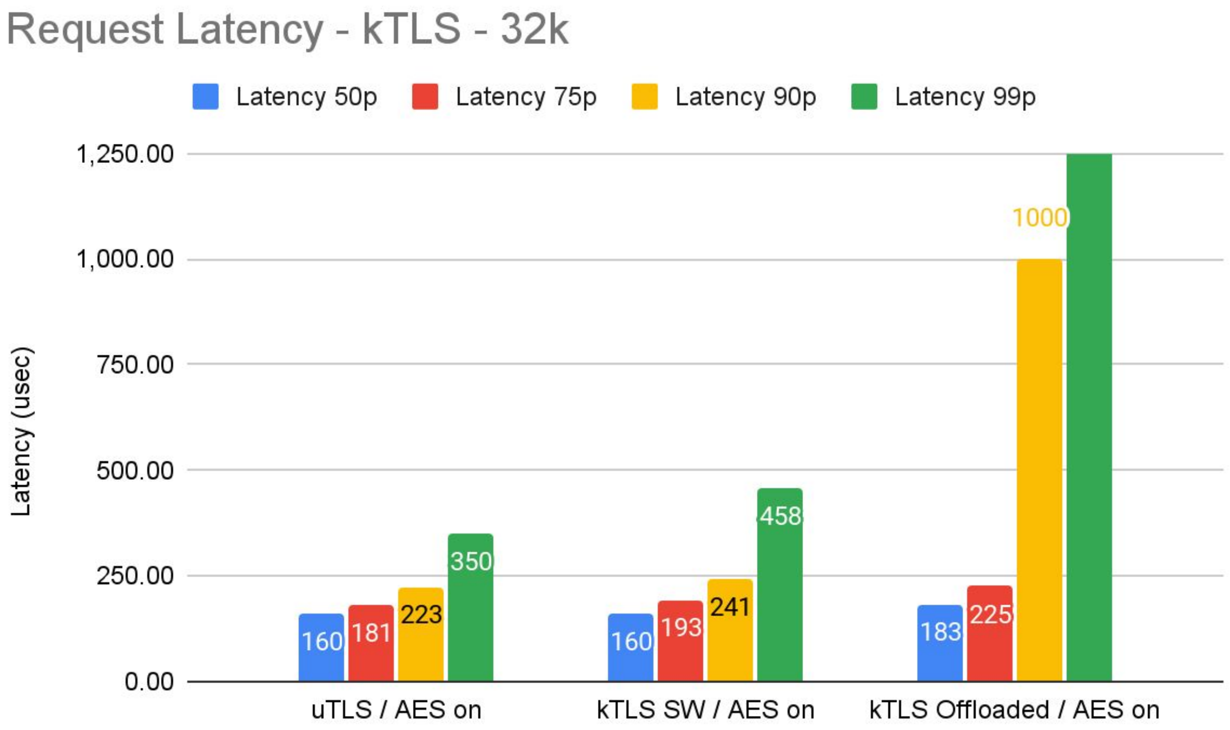
Transactional Testing: 128K files



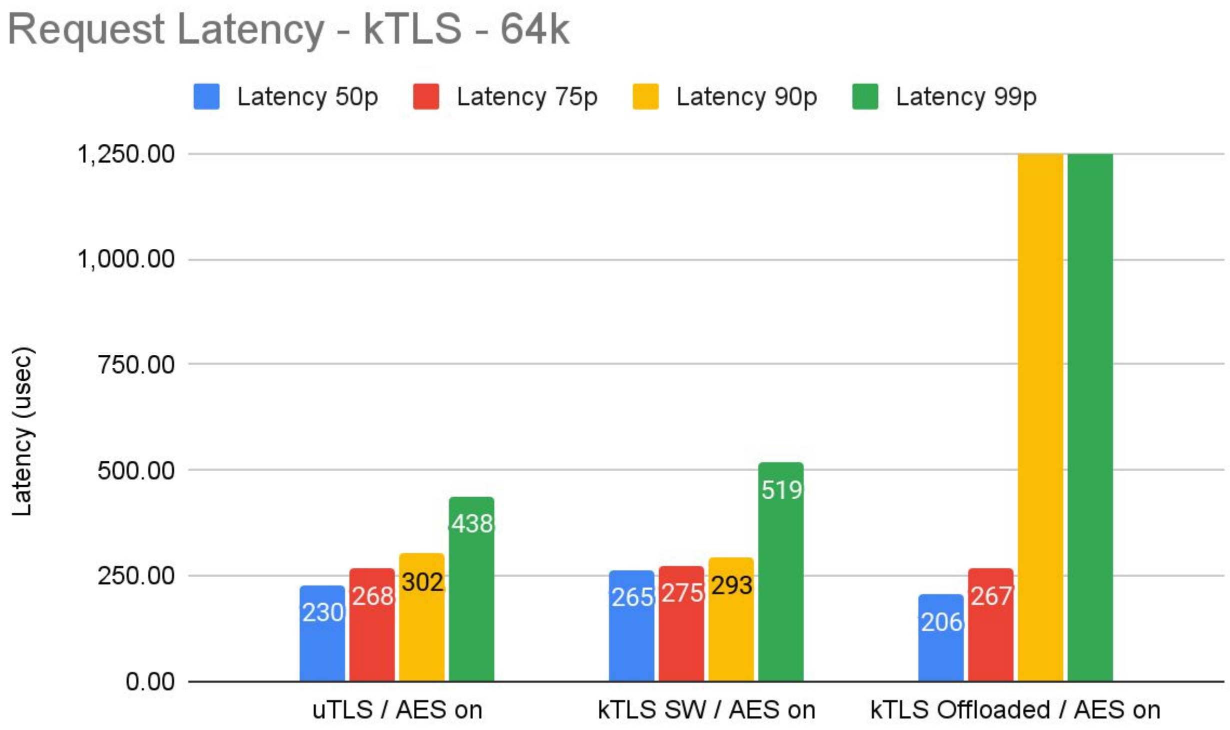


Latency tests

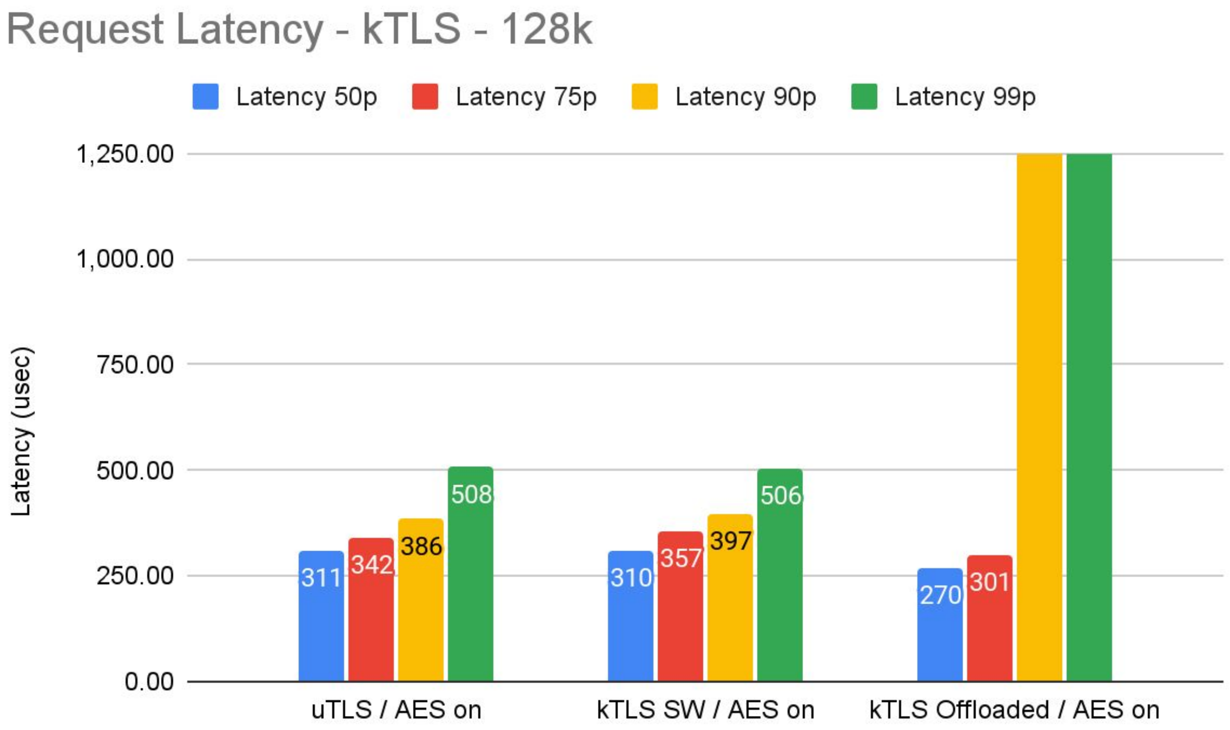
Request Latency Testing (Lower Better)



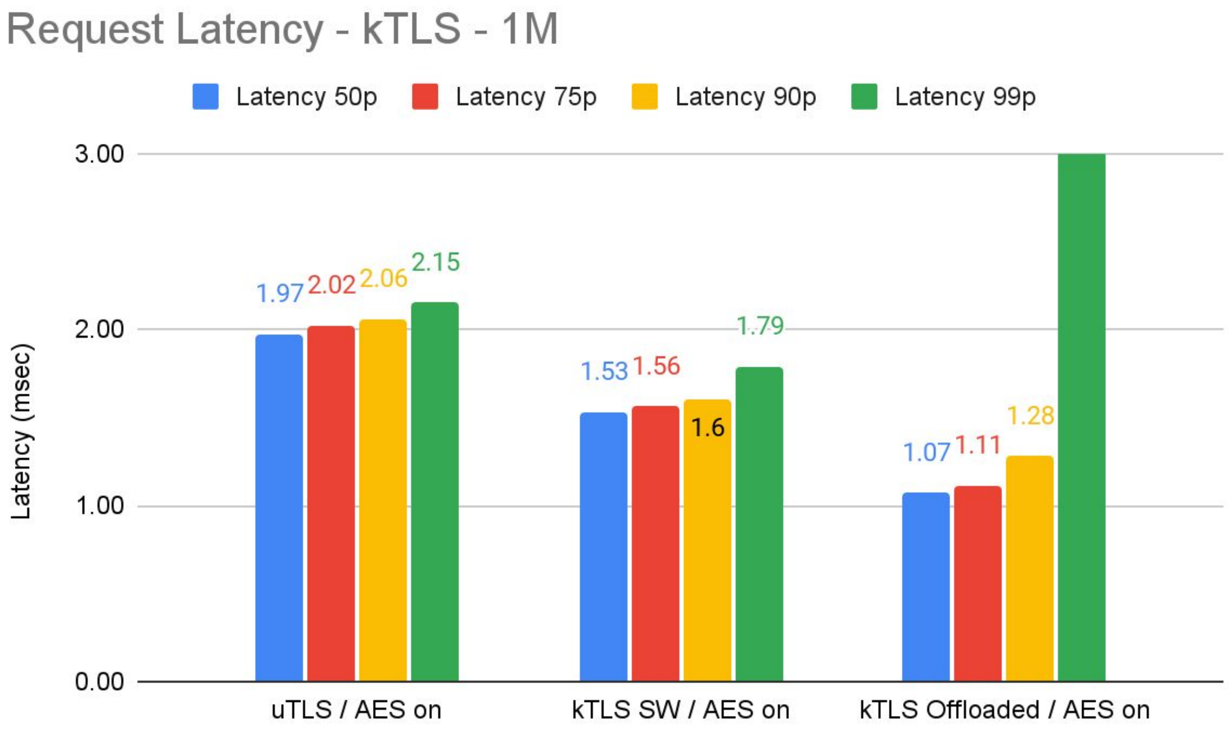
Request Latency Testing (Lower Better)



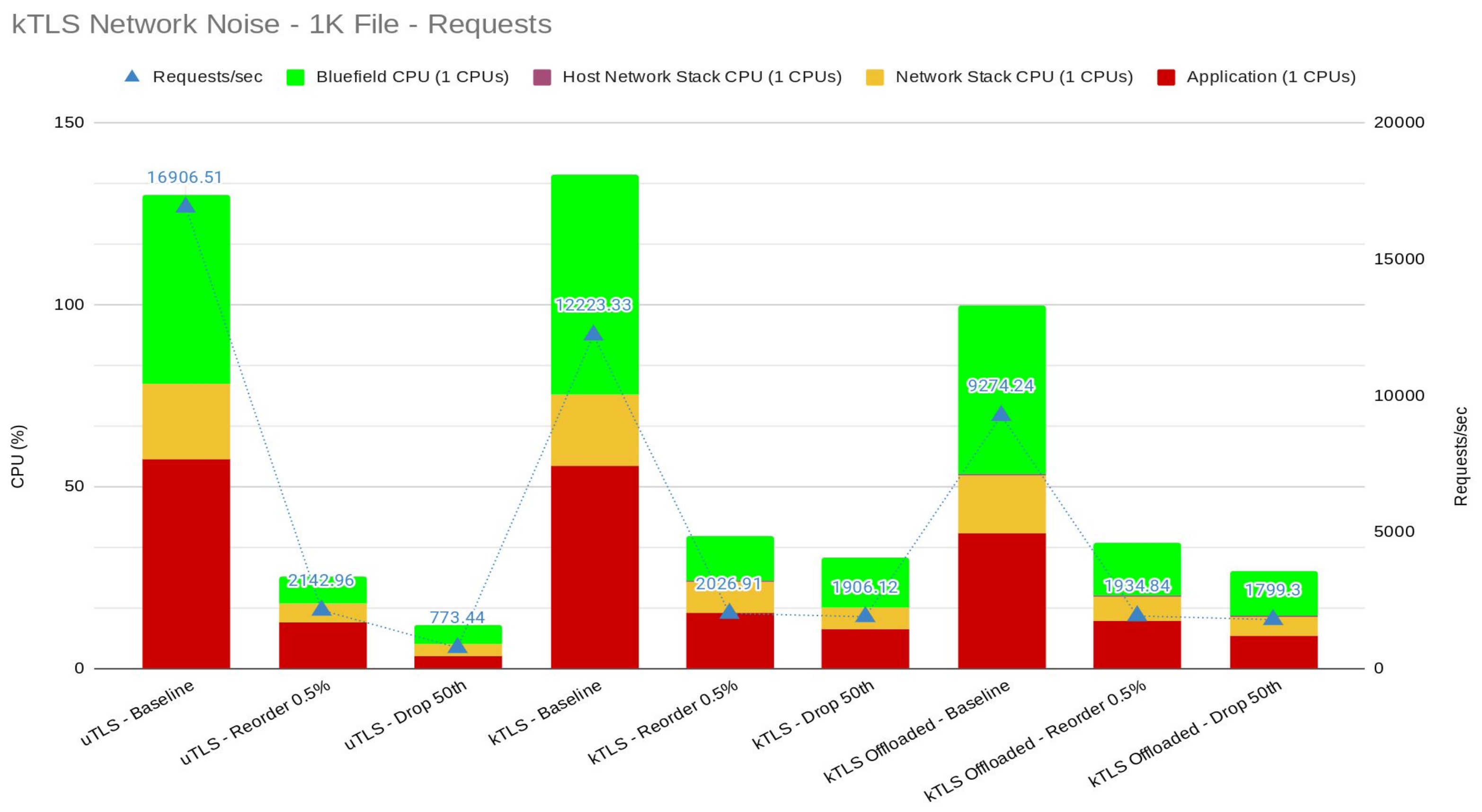
Request Latency Testing (Lower Better)



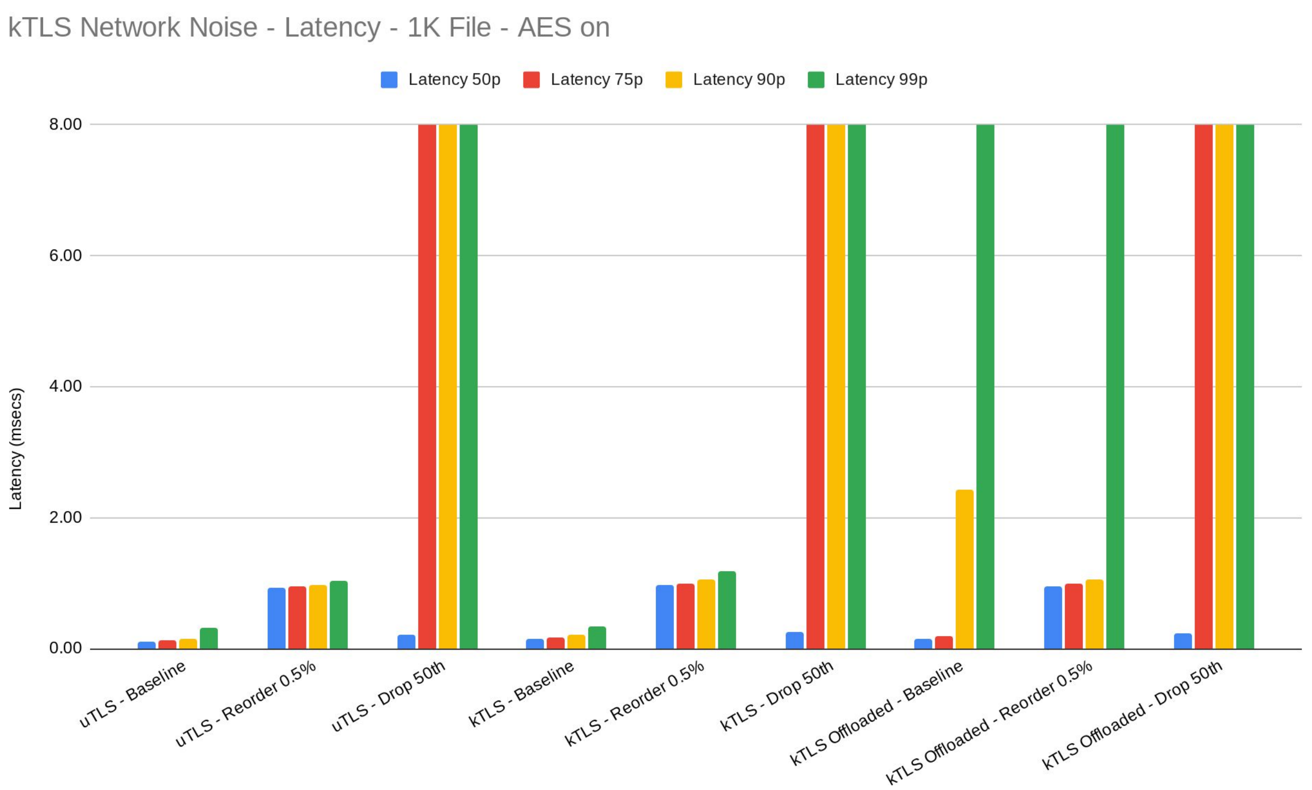
Request Latency Testing (Lower Better)

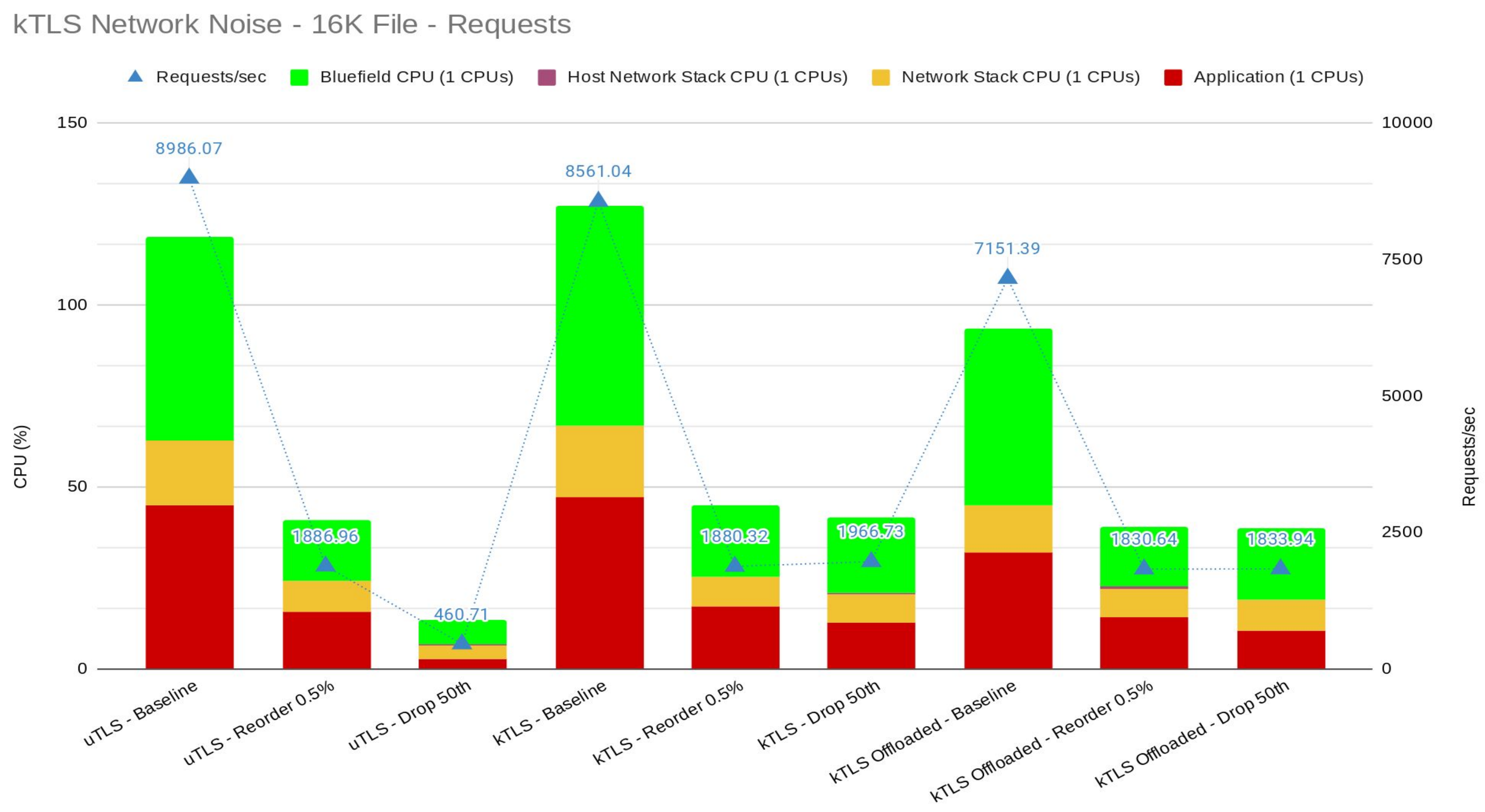


Network Noise

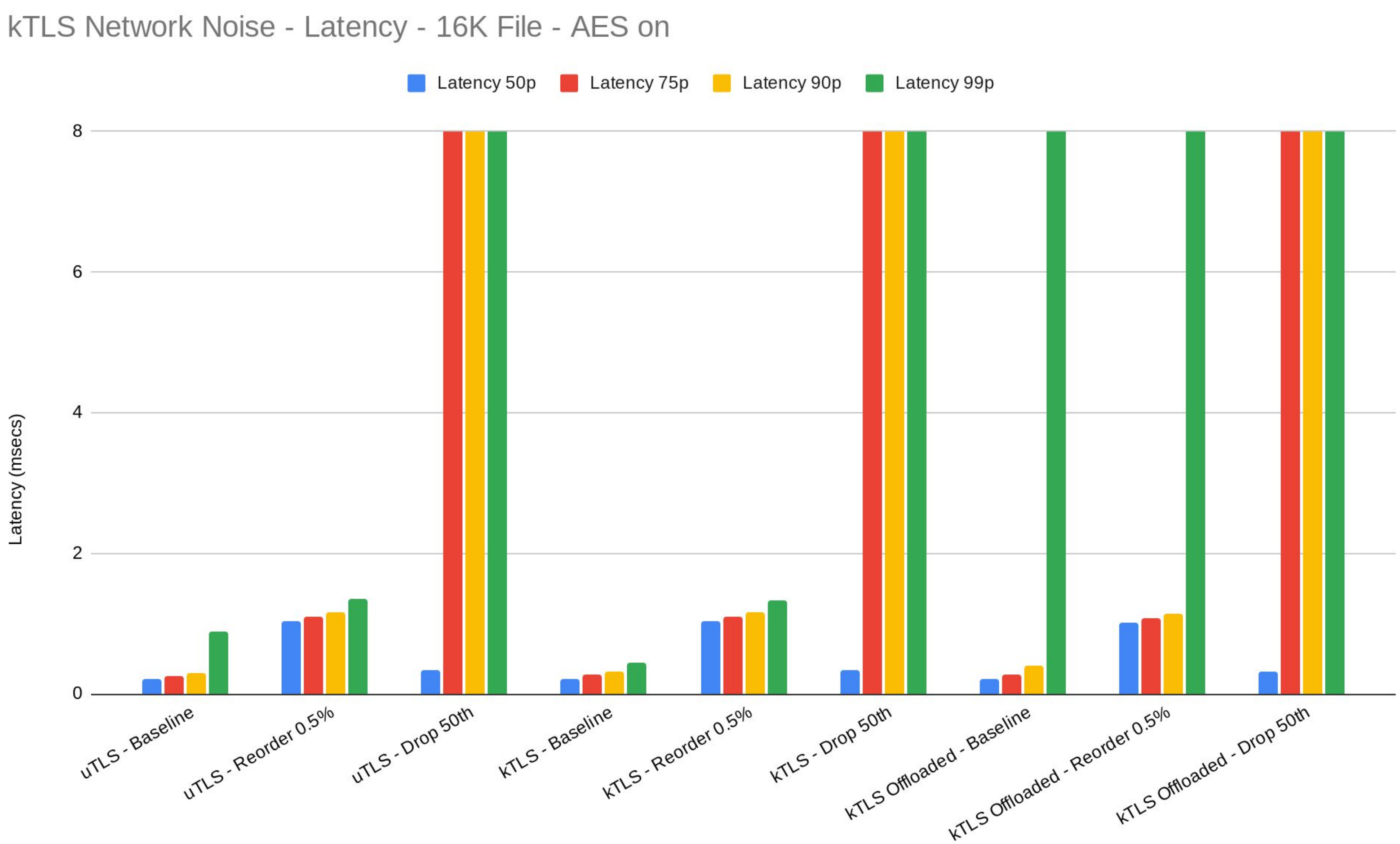


kTLS Network Noise - 1K File - Requests
Requests/sec
Bluefield CPU (1 CPUs)
Host Network Stack CPU (1 CPUs)
Network Stack CPU (1 CPUs)
Application (1 CPUs)
CPU (%)
Requests/sec
150
100
50
0
20000
15000
10000
5000
0
16906.51
2142.96
773.44
12223.33
2026.91
1906.12
9274.24
1934.84
1799.3
uTLS - Baseline
uTLS - Reorder 0.5%
uTLS - Drop 50th
kTLS - Baseline
kTLS - Reorder 0.5%
kTLS - Drop 50th
kTLS Offloaded - Baseline
kTLS Offloaded - Reorder 0.5%
kTLS Offloaded - Drop 50th

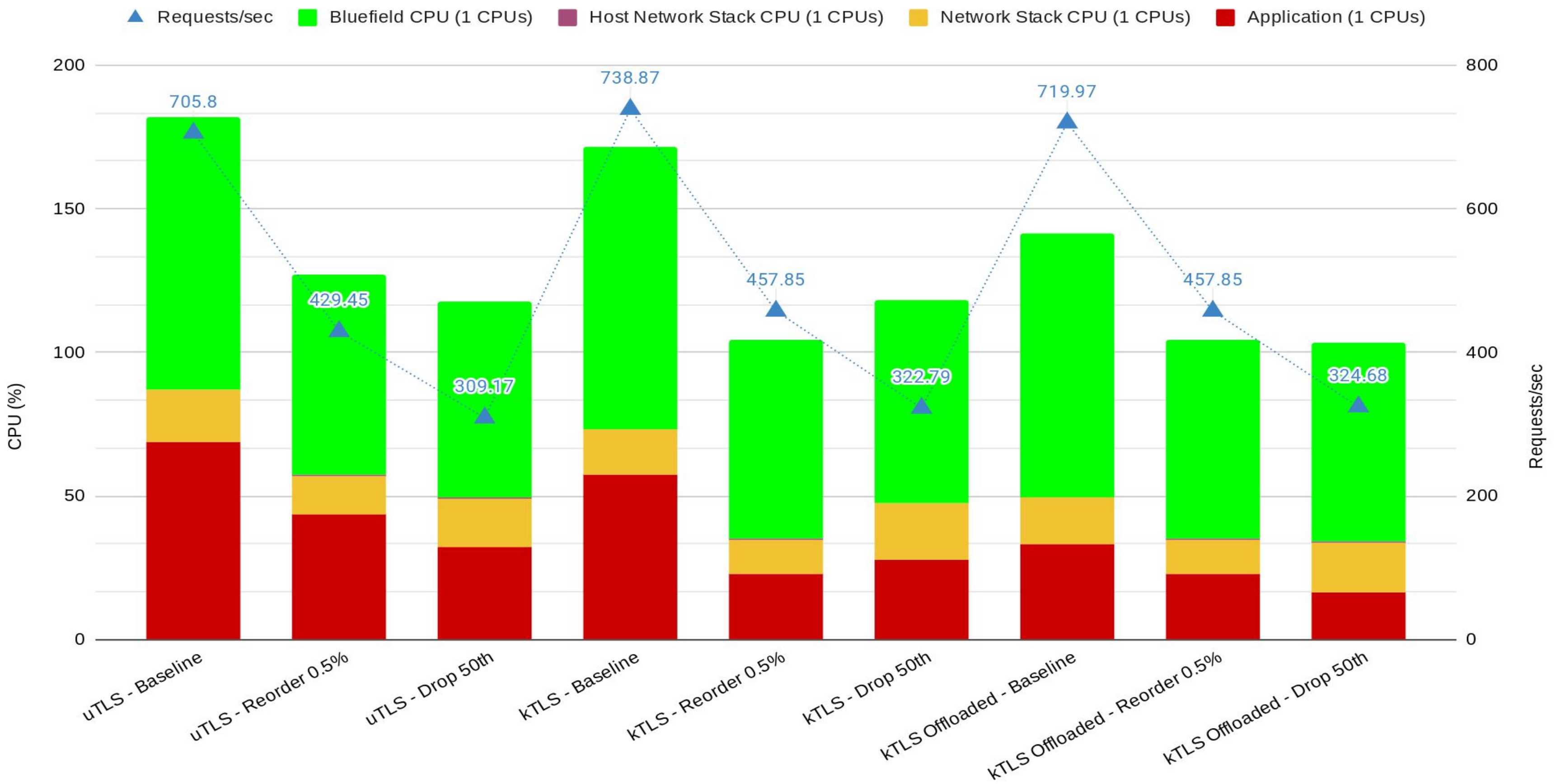

kTLS Network Noise - Latency - 1K File - AES on
Latency 50p
Latency 75p
Latency 90p
Latency 99p
Latency (msecs)
8.00
6.00
4.00
2.00
0.00
uTLS - Baseline
uTLS - Reorder 0.5%
uTLS - Drop 50th
kTLS - Baseline
kTLS - Reorder 0.5%
kTLS - Drop 50th
kTLS Offloaded - Baseline
kTLS Offloaded - Reorder 0.5%
kTLS Offloaded - Drop 50th



kTLS Network Noise - 16K File - Requests
Requests/sec
Bluefield CPU (1 CPUs)
Host Network Stack CPU (1 CPUs)
Network Stack CPU (1 CPUs)
Application (1 CPUs)
CPU (%)
Requests/sec
150
100
50
0
10000
7500
5000
2500
0
8986.07
1886.96
460.71
8561.04
1880.32
1966.73
7151.39
1830.64
1833.94
uTLS - Baseline
uTLS - Reorder 0.5%
uTLS - Drop 50th
kTLS - Baseline
kTLS - Reorder 0.5%
kTLS - Drop 50th
kTLS Offloaded - Baseline
kTLS Offloaded - Reorder 0.5%
kTLS Offloaded - Drop 50th

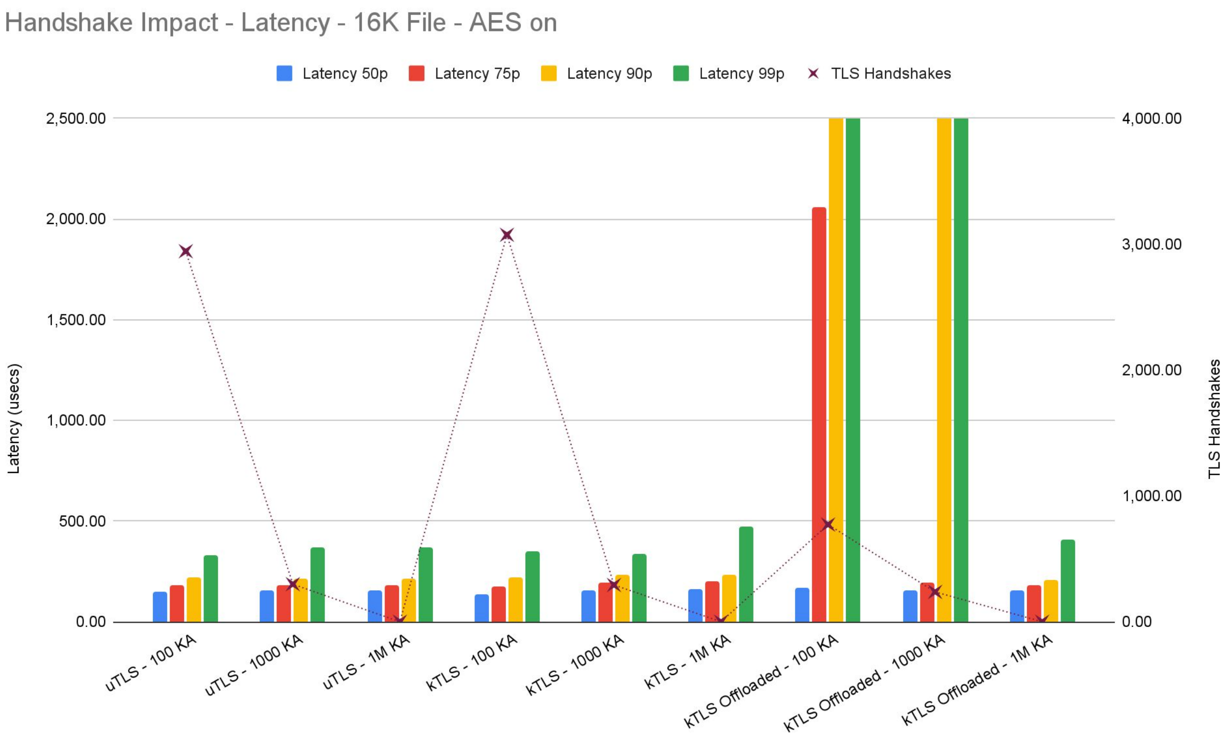

kTLS Network Noise - Latency - 16K File - AES on
Latency 50p
Latency 75p
Latency 90p
Latency 99p
Latency (msecs)
8
6
4
2
0
uTLS - Baseline
uTLS - Reorder 0.5%
uTLS - Drop 50th
kTLS - Baseline
kTLS - Reorder 0.5%
kTLS - Drop 50th
kTLS Offloaded - Baseline
kTLS Offloaded - Reorder 0.5%
kTLS Offloaded - Drop 50th

kTLS Network Noise - 1M File - Requests

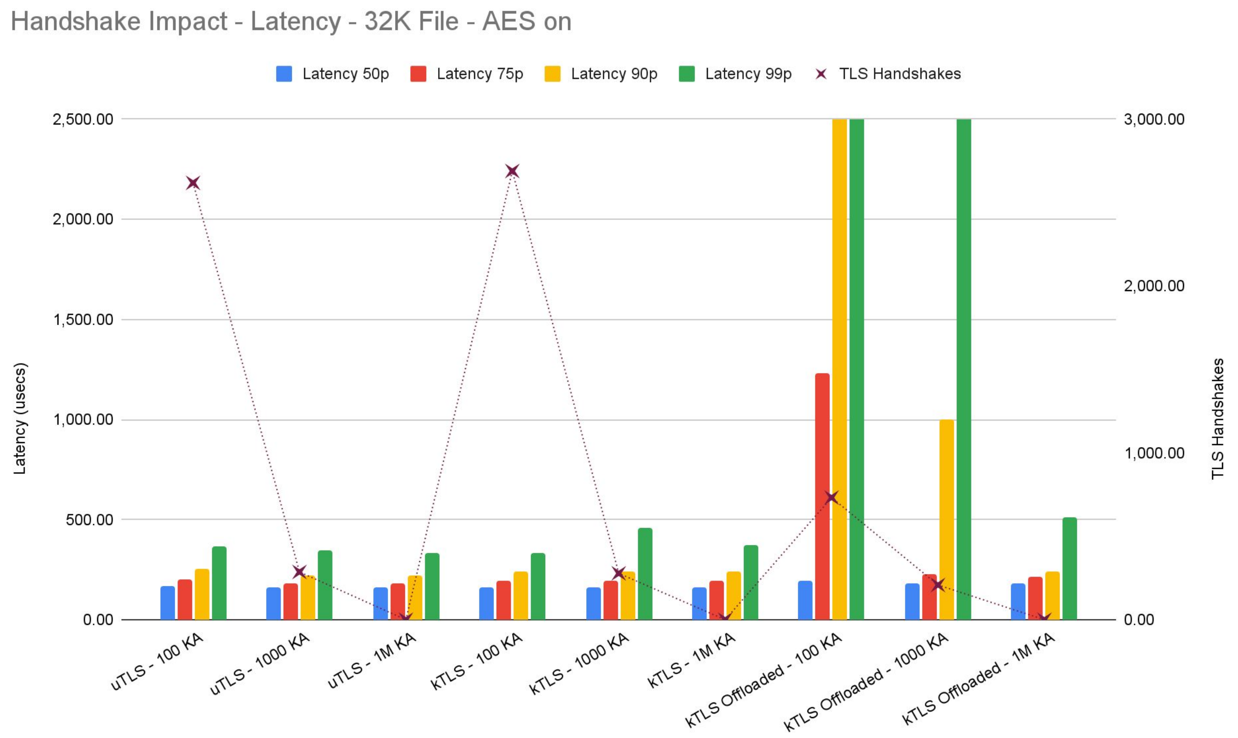


Handshake Latency tests

Handshake Impact: Latency



Handshake Impact: Latency



Handshake Impact: Latency