Unlocking Security to the Board: An Evaluation of SmartNIC-driven TLS Acceleration with kTLS

Felipe A. S. Novais
Department of Computer Science
UFSCar
São Carlos, Brazil
felipenovais@estudante.ufscar.br

Fábio L. Verdi
Department of Computer Science
UFSCar
São Carlos, Brazil
verdi@ufscar.br

***Abstract*—This work delves into the vital role of TLS (Transport Layer Security) in securing web applications today. We explore kTLS (Kernel TLS) offloading as a possible solution to alleviate resource strain such as CPU time, power consumption, and network speed. By shifting cryptographic tasks closer to the CPU in software offloading or away from the main CPU in hardware offloading, kTLS can improve resource efficiency. Our experimental studies assess various offloading strategies, including software-based kTLS that bring it closer to the Kernel and cutting-edge hardware-accelerated modes such as TOE and coprocessor configurations using the Chelsio T6 SmartNIC. We highlight the immense potential of kTLS and network adapters in reshaping performance and efficiency dynamics for some network environments, considering each approach's benefits and potential drawbacks.**

***Index Terms*—Offloading, acceleration, cryptography, SmartNIC, carbon-awareness.**

I. Introduction

Offloading represents an approach for mitigating CPU load by harnessing external processing units. Within computer networks, offloading stands as a pivotal technique aimed at alleviating the computational burden on server processors, thereby enabling the primary CPU to dedicate more resources to applications [1]. In the contemporary landscape, characterized by the burgeoning adoption of microservices architecture, offloading has assumed a prominent role in enhancing the resource efficiency of these distributed applications [2].

In addition to offloading, TLS is a protocol that secures communications and is essential to modern Web applications. With the evolution of the Web, TLS has become crucial, a mandatory requirement in most HTTP/2 implementations, and part of the proposal of the HTTP/3 protocol (HTTP over QUIC) [3]. TLS ensures data privacy, integrity, and security in online transactions, making it fundamental to the functioning of web applications [4].

kTLS is a software module that provides an interface for offloading the processing of the TLS protocol to specialized hardware or software [5]. The objective of kTLS is to ease the offloading of TLS by providing a standard interface for hardware acceleration of cryptographic functions [6]. This approach can significantly reduce the computational burden on server processors and enhance the resource efficiency of distributed applications, even in cloud environments [7]. In addition to hardware acceleration, kTLS also supports a software mode that brings TLS closer to the kernel, reducing context switches. By providing a kernel interface for TLS processing, kTLS reduces the need for user-space libraries to handle TLS, resulting in faster and more efficient processing of TLS connections. Furthermore, the software mode enables kTLS to fully utilize the available CPU resources, resulting in improved performance and reduced latency [8]. By supporting both software and hardware offloading, kTLS provides a flexible and efficient approach to offloading TLS, enabling applications to take advantage of the latest hardware technologies for enhanced performance and resource efficiency.

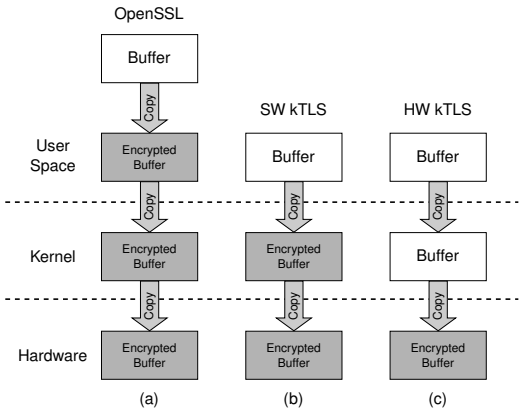

Fig. 1. Illustration of the data flow paths for kTLS (Kernel TLS) and its hardware (HW) and software (SW) modes in comparison to the typical user-space TLS libraries like GNUTLS or OpenSSL. The diagram showcases how kTLS brings TLS operations closer to the kernel, while hardware offloading leverages specialized hardware (SmartNIC) for enhanced cryptographic processing and optimized TLS traffic handling.

Figure 1 shows the datapath of three distinct modes using kTLS: the usual mode without kTLS (Fig. 1 (a)); The datapath with kTLS in software mode (Fig. 1 (b)); and datapath with kTLS in hardware mode (Fig. 1 (c)).

Our study aims to analyze the effectiveness of different offloading scenarios using kTLS, with the SmartNIC Chelsio T6 as a specialized hardware. We assess these scenarios in bare-metal and containerized environments, with the

containerized setup resembling a cloud-native microservice architecture. We also evaluate scenarios with and without dedicated hardware under various offloading modes, including No Offloading (user-space library TLS), Software kTLS, Inline mode hardware kTLS utilizing the TOE (TCP Offloading Engine), and Coprocessor mode hardware kTLS (similar to AES-NI [9] but in the SmartNIC). Our findings underscore the potential benefits and opportunities kTLS presents when paired with network adapters. We observe positive tendencies on resource effectiveness for software and hardware offloading, especially in bare-metal scenarios, where the hardware acceleration shows significant performance improvements. However, in a cloud-native architecture (containerized environments), we could not verify a significant difference between hardware and software offloading.

Our experiments showed that implementing kTLS requires a Linux kernel version of at least 4.13, a supporting cryptographic backend, and code compilation against this backend. This suggests that the implementation process is not straightforward.

To run all the evaluations, we carefully prepared a set of artifacts to fine-tune the experiments and ease the reproducibility. The methodology adopted for conducting the experiments is of paramount importance for the accuracy and analysis of the results. As such, we consider that such methodology, materialized in scripts and tools, all publicly available[1], is also a valuable contribution that may be used by others to evaluate SmartNICs from different vendors.

The rest of the paper is organized as follows: Section II presents the related work. Section III briefly explains the main concepts involved in this research. Section IV explains the methodology used to set up the experiments and evaluate their results, describes experiments done during the research, and discusses its results. Section V discusses our concluding remarks and future opportunities.

II. Related Work

In this section, we present an overview of related works in the field of offloading and the use of SmartNICs, particularly focusing on their relevance to our research on TLS offloading.

The paper featured on [10] discusses the usage of FPGA-based SmartNICs to accelerate general-purpose cloud applications and their capabilities for offloading hypervisor network infrastructure. NICA [10] is presented as a hardware and software framework designed to accelerate the data plane of applications on F-NICs[2] in multi-tenant systems. A new programming abstraction called ikernel is introduced to allow application control of F-NIC computations. NICA is implemented on Mellanox F-NICs and integrated with the KVM hypervisor, demonstrating significant acceleration of real-world applications in virtualized and bare-metal environments, with minor code modifications. The throughput of an ikernel added to a memcached server reaches 40 Gbps and scales linearly to up to six independent virtual machines.

The work in [11] explores the potential of emerging SmartNICs with advanced processing capabilities like multicore processors, onboard DRAM memory, programmable DMA engines, and accelerators for generic datacenter server tasks. However, efficiently harnessing SmartNICs for maximum offloading benefits in distributed applications remains uncertain. To address this, the authors evaluate four commercial SmartNICs, examining offloading performance in terms of traffic control, computational power, onboard memory, and host communication. Using these findings, they introduce iPipe [11], an actor-based framework for offloading distributed applications on SmartNICs. At its core, iPipe employs a hybrid scheduler combining FCFS-based processor sharing and DRR, effectively handling tasks with varying running costs and optimizing NIC processing. With iPipe, the authors develop real-time data analysis engines, distributed transaction systems, and replicated key-value stores, evaluating their performance on commercial SmartNICs. Results indicate significant savings in Intel core usage and reduced application latencies, particularly when processing at 10/25 Gbps application bandwidths.

The work in [12] addresses the issue of CPU overhead caused by symmetric encryption and TLS authentication in data center servers handling encrypted traffic. The article suggests offloading TLS symmetric cryptographic processing to network devices, specifically using a kernel TLS module (kTLS) that leverages inline TLS acceleration. This allows the network device to process TLS connections and decrypt transmitted packets before reaching the kernel stack. The kTLS module's functions, requirements, and performance benefits are detailed. This offloading approach is flexible and can significantly improve the performance of some environments.

Certainly, the authors in [13] uncovered intriguing insights; for instance, employing external hardware to alleviate CPU load did not invariably yield performance benefits. They observed a minor performance dip when varying message sizes for tasks like hashing. While the notion that SNICs[3] might consistently enhance cryptography scenarios is intuitive, it is prudent to acknowledge that such uniformity may not apply across all scenarios. Rather, a complex interplay of factors must be considered when devising an offloading strategy. It is important to acknowledge the research conducted in [13] for its valuable findings and contribution to the field. However, it is also worth noting that the analysis did not include an evaluation of CPU time and power consumption, which are important factors in assessing the efficiency of offloading techniques. Despite this limitation, the research remains an insightful and informative study that has advanced our understanding of the subject.

Despite related works effectively presenting the advantages of offloading to SNICs in general, few studies delve into the application of kTLS as a TLS offloading method. This research centers on evaluating the kTLS kernel module's ability to abstract cryptographic functions and enable offloading through both software and hardware channels. The primary objective

[1] https://github.com/dcomp-leris/tls-offloading-research.

[2] Shortened term for SmartNICs utilizing FPGA technology.

[3] Commonly employed acronym for SmartNIC.

of this paper is to showcase the module's advantages within the context of web applications.

Upon a comprehensive review of the academic and commercial landscape documented in the literature, it becomes evident that significant attention has been directed toward TLS hardware offloading. However, few references do specifically address these scenarios incorporating kTLS. Consequently, this paper endeavors to assess cryptographic offloading on NICs using kTLS, filling a noteworthy gap in the current body of research.

III. Background Concepts

This section presents an overview of terms in the paper, focusing on Offloading and different modes that the SmartNIC used in the experiments provides.

A. Offloading

Offloading is the technique that transfers processing tasks from one component to another to reduce the workload of a system and increase its efficiency [14]. This technique is commonly used in networking systems, where offloading specific tasks to specialized devices (such as network cards, for example) can significantly improve system performance and reduce overhead on the main CPU. Offloading is applied to compression/decompression, encryption, video decoding, and other tasks [15].

B. kTLS - Kernel Transport Layer Security

The kTLS kernel module is a security extension that enhances the TCP transport protocol on Unix-like systems that support it by adding encryption and authentication features. With kTLS, it is possible to implement TLS encryption in the operating system kernel, which can offer superior performance compared to user-space software solutions. Moreover, kTLS can be utilized for TLS offloading on network cards that support it, enabling encryption and authentication to be carried out directly by the network card [5]. This can help reduce CPU load and enhance the overall performance of the system.

C. Chelsio's Coprocessor and Inline offloading

For the experiments in this paper, we used the SmartNIC Chelsio T62100-LP-CR. There are other SmartNICs on the market capable of performing TLS acceleration via kTLS, such as NVIDIA's Bluefield line [16]. We chose the Chelsio T6 line due to its cost-effectiveness and the availability of documentation that can be found on the Internet. It is important to contextualize that the Chelsio T6 has two offloading modes: Inline and Coprocessor modes. In the Inline mode, the offloading happens in the context of Chelsio's proprietary TOE of the SmartNIC, and it occurs in parallel to other offloading operations like Direct Data Placement (DDP), Direct Data Sourcing (DDS) and checksum calculation using CRC algorithm. The Coprocessor mode is analog to AES-NI but in the SmartNIC. In contrast, the Inline mode only supports offloading crypto functions related to TLS/DTLS. In the Coprocessor Mode, the SmartNIC can run crypto functions like data encryption at rest, SMB, IPSec, and hashing algorithms.

IV. Evaluation

A. Setup

During the experiments, two machines were used, one acting as a server and the other as a client. The server has a 1 dual port 100GbE Chelsio T62100-LP-CR installed and connected back to back to the client through a splitter cable, splitting one of the 100G ports into 4x10GbE ports. We used the Chelsio Unified Wire v3.18.0.0 to install and update the firmware, drivers, and utilities required to use the offloading capabilities of the T6 and to make sure it all worked as it should. We used the specific kernel version recommended by the vendor as specified in Table I. All other configurations used can also be seen in Table I.

We used separate hosts for the client and server, as this approach minimizes resource interference, simulates real network conditions, and provides better control over variables, ensuring accurate results. Using a single host simplifies testing but may introduce uncertainties and reduce precision.

It is essential to keep in mind that, aside from the client and server having NICs with 10G and 25G capacities, respectively, these capabilities can be influenced by various other factors, including the performance of the computers, the resource load on the hosts, and even physical issues such as cabling [17]. No rate limit was imposed on the programs used for generating traffic, and we made efforts to isolate these machines from physical interferences so the traffic generators could use as much bandwidth as they could from the network connection. We can see how resources can affect the throughput as we observe that in the bare-metal setup, the maximum throughput achieved during this test was 9.36 Gbps, indicating robust performance. In contrast, the containerized experiments yielded a lower maximum throughput of 588 Mbps, likely due to resource constraints on the client and server hosts.

OpenSSL version v3.0 or higher is required for the kTLS module to be supported, so we used OpenSSL 3.0.7 compiled with the `'enable-ktls'` flag. NGINX 1.22 was chosen as the web server, effectively performing the roles of a load balancer for other applications and delivering binary files of varying sizes to capture relevant traffic metrics.

The evaluations were split into two big scenarios: *Bare-metal* and *containerized*. To build the microservice environment for the containerized evaluations, we opted for an NGINX Docker [18] image with kTLS configuration and a static blob file. This setup allowed us to conduct tests that closely paralleled the bare-metal scenarios, ensuring consistency and comparability in our evaluations.

To conduct the experiments and collect essential data samples, we employed a combination of tools, including `cURL` [19] and `wrk` [20], for sending requests to NGINX. The orchestration of these requests was facilitated through Bash and Python scripts. cURL was used just to generate traffic while collecting CPU and Power data, while wrk supplied network statistics, enabling us to calculate the average throughput and latency of the conducted tests.

Component	Server	Client
Processor	i7-7700K	Xeon E5-2420
RAM Memory	32GB (4x8 DDR4 2133mhz)	32GB (2x16 DDR3 1600mhz)
Motherboard	Gigabyte Z170XP-SLI	Dell PowerEdge R420
Operating System	RHEL 9.2 (5.14.0-284.11.1)	Ubuntu 22.04.2 (5.15.0-72)
Network Adapter	Chelsio T62100-LP-CR	Intel 82599ES

TABLE I
CLIENT AND SERVER SPECIFICATIONS

Collecting the metrics related to the CPU time was done using `CollectD` [21] as it provides a reliable way to get the CPU time per process with low overhead of the collecting agent. To get more precise CPU use by the target processes, we used the CPU affinity method, that in Linux can be achieved using isolcpus and taskset to bind the processes related to the tests to a single CPU core and avoid resource sharing with other processes.

The Intel RAPL interface facilitates the reporting of power consumption metrics. To streamline the collection of sensor metrics, we employed Scaphandre Prometheus Exporter. Additionally, we utilized Prometheus and Grafana to visualize and interact with the data through a web API.

B. Experiments

During the experiments, we conducted tests for each of the four operational modes: No Offloading, Software kTLS, Coprocessor, and Inline Mode.

- **No Offloading:** This method occurs in user-space, where traditional encryption and decryption tasks are processed mainly by the CPU;
- **Software kTLS:** This mode uses the kTLS module to offload encryption tasks to the kernel, reducing CPU workload. It is a software-based approach that requires compatible applications and kernel support;
- **Coprocessor:** The Chelsio's T6 Coprocessor mode allows for hardware offloading of TLS encryption and decryption tasks to a specialized coprocessor, similar in concept to Intel's AES-NI instructions for accelerating encryption and decryption operations;
- **Inline:** The Chelsio's T6 Inline mode utilizes a TOE to handle TLS tasks directly in the network stack. It offloads some encryption tasks of the TLS process to dedicated hardware.

Each offloading mode of Chelsio requires a different Linux module to be loaded. In the case of Inline mode, the TOE has to be loaded with a module called `t4_tom`. For the Coprocessor mode, the module `chcr` has to be loaded. At the time this paper was written, `t4_tom` was not able to be hot-loaded and unloaded during the system execution, so every time you needed to change back to another mode, you had to reboot the system to make sure the modules were unloaded.

The experiments take time to run, as they involve several complex procedures and measurements. To automate the system restarts and reduce human intervention during the tests, a state machine was implemented using SystemD [22]. This not only streamlined the testing process but also significantly augmented reproducibility. With this method, for every system restart, the server seamlessly alternated between experimental scenario modes: No Offloading, Software kTLS, Hardware kTLS (Inline), and Hardware kTLS (Coprocessor). In each iteration, the server interacted with the client to execute the tests and then automatically restarted to transition to the next step and activate the subsequent scenario mode. This utilization of SystemD ensured that the necessary kernel modules were loaded accurately, contributing to precise and replicable experiment outcomes.

During our testing and analysis, we tracked the main metrics that may affect the decision about using offloading or not: CPU time, Power Consumption, Latency, and Throughput.

- **CPU Time:** Measured by Collectd, indicates the time the CPU spends in various states, such as executing user code, executing system code, waiting for I/O operations, and being idle. It provides insights into CPU utilization and performance. In our case, this value was accumulated over the time a file was downloaded, and the raw data, which is in ns, was converted to ms to provide better visualization.
- **Power Consumption:** Metrics collected by Scaphandre focus on electrical power usage in technology services. It measures the energy consumed by servers, storage equipment, and network infrastructure. In our case, this value was accumulated over the time a file was downloaded, and the raw data was in microwatts (μS) and then converted to watts (W).
- **Latency:** Reported by the wrk benchmarking tool, represents the time it takes for a system to respond to an HTTP request. It provides insights into the responsiveness and performance of a web server or application, with lower latency indicating quicker responses. A custom script was used to export data to CSV, where the average latency that wrk calculates could be obtained. Raw data was provided in μs and, for better visualization, converted to ms.
- **Throughput:** Measured by wrk, quantifies the volume of requests a web server can handle within a specified time frame. It reflects the server's capacity to process incoming requests efficiently. Higher throughput values indicate better server performance under load. A custom script was used to export data to CSV and to calculate the throughput (bytes/duration), then the value was converted to MB/s.

While the tests were running, CollectD, Scaphandre, and Prometheus ran in the background, monitoring resource usage and recording it in a time series database. To process the time series data and analyze its values, Grafana was used to visualize Scaphandre data exported to Prometheus, and for RDD, the database format used by CollectD could be visualized by RDDTool [23]. To further analyze the data and generate graphics, they were exported to CSV and processed through Python

using libraries like Pandas [24] and Numpy [25]. For plotting this data into a visualization[4], libraries like Matplotlib [27], and Seaborn [28] were used, and to enhance their usability, a Jupyter [29] environment was employed.

Python, in conjunction with Bash Scripting, played a pivotal role in automating various processes throughout the experiment. Python was employed for orchestrating test sequences, performing essential tasks such as loading the requisite kernel modules for each test scenario, regulating the volume of requests directed to the server, facilitating data retrieval via the Python Requests library, and managing remote machine reboots to transition between successive test scenarios, among other functions.

The Podman [30] was used as the container management tool for the containerized tests. A Debian image with OpenSSL and NGINX compiled with kTLS flags was made. A bash script was responsible for provisioning several containers, and in front of the machines, an NGINX Load Balancer was running in the bare-metal machine in Round Robin mode. This way, each connection would cycle by each application in each container every time the server receives a request.

To keep track of the experiments and grant ease of reproducibility, a series of scripts was used to automate the pipeline of the experiments. One of this automation was reporting the partial result of the runs in instant messaging apps like Telegram [31] and Discord [32]. In this way, if there was a malfunction during the experiments, we would know and have external logs of these occurrences.

C. Results

In accordance with the experimental configuration, we conducted a battery of tests to systematically assess each scenario, distinguishing between bare-metal and containerized setups.

CPU time and power consumption were acquired as accumulated values during the experiments. Scaphandre and CollectD continuously collected data while cURL generated network traffic by downloading a large file. These cumulative values were instrumental in understanding resource utilization and power efficiency. In contrast, throughput and latency metrics were sourced from statistics provided by the wrk tool. wrk operated within a defined time frame, meticulously tracking the volume of requests completed within that timeframe while varying the number of concurrent connections. This approach allowed us to precisely assess download speeds and response times under different scenarios, providing invaluable insights into system performance.

1) ***Bare-metal Experiments****:* In this section, we delve into the outcomes of our experiments conducted in a bare-metal environment. These experiments provide essential insights into the performance of the system with a focus on CPU time, throughput, latency, and power consumption. These tests were done using an NGINX server running directly on the bare-metal server mentioned in Table I.

[4]IBM's accessibility palette for visually impaired people was used to improve contrast for people that suffer from color blindness [26].

CPU - The scenario used was an NGINX serving a 30GB file that gets downloaded a hundred times to get the CPU time. The exact procedure will run in every test scenario: without kTLS, with software kTLS, with hardware kTLS using Chelsio's Inline mode, and with hardware kTLS using Chelsio's Coprocessor mode.

The results showed in Figure 2 for accumulated CPU time were reflected on the CPU time share used during a file download. It was possible to observe that the CPU time was progressive, varying from a scenario without offloading to a scenario using offloading (SW and HW). Without offloading, the expected behavior was the CPU being extensively used when compared to Software kTLS Mode. Software kTLS Mode performed better than No Offloading, and it can be explained because it reduces the number of kernel context switches as the operations to handle TLS are closer to the system kernel (as we see in Figure 1). Finally, the hardware modes (Coprocessor and Inline) were expected to perform better as they use an external processing unit (Chelsio's T6 in our case) to handle traffic encryption operations. **Takeaway:** *In terms of CPU time, hardware offloading, especially in Coprocessor Mode, proved to be the most efficient option.*

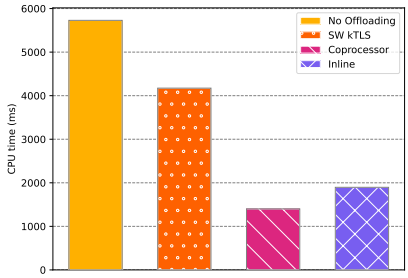

Fig. 2. CPU time (converted from ns to ms) analysis in bare-metal scenarios (without containers).

Throughtput - We analyzed the throughput of parallel connections to determine the potential benefits and drawbacks of utilizing kTLS offloading. Our findings indicated that in certain scenarios, there was a disparity in performance compared to other modes.

As shown in Figure 3, when the server managed only one connection, the performance of hardware offloading was worse than the other modes. The Coprocessor mode obtained only ≈1.7Gbps. However, as the number of parallel connections increased, leading to higher CPU time demands, the implementation of kTLS offloading emerged as an effective means to enhance the average throughput of TLS traffic. It is worth noting that despite the Inline mode utilizing more CPU than the Coprocessor mode, the Inline mode was still able to deliver better results for throughput (9.36Gbps). This is due, in part, to its dual functionality, which includes both TLS offloading and essential TCP offloading tasks, such as checksum verification (CRC offloading), as described in III. **Takeaway:** *Hardware offloading delivered superior throughput performance when dealing with multiple parallel connections.*

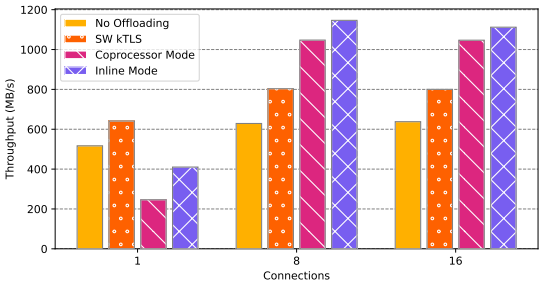

Fig. 3. Evaluation of download throughput in bare-metal setups (without containers) to examine how different offloading modes influence download speeds in different environments.

Latency - Response time was impacted by the different offloading modes we tested. As can be observed in Figure 4, the number of connections influenced the results. In scenarios with a single connection, the No Offloading scenario performed best as the CPU was less active than in other scenarios where the server had to handle multiple connections. An interesting observation is that for a single connection, the Inline Mode performed much worse than the others. This may be explained by the overhead needed to make the Inline Mode run, such as the TOE, which competed for resources more than in the other modes. However, the Inline Mode made a comeback when we observed scenarios with more parallel connections. In such cases, the server achieved better latency in the hardware offloading scenarios, as expected, due to the shorter data path in these modes, resulting in reduced latency. **Takeaway:** *Hardware offloading, particularly in Coprocessor Mode, delivered superior performance for latency when dealing with multiple parallel connections.*

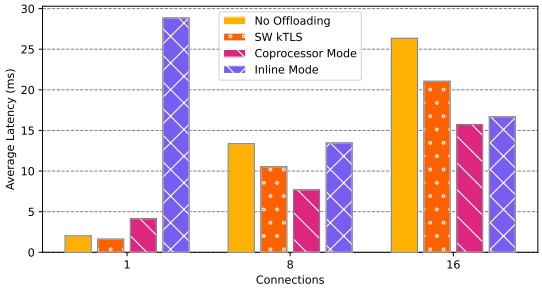

Fig. 4. Latency during concurrent downloads in bare-metal environments (without containers).

Power Consumption - The results presented in Figure 5 regarding power consumption can be elucidated by examining the interplay between CPU time and throughput. Firstly, in scenarios characterized by higher throughput, we anticipated faster processing, thereby reducing CPU time over time as data transmission occurred quicker, necessitating less CPU time. Indeed, we observed that the data exhibited a notable resemblance to the CPU time metric, reinforcing this correlation. However, a notable anomaly surfaced when considering the Inline Mode, which, despite its superior throughput and efficient CPU utilization, demonstrated comparatively poorer performance in terms of power consumption. This unexpected behavior can possibly be attributed to the substantial overhead associated with the complete execution of the TOE required by the Inline Mode, rendering it considerably more CPU-intensive than the other modes [33]. Consequently, although Inline Mode may enhance speed, it appears to compromise energy efficiency in the process.

In contrast, the Coprocessor Mode presented an intriguing contrast. It exhibited the potential for significant power consumption gains. It offloads critical cryptographic operations to specialized hardware, reducing the CPU's load. This reduction in CPU utilization translates directly into lower power consumption, suggesting that the Coprocessor Mode offers a promising avenue for optimizing energy efficiency in scenarios where cryptographic offloading is critical.

In the context of the United States Data Center Energy Usage Report [34], our study aligns with the industry's aim to optimize energy efficiency in data centers. As data centers seek to reduce electricity consumption while meeting growing digital service demands, our kTLS analysis highlights the need for technologies that balance performance and power efficiency. **Takeaway:** *Coprocessor Mode exhibits a small advantage in efficiency compared to the others, while Inline Mode, pays the trade-off of throughput with higher power consumption.*

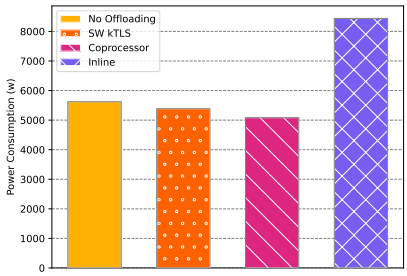

Fig. 5. Power consumption patterns, measured as accumulated values, in bare-metal test scenarios (without containers).

The Inline mode showed a worse performance compared to other test scenarios. It may be related to the Inline mode requiring the TOE, as it performs additional functions beyond cryptographic offloading.

*2) **Containerized Tests**:* When evaluating the system's performance in TLS offloading, it became clear that containerization is a must when working with microservices. This investigation aimed to determine whether SmartNIC offloading could alleviate the infrastructure overhead associated with running multiple containers. To achieve this, we used a script to orchestrate the number of containers running behind the load balancer (described in IV - A - Setup), allowing us to test scenarios for different numbers of containers progressively. By instantiating varying numbers of containers on the host machine, each running a lightweight web application tasked with serving data, we were able to assess the impact on the system's performance. It is essential to note that the coproces-

sor mode of Chelsio did transparently work for containerized scenarios, which could be validated by the driver utilities that count encryption operations done by the NIC. Unfortunately, Chelsio's Inline mode did not work for the containerized scenario. Therefore, this section will not present results related to this mode.

The results showed that the offloading scenarios exhibited superior performance when resource consumption scaled up, particularly with an increase in the number of containers, while the No Offloading approach performed well in situations where the server has vast resources to manage requests.

It is worth mentioning that the tests and the numbers shown in this paper were performed with default Docker settings to evaluate a standard scenario. However, performance enhancements are achievable through fine-tuning. For example, we reached a throughput of around 2.60 Gbps by configuring the container to utilize a `--net=host` type network with a mounted volume for static files.

CPU - In the containerized scenario, as illustrated in Figure 6, a similar pattern to the bare-metal results emerged. Initially, with plenty of resources available for the server to handle requests and manage the container backend, Software kTLS showcased its potential. However, as the number of containers increased, it became increasingly apparent that Software kTLS presented a more favorable option. **Takeaway:** *Over time, as resources become shorter, Software kTLS may offer a competitive advantage compared to the alternatives.*

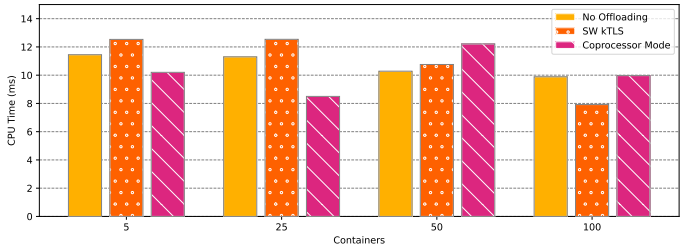

Fig. 6. Accumulation of CPU time (converted from ns to ms) within a containerized environment.

Throughput - Upon an examination of Figure 7, a clear trend emerges: as the number of concurrently executed containers tests run, Software kTLS consistently demonstrates slightly superior performance compared to the Coprocessor, and both performed better than no offloading. These findings highlight the effectiveness of Software kTLS, showcasing its competitive edge in throughput, especially in high-demand scenarios with containers. Interesting to note that the containerized scenario obtained a very low throughput (588Mbps) when compared with the bare-metal (9.36Gbps). **Takeaway:** *The results clearly indicate that offloading, be it hardware or Software kTLS, offers advantages in cloud-native setups when talking about throughput.*

Latency - As we observed in Figure 8, the increase in the number of containers showed slight differences in response time. The most significant difference was when comparing the single container scenario to the others. Notably, this impact on latency closely resembled the impact on throughput, where it was very similar between the scenarios with different container numbers except for the single container scenario. As the results from throughput, Software kTLS exhibited an advantage over the other modes, so it did for latency. In contrast, the Coprocessor mode was expected to perform better as it got better CPU time and power consumption results, however, Coprocessor mode performed worse than Software kTLS. **Takeaway:** *Software kTLS consistently exhibited lower latency than the other modes.*

Power Consumption - Based on the experiments conducted, we can observe in Figure 9 that while there was a slight improvement in performance from 25 to 50 containers, the benefits of hardware TLS offloading became more evident when we scaled up to 100 containers. It seems that in high resource usage scenarios, the offloading methods tend to slightly reduce power consumption compared to the No Offloading approach. This trend was particularly noticeable with 100 containers, highlighting the importance of utilizing hardware offloading techniques when dealing with large-scale microservices. **Takeaway:** *Our results indicate that over time, as resource usage increases (number of containers), offloading reduces power consumption, particularly in resource-intensive situations.*

D. Observations

1) Legacy Applications: - Enabling kTLS support in an application typically entails compiling it with a specific flag-enabled version of OpenSSL (or other cryptographic backends that supports it), like OpenSSL 3.0.0, which allows the application to utilize kTLS for improved performance in handling TLS traffic. However, a notable challenge arises when dealing with applications for which the source code is inaccessible, as you may lack the ability to modify or recompile them to enable kTLS. This limitation can impede the integration of kTLS into closed-source or proprietary applications, hindering the realization of its performance benefits in such cases. Additionally, It is worth noting that kTLS requires a kernel version of at least 4.13 for full support [35].

2) TLS v1.3 Support: - One of the primary advancements in the TLS protocol was the release of TLS version v1.3 in 2018 [36]. TLS v1.3 introduces a new handshake procedure that substantially reduces the time required to encrypt a connection, thereby improving the protocol's performance. In our tests, we observed that TLS v1.3 was partially supported in hardware mode, specifically for Chelsio's T6 model. However, we discovered that TLS v1.3 was only supported in the Coprocessor Mode, whereas the Inline Mode exhibited unexpected behavior when attempting to run the mode along with TLS v1.3.

3) QUIC Support: - QUIC is a protocol that offers a low-latency alternative to TCP. It operates over the UDP protocol, as its name suggests, which stands for Quick UDP Internet Connections [37]. One of its main features is built-in encryption for all connections [38]. While the QUIC protocol

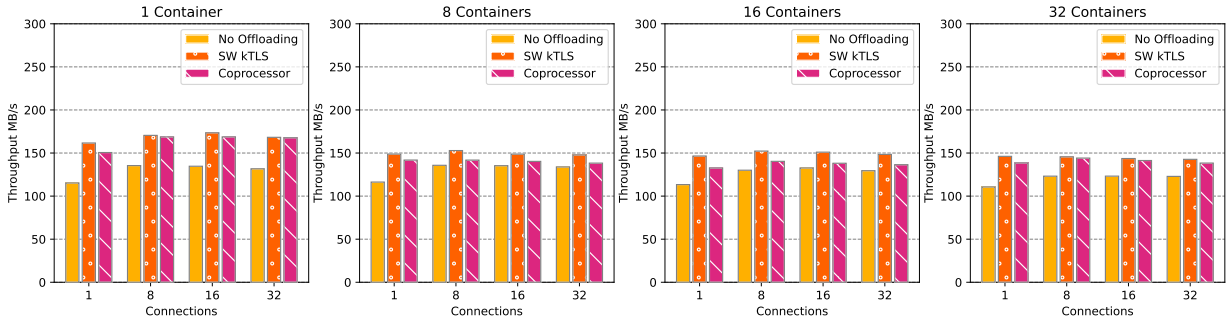

Fig. 7. Download throughput in containerized setups, considering different numbers of concurrently running containers.

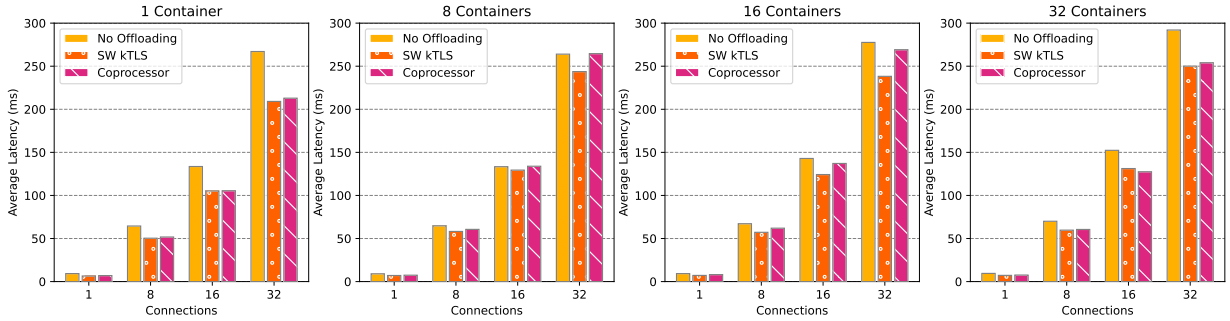

Fig. 8. Latency dynamics during concurrent downloads in containerized environments.

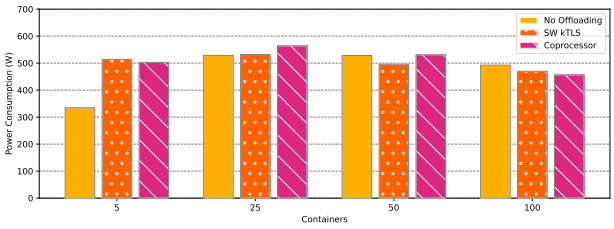

Fig. 9. Power consumption patterns measured as accumulated values in containerized test scenarios.

specification does not explicitly specify the version of TLS, most QUIC implementations use TLS v1.3.

During our testing, we assessed various QUIC implementations, including ngtcp2 [39]. We aimed to configure ngtcp2 with OpenSSL build with kTLS flags targetting Coprocessor Mode. However, enabling QUIC with hardware TLS offloading on our Chelsio T6 posed challenges. Chelsio's documentation indicates that QUIC offloading support will be available in the T7 ASIC.

V. Conclusions and Future Work

In conclusion, our assessment of kTLS in various operational modes within bare-metal and containerized environments reveals a nuanced landscape of performance trade-offs and underscores key considerations for future implementations. Our power consumption analysis revealed interesting trades. Notably, for some scenarios, the Inline Mode exhibited unexpected energy inefficiencies while delivering other performance benefits. This discovery highlights the intricate relationship between performance optimization and power management in offloading technologies.

To address real-world challenges associated with integrating kTLS into closed-source or proprietary applications, this endeavor should be prioritized in future efforts. Developing mechanisms or compatibility layers that enable seamless kTLS integration, even when source code is inaccessible, can significantly enhance the technology's practicality. Furthermore, to keep kTLS at the forefront of secure network communications, vendors must extend their support to newer encryption standards like TLS 1.3 and evolving protocols like QUIC. Adapting to these emerging standards will enhance kTLS's relevance and applicability in contemporary networking scenarios.

Exploring innovative methods to make kTLS more accessible to a broader range of network software presents an ongoing challenge. Subsequent research should focus on creating novel tools, frameworks, or middleware that streamline kTLS integration across various application landscapes, ensuring its advantages are accessible to a broader community of users.

Acknowledgment

This work was supported by the R&D and Innovation Center, Ericsson Ltda, and by the São Paulo Research Foundation (FAPESP), grant 2021/00199-8, CPE SMARTNESS.

References

[1] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding pcie performance for end host networking," in Proceedings of the 2018 Conference of the ACM

Special Interest Group on Data Communication, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 327–341. [Online]. Available: https://doi.org/10.1145/3230543.3230560.
[2] L. Suresh, P. Bodik, I. Menache, M. Canini, and F. Ciucu, "Distributed resource management across process boundaries," in Proceedings of the 2017 Symposium on Cloud Computing, ser. SoCC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 611–623. [Online]. Available: https://doi.org/10.1145/3127479.3132020.
[3] M. Bishop, "HTTP/3," RFC 9114, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9114.
[4] R. R. Stewart and J. M. Gurney, "Optimizing tls for high–bandwidth applications in freebsd," AsiaBSDCon 2015, 2015.
[5] "Kernel tls offload - the linux kernel documentation." [Online]. Available: https://docs.kernel.org/networking/tls-offload.html
[6] B. Pismenny, I. Lesokhin, and L. Liss, "Offload to network devices-rx offload," Netdev 2.2, 2017.
[7] S. Grant, A. Yelam, M. Bland, and A. C. Snoeren, "Smartnic performance isolation with fairnic: Programmable networking for the cloud," in Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 681–693. [Online]. Available: https://doi.org/10.1145/3387514.3405895.
[8] D. Watson, "Ktls: Linux kernel transport layer security," Netdev 1.2, 2016.
[9] J. K. Rott, "Intel® advanced encryption standard instructions (aes-ni)," Feb. 2012. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html
[10] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein, NICA: An Infrastructure for Inline Acceleration of Network Applications. USENIX Association, 2019. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/eran.
[11] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading distributed applications onto smartnics using ipipe," in SIGCOMM 2019 - Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication. Association for Computing Machinery, Inc, aug 2019, pp. 318–333.
[12] B. Pismenny, I. Lesokhin, L. Liss, and H. Eran, "Tls offload to network devices," Netdev 1.2, 2016.
[13] J. Zhao, M. Neves, and I. Haque, "On the (dis)advantages of programmable nics for network security services," in 2023 IFIP Networking Conference (IFIP Networking), 2023, pp. 1–9.
[14] T. Xing, H. Tajbakhsh, I. Haque, M. Honda, and A. Barbalace, "Towards portable end-to-end network performance characterization of smartnics," in Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems, ser. APSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 46–52. [Online]. Available: https://doi.org/10.1145/3546591.3547528.
[15] A. Roulin, "Advancing the state of network switch asic offloading in the linux kernel," Master's thesis, École Polytechnique Fédérale de Lausanne, 2018. [Online]. Available: http://infoscience.epfl.ch/record/254829.
[16] J. Liu, C. Maltzahn, C. D. Ulmer, and M. L. Curry, "Performance characteristics of the bluefield-2 smartnic," CoRR, vol. abs/2105.06619, 2021. [Online]. Available: https://arxiv.org/abs/2105.06619.
[17] "curl docs - rate limiting." [Online]. Available: https://everything.curl.dev/usingcurl/transfers/rate-limiting
[18] "Docker overview." [Online]. Available: https://docs.docker.com/get-started/overview/
[19] "curl - command line tool and library for transferring data with urls (since 1998)." [Online]. Available: https://curl.se/
[20] "wrk - a http benchmarking tool." [Online]. Available: https://github.com/wg/wrk
[21] "collectd – the system statistics collection daemon." [Online]. Available: https://collectd.org/
[22] "systemd — linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man1/systemd.1.html
[23] "Rddtool." [Online]. Available: https://oss.oetiker.ch/rrdtool/
[24] "pandas - python data analysis library." [Online]. Available: https://pandas.pydata.org/
[25] "Numpy." [Online]. Available: https://numpy.org/
[26] D. Nichols, "Coloring for colorblindness." [Online]. Available: https://davidmathlogic.com/colorblind/
[27] "Matplotlib: Visualization with python." [Online]. Available: https://matplotlib.org/
[28] "Seaborn: Statistical data visualization." [Online]. Available: https://seaborn.pydata.org/
[29] "Jupyter." [Online]. Available: https://jupyter.org/
[30] "Podman." [Online]. Available: https://podman.io/
[31] "Telegram messenger." [Online]. Available: https://telegram.org/
[32] "Discord — your place to talk and hang out." [Online]. Available: https://discord.com/
[33] E. Freitas, A. T. de Oliveira Filho, P. R. do Carmo, D. Sadok, and J. Kelner, "A survey on accelerating technologies for fast network packet processing in linux environments," Comput. Commun., vol. 196, no. C, p. 148–166, dec 2022. [Online]. Available: https://doi.org/10.1016/j.comcom.2022.10.003.
[34] A. Shehabi, S. J. Smith, D. Sartor, R. Brown, M. K. Herrlin, J. Koomey, E. Masanet, N. Horner, I. M. L. Azevedo, and W. Lintner, "United states data center energy usage report," Building Technologies Department, Building Technology and Urban Systems Division, Electronics, Lighting and Networks, Sustainable Energy Systems Group, Sustainable Energy Department, Energy Analysis and Environmental Impacts Division, Tech. Rep., 2016.
[35] J. Baldwin, "Tls offload in the kernel," FreeBSD Journal, May/Jun. 2020.
[36] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446.
[37] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9000.
[38] X. Yang, L. Eggert, J. Ott, S. Uhlig, Z. Sun, and G. Antichi, "Making quic quicker with nic offload," in Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC, ser. EPIQ '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 21–27. [Online]. Available: https://doi.org/10.1145/3405796.3405827.
[39] "ngtcp2." [Online]. Available: https://github.com/ngtcp2/ngtcp2