Felipe Aparecido dos Santos Novais

Aludel: Transparent Crypto-Offloading for Enhanced Legacy Application Security

Sorocaba, SP

1, March 2024

Felipe Aparecido dos Santos Novais

Aludel: Transparent Crypto-Offloading for Enhanced Legacy Application Security

Masters dissertation presented to the Postgraduate Program in Computer Science (PPGCC-So) at the Federal University of São Carlos as part of the requirements for obtaining the Master's degree in Computer Science. Research area: Software Engineering and Computing Systems.

Federal University of São Carlos – UFSCar

Center for Science in Management and Technology – CCGT

Postgraduate Program in Computer Science – PPGCC-So

Supervisor: Prof. D.Sc. Fábio Luciano Verdi

Sorocaba, SP

1, March 2024

dos Santos Novais, Felipe Aparecido

Aludel: Transparent Crypto-Offloading for Enhanced Legacy Application Security / Felipe Aparecido dos Santos Novais -- 2024.
68f.

Dissertação (Mestrado) - Universidade Federal de São Carlos, campus Sorocaba, Sorocaba
Orientador (a): Fábio Luciano Verdi
Banca Examinadora: Marco Aurélio Amaral Henriques,, Christian Esteve Rothenberg
Bibliografia

1. Offloading. 2. Acceleration. 3. Cryptography. I. dos Santos Novais, Felipe Aparecido. II. Título.

Ficha catalográfica desenvolvida pela Secretaria Geral de Informática (SIn)

DADOS FORNECIDOS PELO AUTOR

Bibliotecário responsável: Maria Aparecida de Lourdes Mariano - CRB/8 6979

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências em Gestão e Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Dissertação de Mestrado do candidato Felipe Aparecido dos Santos Novais, realizada em 26/03/2024.

Comissão Julgadora:

Prof. Dr. Fabio Luciano Verdi (UFSCar)

Prof. Dr. Marco Aurelio Amaral Henriques (UNICAMP)

Prof. Dr. Christian Rodolfo Esteve Rothenberg (UNICAMP)

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

To my parents, who, despite lacking formal education, tirelessly emphasized its value.
Thank you for being my best teachers.

Acknowledgements

I'd like to thank,

God for the gift of life and countless second chances.

My parents, who showed me the power of self-sacrifice and the importance of discipline.

My brother, who always helped me to get my head in the clouds and my feet on the ground.

Tamires, my life partner, whose love and support became essential during challenging times.

Professor Fábio Verdi, my supervisor. Thank you for showing me the path to science (since my undergraduate years) and for your belief in me, even during moments when I doubted myself.

My colleagues from LERIS and LASID lab (UFSCar), SMARTNESS lab (Unicamp and UFSCar), and peers at Ericsson Research, whose friendship and collaboration enriched my research experience.

To all my friends and their support throughout this journey. I am grateful for those who stood by me in difficult times and understood the sacrifices made. I love you all.

My gratitude also goes to every teacher who has shaped my journey since childhood.

Last but not least, thanks to the R&D and Innovation Center, Ericsson Ltda, and to the São Paulo Research Foundation (FAPESP), for the grant 2021/00199-8, CPE SMARTNESS.

“Non est ad astra mollis e terris via.“

(Seneca)

“First say to yourself what you would be, and then do what you have to do.“

(Epictetus)

“It is not death that a man should fear, but he should fear never beginning to live.“

(Marcus Aurelius)

Abstract

This dissertation explores the essential role of Transport Layer Security (TLS) in securing web applications. Our focus is on exploring the potential of kernel TLS (kTLS) offloading to alleviate resource usage, including CPU time, power consumption, and communication speed. Our main objective is to assess the viability of kTLS offloading, both in software and hardware configurations, to enhance resource efficiency in securing web applications. A variety of offloading strategies are analyzed, such as software-based kTLS implementation that brings cryptographic tasks closer to the Kernel and cutting-edge hardware-accelerated modes such as TOE (TCP Offload Engine) and coprocessor configurations, where we used 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. One challenge identified is the complexity of implementing kTLS in the current context. The discussion digs into the implications of this challenge and its potential impact on the broader adoption of kTLS in real-world applications. To address the difficulty of implementing kTLS and to ease legacy applications in taking advantage of the benefits of hardware offloading, the dissertation introduces Aludel. This solution provides a mechanism for legacy applications to seamlessly support kTLS, even without such support, giving interesting results on resource usage.

Keywords: Offloading, acceleration, cryptography, SmartNIC, carbon footprint.

Resumo

Essa dissertação explora o papel essencial do Transport Layer Security (TLS) na segurança de aplicações web. Nosso foco está em investigar o potencial offloading usando o kernel TLS (kTLS) para aliviar o uso de recursos, incluindo tempo de CPU, consumo de energia e velocidade de comunicação. Nosso principal objetivo é avaliar a viabilidade do offloading via kTLS, tanto em software quanto em hardware, para aprimorar a eficiência de recursos na proteção de aplicações web. Diversas estratégias de offloading foram analisadas, como a implementação de kTLS baseada em software que aproxima as funções criptográficas ao contexto de Kernel, e modos de aceleração de hardware, como o modo TOE (TCP Offload Engine) e o modo coprocessador, da SmartNIC Chelsio T6. Destacamos o imenso potencial do kTLS e das placas de rede em remodelar dinâmicas de desempenho e eficiência para alguns ambientes de rede, considerando os benefícios e possíveis desvantagens de cada abordagem. Um desafio identificado é a complexidade da implementação do kTLS no contexto atual. A discussão aprofunda as implicações desse desafio e seu impacto potencial na adoção mais ampla do kTLS em aplicações do mundo real. Para enfrentar a dificuldade de implementação do kTLS e facilitar a utilização por aplicações legadas dos benefícios do offloading através de hardware, a dissertação apresenta o Aludel. Essa solução oferece um mecanismo para que aplicações legadas suportem kTLS de maneira transparente, mesmo sem suporte explícito, proporcionando resultados interessantes no uso de recursos.

Palavras-chave: Offloading, aceleração, criptografia, SmartNIC, pegada de carbono.

List of Figures

Figure 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. 26
Figure 2 – CPU time analysis in bare-metal scenarios (without containers). 40
Figure 3 – Evaluation of download throughput in bare-metal setups (without containers) to examine how different offloading modes influence download speeds in different environments. 41
Figure 4 – Latency during concurrent downloads in bare-metal environments (without containers). 42
Figure 5 – Power consumption patterns, measured as accumulated values, in bare-metal test scenarios (without containers). 43
Figure 6 – Accumulation of CPU time within a containerized environment. 44
Figure 7 – Download throughput in containerized setups, considering different numbers of concurrently running containers. 44
Figure 8 – Latency dynamics during concurrent downloads in containerized environments. 44
Figure 9 – Power consumption patterns measured as accumulated values in containerized test scenarios. 45
Figure 10 – Aludel is thought to be a way to turn applications that do not have kTLS support a) to make it support kTLS using Aludel as a middleware b). 48
Figure 11 – In Reverse Proxy architecture, a server exists between the client and the destination servers. This server can have capabilities like WAF, load balancing, cache, etc. 51
Figure 12 – Overview of Aludel Proxy data communication. 54
Figure 13 – Aludel Cache is capable of creating a cache copy where it can use `sendfile()` to reduce read operations. 55
Figure 14 – Aludel Proxy CPU Time in bare-metal setup. 56
Figure 15 – Aludel Proxy download throughput performance in bare-metal settings, analyzing the influence of diverse offloading modes on download speeds. 56
Figure 16 – Examination of latency in Aludel Proxy under bare-metal conditions, analyzing the impact of diverse offloading modes on response times. . . 56
Figure 17 – Evaluation of power consumption in Aludel Proxy. 57

Figure 18 – CPU time analysis within a bare-metal setup, showcasing the performance of Aludel Cache. 58
Figure 19 – Aludel Cache download throughput in bare-metal environments, highlighting how various offloading modes impact download speeds. 58
Figure 20 – Latency values of Aludel Cache within bare-metal setups, analyzing the effect of different offloading modes on response times. 58
Figure 21 – Assessment of power consumption in Aludel Cache, performed in a bare-metal setting, emphasizing how different offloading modes influence power efficiency. 59

List of Tables

Table 1 – Comparative Analysis of Offloading Techniques in Related Works 33
Table 2 – Client and Server specifications. 36
Table 3 – Pros and Cons Table of the Runtime Patching Approach 49
Table 4 – Pros and Cons Table of the Binary Patching Approach 50
Table 5 – Tool Descriptions and Capabilities for Binary/Runtime Patching. Columns legend: BP - Binary Patching, RP - Runtime Patching, LA - Local Approach, GA - Global Approach 50

List of abbreviations and acronyms

AES	Advanced Encryption Standard
AES-NI	Advanced Encryption Standard New Instructions
API	Application Programming Interface
HTTPS	Hypertext Transfer Protocol Secure
HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
kTLS	Kernel TLS
Mbps	Megabits Per Second
ms	Milliseconds
ns	Nanoseconds
PoC	Proof of Concept
RFC	Request for Comments
µs	Microseconds
s	Seconds
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
µW	Microwatts
W	Watts
WAF	Web Application Firewall

Contents

1 INTRODUCTION . . . 25
1.1 Context . . . 25
1.2 Research Goals . . . 26
1.3 Contributions . . . 27
1.4 Organization . . . 28

2 BACKGROUND CONCEPTS & RELATED WORKS . . . 29
2.1 Background Concepts . . . 29
2.1.1 Cryptography . . . 29
2.1.2 TLS - Transport Layer Security . . . 29
2.1.3 Offloading . . . 29
2.1.4 SmartNICs - Smart Network Interface Card . . . 30
2.1.5 kTLS - Kernel Transport Layer Security . . . 30
2.1.6 Chelsio's Coprocessor and Inline offloading . . . 30
2.2 Related Work . . . 30

3 EXPLORATORY EXPERIMENTS . . . 35
3.1 Setup . . . 35
3.2 Experiments . . . 36
3.3 Results . . . 39
3.3.1 **Bare-metal Experiments** . . . 39
3.3.2 **Containerized Tests** . . . 42
3.4 Observations . . . 46
3.4.1 Legacy Applications . . . 46
3.4.2 TLS v1.3 Support . . . 46
3.4.3 QUIC Support . . . 46

4 ALUDEL . . . 47
4.1 Problem . . . 47
4.2 Design . . . 48
4.3 Implementation . . . 51
4.4 Experiments . . . 52
4.4.1 Aludel Proxy . . . 53
4.4.2 Aludel Cache . . . 54
4.5 Results . . . 55
4.5.1 Aludel Proxy . . . 55

4.5.2 Aludel Cache 57
4.6 Observations 59

5 CONCLUSIONS REMARKS 61
5.1 Achievements 61
5.2 Conclusions 61
5.3 Future Work 62

Bibliography 63

1 Introduction

1.1 Context

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 (NEUGEBAUER et al., 2018). 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 (SURESH et al., 2017).

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) (BISHOP, 2022). TLS ensures data privacy, integrity, and security in online transactions, making it fundamental to the functioning of Web applications (STEWART; GURNEY, 2015).

kTLS is a software module that provides an interface for offloading the processing of the TLS protocol to specialized hardware or software (Linux Kernel Organization, 2024). The objective of kTLS is to ease the offloading of TLS by providing a standard interface for hardware acceleration of cryptographic functions (PISMENNY; LESOKHIN; LISS, 2017). This approach can significantly reduce the computational burden on server processors and enhance the resource efficiency of distributed applications, even in cloud environments (GRANT et al., 2020). 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 (WATSON, 2016). 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.

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)).

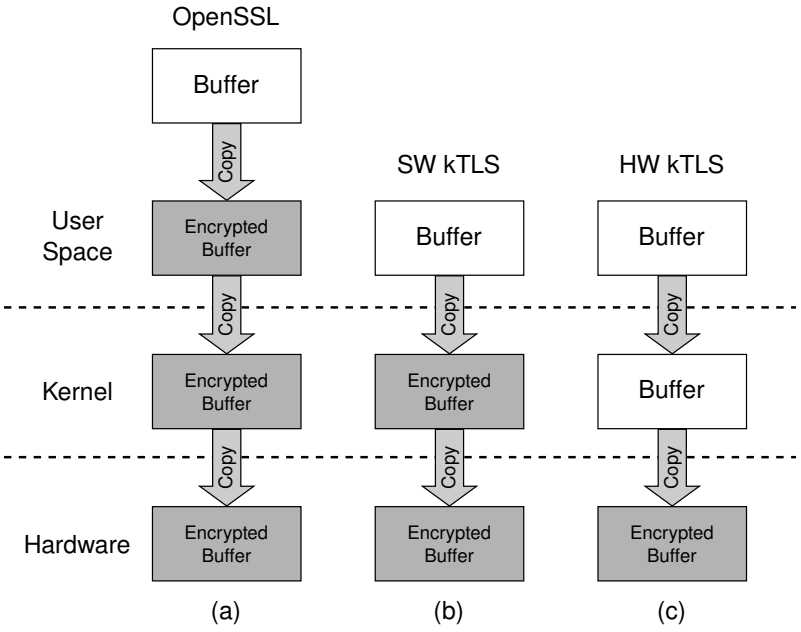

Figure 1 – Illustration of the data flow paths for kTLS (Kernel TLS) and its hardware (HW) and software (SW) modes in comparison to the typical user-space TLS libraries like GNUTLS or OpenSSL. The diagram showcases how kTLS brings TLS operations closer to the kernel, while hardware offloading leverages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

1.2 Research Goals

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 (ROTT, 2012) but in the SmartNIC). Our findings underscore the potential benefits and opportunities kTLS presents when paired with network adapters. We observe positive tendencies in 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.

Another drawback we found is the difficulty of making an application able to support kTLS. Coming from this challenge, we propose Aludel, a middleware solution designed to address the lack of kTLS support in closed-source applications. Aludel brings a

way of seamless support kTLS in applications without the need of any code modification.

By providing this interface between an kTLS unsupported legacy application, Aludel brings extra life to closed-source software, ensuring that it can benefit from contemporary security protocols and encompass the modern security needs in the current cybersecurity-centric world.

Aludel eases the adoption of kTLS even in scenarios where applications are built on dependencies that have absolute zero kTLS capabilities, as such OpenSSL 1.0.2-based applications, which is the case for most of the public-facing Web applications around the Internet (Web Technology Survey, 2024). Providing solutions that, even in this scenario, use transparent hardware offloading provided by kTLS.

Since alchemy, the pseudoscience known for preceding Chemistry, until nowadays, Aludel is a tube used in the process of sublimation, turning something solid into a gaseous state. Cryptography is commonly said to turn something that can be seen clearly with your eyes (as a solid state) into something you can see but can not understand (as a gaseous state). Aludel became the ideal name for a middleware (tube), contributing to the improved execution of cryptography.

The idea of Aludel came after an Exploratory Experiment presented in section 3 that investigates the capabilities of kTLS in the context of software and hardware offloading, particularly using the Chelsio T6 SmartNIC. This initial result was accepted in the IEEE NOMS 2024 (NOVAIS; VERDI, 2024) and this dissertation is heavily based on this paper, mainly Section 3.

1.3 Contributions

Our experiments showed that implementing kTLS requires a Linux kernel version of at least 4.13 and generally a cryptographic backend capable of supporting kTLS, like OpenSSL 3.0.0. So, the target application needs to implement kTLS directly or use a cryptographic backend that supports kTLS. This suggests that the implementation process is not straightforward because of the code changes needed to do this.

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 others may use to evaluate SmartNICs from different vendors.

Finally, as a way to address the challenges of kTLS implementation, we developed

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

Aludel[2] a complete open-source software solution to create a middleware to make applications that would never be able to benefit from the kTLS features like transparent hardware offloading and giving this closed-source application an extra life making them safer by enabling them to encompass the latest security standards and even faster and lighter by providing a way to offloading the processing to a efficient unit like a SmartNIC.

1.4 Organization

The rest of the dissertation is organized as follows: Section 2 presents the related work and briefly explains the main concepts involved in this research. Section 3 explains the methodology used to set up the Exploratory Experiments and evaluate their results, describes experiments done during the research, and discusses its results. Section 4 presents Aludel as a solution to the flaws found in the Exploratory Experiments in Section 3, explains the design & implementation of the solution methodology used to set up the experiments and evaluate their results, describes experiments done during the research, and discusses its results. Finally, Section 5 discusses our concluding remarks and future opportunities.

[2] https://github.com/dcomp-leris/aludel.

2 Background Concepts & Related Works

2.1 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.

2.1.1 Cryptography

Cryptography is the science of secret writing to hide the meaning of a message (PAAR; PELZL, 2010). The way to hide a message is usually by doing some transformation in plain text and turning it into a cipher text, and having the possibility of turning this cipher text back to the plain text. In the context of a computer network, the secrecy of a message being transmitted between two nodes is essential to guarantee four objectives popularly associated with the benefits of cryptography: Integrity, Confidentiality, Non-Repudiation, and Authentication. Without cryptography, modern uses of computer networks like online shopping would be very difficult to implement safely.

2.1.2 TLS - Transport Layer Security

TLS stands for "Transport Layer Security" and is a security protocol used to protect data communication on the Internet (RESCORLA, 2018). It provides end-to-end encryption, server and client authentication, and transmitted data integrity. TLS is often used to protect sensible information such as passwords, credit card information, and general personal data. It is widely used in Web applications such as e-commerce websites, online banking, and social networking and is an integral part of the Internet's security infrastructure.

2.1.3 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 (XING et al., 2022). 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 (ROULIN, 2018).

2.1.4 SmartNICs - Smart Network Interface Card

SmartNIC (Smart Network Interface Card) is a network card with additional processing and memory units over a traditional NIC. This improved network card is designed to speed up network packet processing and perform specific hardware supports, reducing overhead on the main server and improving overall network performance (KATSIKAS et al., 2021). SmartNICs can perform various functions such as packet filtering, load balancing, security, network protocol offloading, and other advanced tasks that require intensive processing and low latency.

2.1.5 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 (Linux Kernel Organization, 2024). This can help reduce CPU load and enhance the overall performance of the system.

2.1.6 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 (LIU et al., 2021). 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.

2.2 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.

In (LIU et al., 2019b), the authors investigate SmartNICs accelerated servers' efficiency in running microservices-based applications in the data center. Offloading microservices resources suitable for SmartNIC's low-power processor could improve the server's energy efficiency without latency loss. However, as a Heterogeneous Computing substrate in the host data path, SmartNICs bring several challenges to a microservices platform: routing and load balancing of network traffic, scaling microservices on heterogeneous hardware, and conflict on shared SmartNIC resources.

In (LIU et al., 2019b), the authors present E3, a microservices execution platform for servers accelerated by SmartNICs. E3 follows the design philosophies of the Azure Service Fabric microservices platform. It extends the critical system components to a SmartNIC to address the abovementioned challenges. E3 employs three essential techniques: ECMP-based load balancing via SmartNICs to host, network topology-aware allocation of microservices, and a data plane orchestrator that can detect SmartNIC overload.

E3 prototype uses Cavium LiquidIO SmartNICs and showed that offloading via SmartNIC can improve cluster energy efficiency by up to 3x and cost efficiency by up to 1.9x with up to 4% latency cost for common microservices, including real-time analytics, an IoT hub, and virtual network functions (VNF).

The paper featured on (ERAN et al., 2019) discusses the usage of FPGA-based SmartNICs to accelerate general-purpose cloud applications and their capabilities for offloading hypervisor network infrastructure. NICA (ERAN et al., 2019) is presented as a hardware and software framework designed to accelerate the data plane of applications on F-NICs[1] 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 (LIU et al., 2019a) 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 (LIU et al., 2019a), 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.

[1] Shortened term for SmartNICs utilizing FPGA technology.

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 (PISMENNY et al., 2016) 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 (ZHAO; NEVES; HAQUE, 2023) 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[2] 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 (ZHAO; NEVES; HAQUE, 2023) 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 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.

The Table 1 provides a comparison of various characteristics across different

[2] Commonly employed acronym for SmartNIC.

Table 1 – Comparative Analysis of Offloading Techniques in Related Works

	E3	NICA	iPipe	Eran	Zhao	Aludel
Hardware Offloading	✓	✓	✓	✓	✓	✓
Software Offloading	✓	×	✓	✓	×	✓
Transparent Offloading	✓	×	×	✓	×	✓
Cryptographic Offloading	×	×	✓	✓	✓	✓
Backward Compatibility	×	×	×	×	×	✓
Uses kTLS	×	×	×	✓	×	✓

works. These characteristics include hardware and software offloading, transparency in implementation, cryptographic functions offloading, backward compatibility with legacy applications, and using kTLS technology. Each characteristic reflects different aspects of offloading techniques and their implications on system performance, energy efficiency, compatibility with existing technologies, and security considerations.

3 Exploratory Experiments

3.1 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 2. All other configurations used can also be seen in Table 2.

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 (cURL Docs, 2024). 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 (DOCKER, 2024) 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.

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 2 – Client and Server specifications.

To conduct the experiments and collect essential data samples, we employed a combination of tools, including `cURL` (STENBERG, 2024) and `wrk` (GLOZER, 2024), 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.

Collecting the metrics related to the CPU time was done using CollectD (FORSTER, 2024) 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, which 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. It is important to notice that the Intel RAPL interface will gather power consumption from every sensor the motherboard can provide, including PCI devices like the Chelsio T6. No external energy other than the power supply for the motherboard is used to provide power to the motherboard and its peripherals, such as the T6, so indeed, the Intel RAPL gatherings represent the total energy consumption it could extract for every available sensor.

3.2 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 (POETTERING, 2024). 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.

The experiments had 2 sets of parameters. The experiments related to CPU time and Power Consumption were done using a 30GB file that was downloaded a hundred times to measure the CPU time and Power consumption while this file was downloaded. And to collect and measure the Throughput and Latency on the client side, we used wrk to download a 10MB file over 10 minutes in 10 iterations (100 minutes total), varying the number of connections between single, 8, and 16 connections.

During our testing and analysis, we tracked the main metrics that may affect the decision about using offloading: 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 s 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 (µW) 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 (OETIKER, 2024). To further analyze the data and generate graphics, they were exported to CSV and processed through Python using libraries like Pandas (MCKINNEY, 2024) and Numpy (OLIPHANT, 2024). For plotting this data into a visualization[1], libraries like Matplotlib (HUNTER, 2024), and Seaborn (WASKOM, 2024) were used, and to enhance their usability, a Jupyter (PEREZ; GRANGER, 2024) 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.

[1] IBM's accessibility palette for visually impaired people was used to improve contrast for people that suffer from color blindness (NICHOLS, 2024).

The Podman (Red Hat, 2024) 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 applications like Telegram (DUROV; DUROV, 2024) and Discord (CITRON, 2024). In this way, if there was a malfunction during the experiments, we would know and have external logs of these occurrences.

3.3 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.

3.3.1 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 2.

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.*

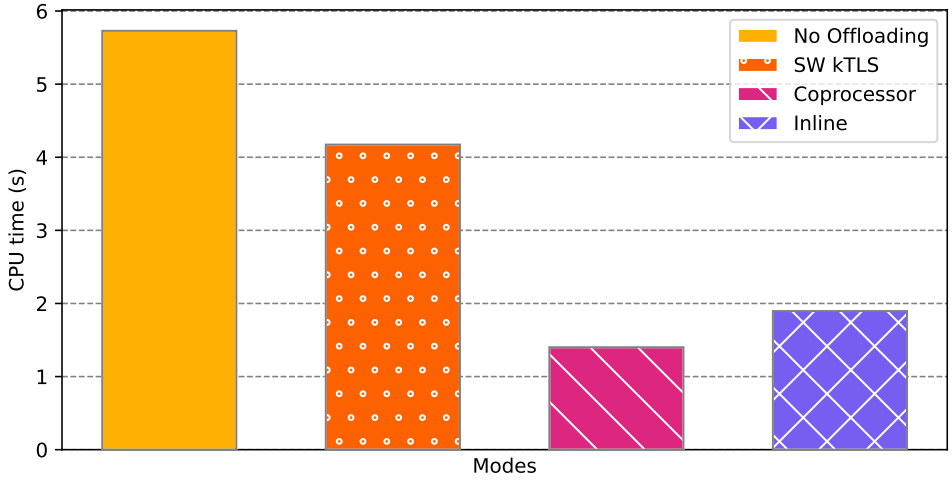

Figure 2 – CPU time analysis in bare-metal scenarios (without containers).

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 2.1. **Takeaway:** *Hardware offloading delivered superior throughput performance when dealing with multiple parallel connections.*

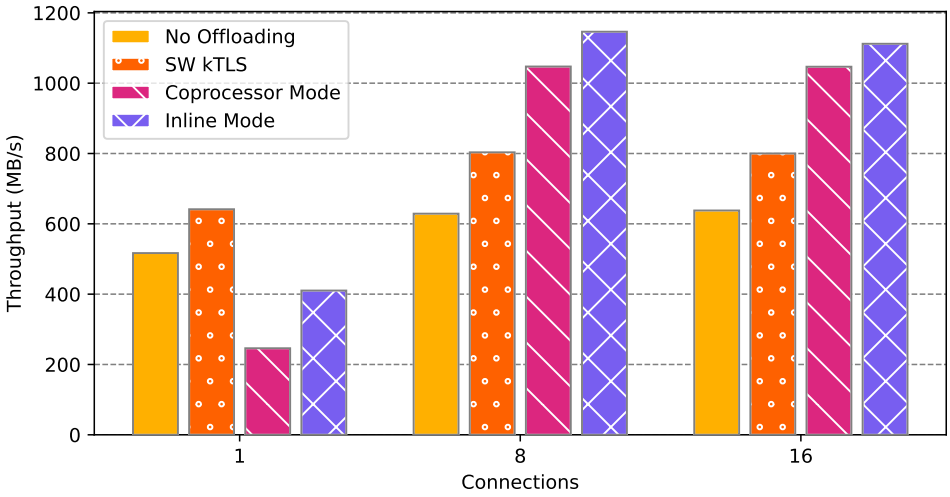

Figure 3 – Evaluation of download throughput in bare-metal setups (without containers) to examine how different offloading modes influence download speeds in different environments.

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.*

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 (FREITAS et al., 2022). Consequently, although Inline Mode may enhance speed, it appears to compromise

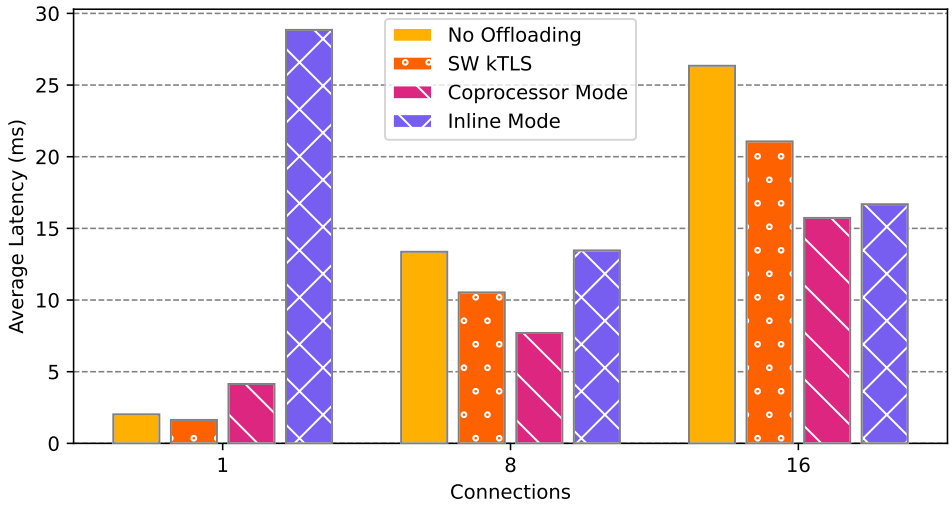

Figure 4 – Latency during concurrent downloads in bare-metal environments (without containers).

energy efficiency in the process.

In contrast, the Coprocessor Mode presented an intriguing difference. It exhibited slight power efficiency 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 (SHEHABI et al., 2016), 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.*

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.

3.3.2 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

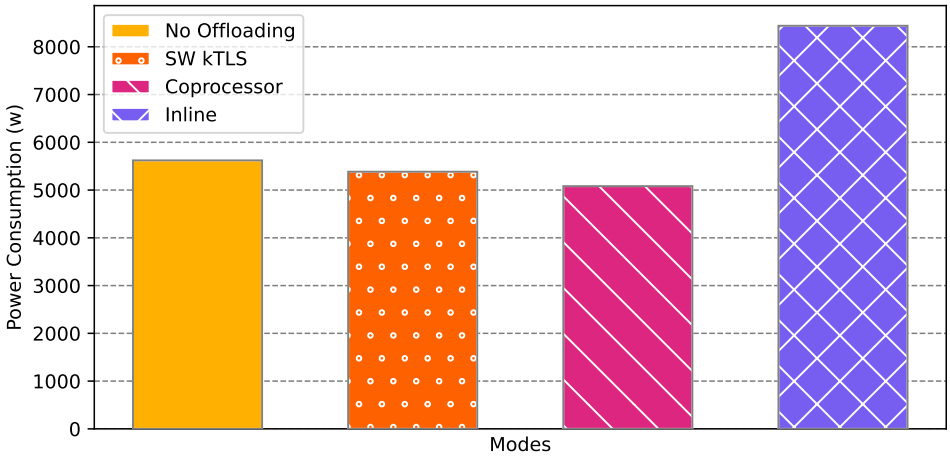

Figure 5 – Power consumption patterns, measured as accumulated values, in bare-metal test scenarios (without containers).

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 coprocessor 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:** *Hardware kTLS with Coprocessor mode showed a good CPU reduction overall, except for the 100 containers scenario, where it performed close to the no offloading scenario. Over time, as*

resources become shorter, Software kTLS may offer a competitive advantage compared to the alternatives we see in the 100 containers scenario.

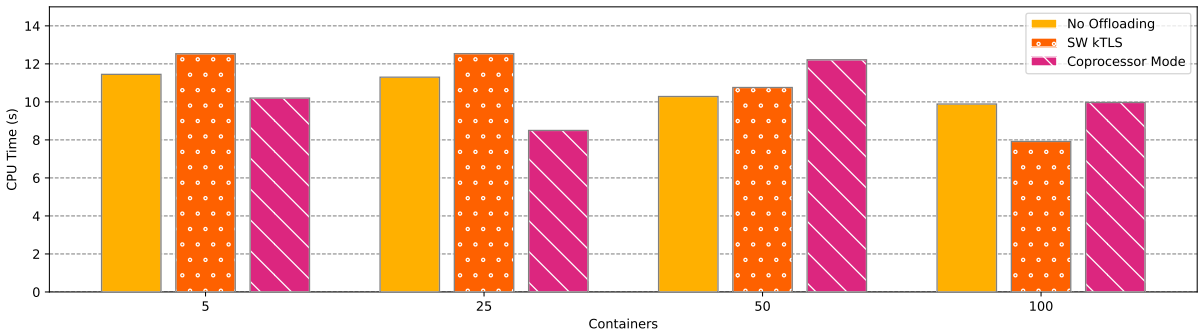

Figure 6 – Accumulation of CPU time within a containerized environment.

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) compared with the bare-metal (9.36Gbps). It is important to mention that scenarios over 32 containers were tested, and they seem to follow the same pattern with very slight differences, so from our discretion, we omitted results from scenarios greater

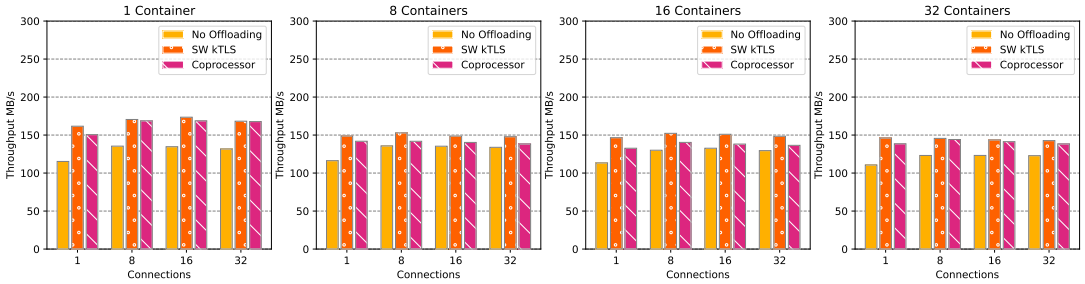

Figure 7 – Download throughput in containerized setups, considering different numbers of concurrently running containers.

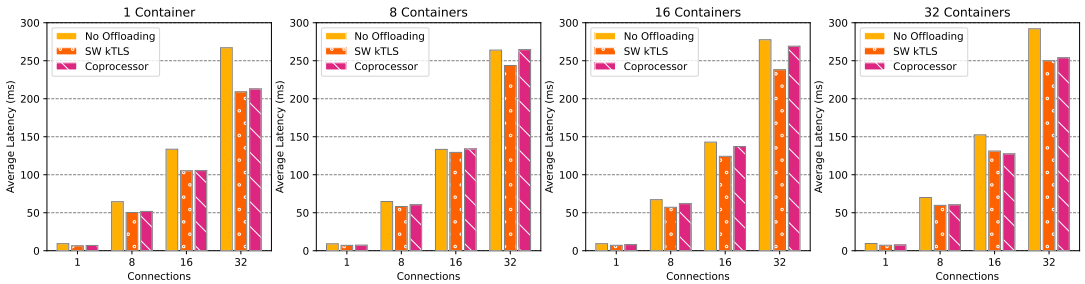

Figure 8 – Latency dynamics during concurrent downloads in containerized environments.

than 32 containers. **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, the Coprocessor mode performed worse than Software kTLS. **Takeaway:** *Software kTLS consistently exhibited lower latency than the other modes, following the trend in throughput as expected.*

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. This result is very counter-intuitive as we can see that the no offloading for the 5 containers scenario consumed lesser power when compared to others. And surprisingly, the hardware mode for the 100 containers scenario we good less power consumption than the coprocessor mode, which was worse in the other scenarios.*

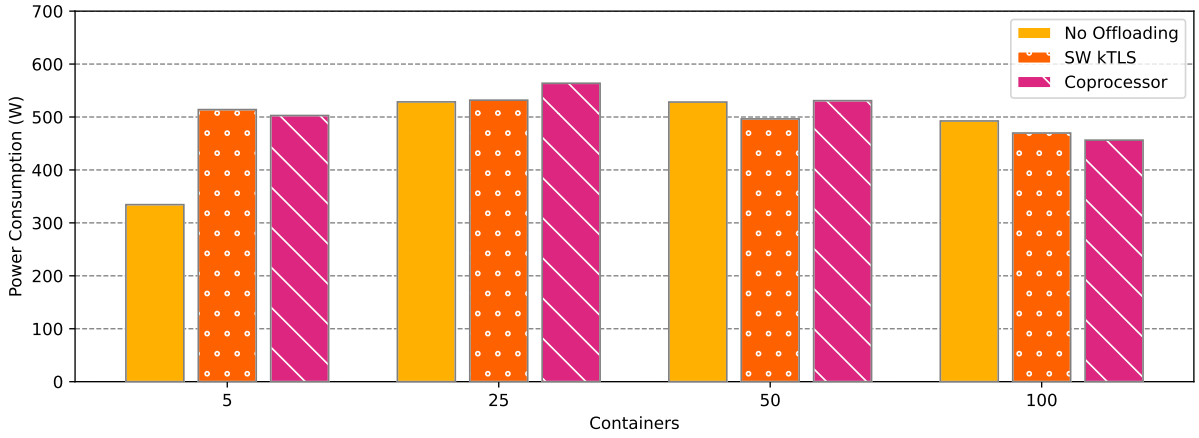

Figure 9 – Power consumption patterns measured as accumulated values in containerized test scenarios.

3.4 Observations

3.4.1 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 support 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 be unable 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 (BALDWIN, 2020).

3.4.2 TLS v1.3 Support

- One of the primary advancements in the TLS protocol was the release of TLS version v1.3 in 2018 (RESCORLA, 2018). 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.

3.4.3 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 (IYENGAR; THOMSON, 2021). One of its main features is built-in encryption for all connections (YANG et al., 2020). While the QUIC protocol 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 (TSUJIKAWA, 2024). 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.

4 Aludel

4.1 Problem

In the fast-changing world of Transport Layer Security (TLS), keeping up with advanced security features, like kernel TLS (kTLS), is crucial for cybersecurity.

Yet, for applications without access to their source code, staying current with the latest security measures is tough. The challenge is that kTLS brings a whole new way of managing sockets, something most applications are not used to. Unlike before, where TLS/SSL capabilities were added through standard sockets using external libraries like OpenSSL, GNUTLS, WolfSSL, etc., kTLS is built directly into the operating system as a socket type. So it is complicated, especially for network administrators who may not know how to code or may lack the source code to add kTLS support to closed-source applications.

When the source code is unavailable, these applications face difficulties keeping up with TLS advancements and adopting new security protocols. This is usually the case for legacy proprietary applications.

Implementing kTLS means changing the entire cryptographic backend and adopting a new socket approach, which can be a challenging process. Some cryptographic backends, including OpenSSL, GnuTLS, and WolfSSL, try to make it easier by seamlessly supporting kTLS. At present, OpenSSL is one of the most used cryptographic backends according to data like the cURL user survey (STENBERG, 2023), and has the most detailed information about kTLS support.

However, it's not all smooth sailing. There are specific requirements for using kTLS through OpenSSL, like needing version 3.0.0 or later. Unfortunately, data from the WebTechSurvey (Web Technology Survey, 2024) website shows that most websites on the Internet still rely on OpenSSL 1.0.2 for their security, creating a barrier to integrating kTLS seamlessly.

While OpenSSL version 3.0.0 or later is usually needed for kTLS, the widespread use of OpenSSL 1.0.2 causes compatibility issues for many applications. This not only limits potential security improvements but also blocks the use of specialized hardware like SmartNICs, which is important for achieving the best security and performance benefits.

In response to this challenge, Aludel comes into play as a middleware solution. It tackles the limitations faced by closed-source applications without native kTLS support. By integrating kTLS without requiring extensive code changes, Aludel enables these

applications to benefit from improved security and performance. The middleware simplifies complexities, aligning closed-source applications with current security standards and ensuring their continued relevance in today's ever-changing digital landscape.

4.2 Design

In a landscape marked by rapid TLS progress, applications without access to source code often fail to adopt modern security features like kTLS due to compatibility constraints. Aludel acts as middleware for seamlessly integrating kTLS support into these closed-source applications. This empowers them to harness enhanced security and performance benefits primarily through hardware offloading using specialized hardware, such as SmartNICs. Aludel revitalizes these applications by abstracting complexities and bypassing the need for extensive code modifications, bridging them with contemporary security standards and ensuring their relevance in today's dynamic digital environment.

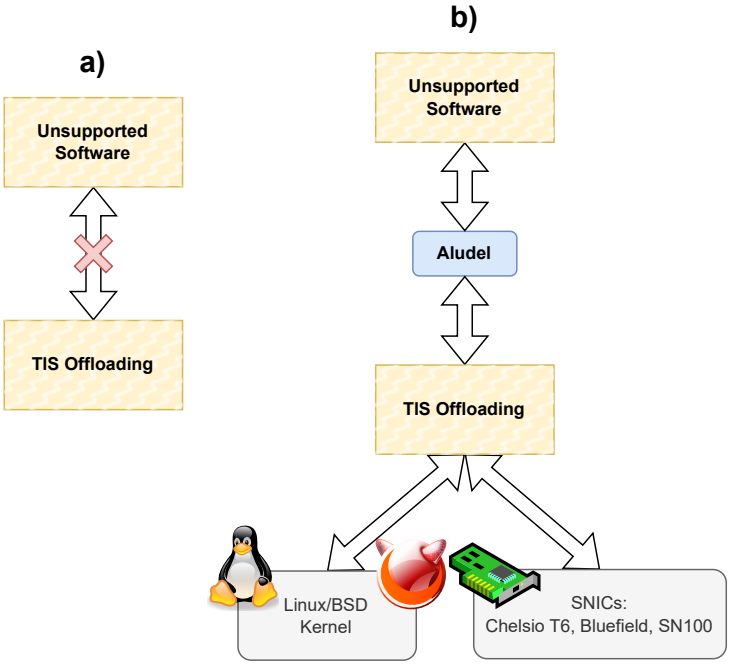

Figure 10 – Aludel is thought to be a way to turn applications that do not have kTLS support a) to make it support kTLS using Aludel as a middleware b).

Aludel was idealized as a software that enabled legacy apps to support kTLS without needing their source code. kTLS is a kernel module with a new socket type that enables the kernel to do the cryptographic functions instead of the library doing this workload in the userspace context. Generally speaking, you can not change the socket type during its creation for an already compiled app. This is also tricky because the userspace library used for encryption goes through many abstractions to process the cryptographic

workload before sending it to the NIC. Going straight to the socket creation without changing these moving parts would generate many complications.

Another way to approach how Aludel could be created is by thinking about how people usually implement kTLS. Some use the interface directly with a socket type that will do all the work seamlessly. In contrast, others will opt for a more familiar approach, such as using a cryptographic backend that supports kTLS transparently, such as OpenSSL 3.0.0. Unfortunately, a significant part of the internet uses OpenSSL version 1.0.2, which does not support kTLS. Following this train of thought, a possible approach would be to implement the missing parts from the OpenSSL 3.0.0 kTLS compatibility in older versions. Modifying behavior within closed-source software, even if the library used is open-source, presents a complex challenge. Techniques from program analysis, like runtime patching of system calls and offline patching of binaries, emerge as potential solutions. We think that two possibilities work to achieve this compatibility:

Online Method - Runtime Patching: Runtime patching involves modifying an application's behavior while running. This can be achieved by intercepting function calls (function hooking) and changing their parameters or return values. Runtime patching is useful to dynamically modify an application's behavior without altering its executable file.

Offline Method - Binary Patching: In binary patching, you directly modify the executable file's binary code using a hex editor or specialized tools. This approach involves locating specific instructions or data in the binary and replacing them with new instructions or data. Binary patching is often used to alter hardcoded values, change conditional logic, or remove or add specific functionality.

Another question to answer is how we could detect which application needs to be targeted by Aludel. There are two distinct approaches we think would be possible to select the application for intervention:

Local Approach: In this scenario, the user specifies the target application to be patched. So, the user has to know which applications need to use Aludel.

Global Approach: Aludel operates as a background OS daemon, continuously monitoring applications that attempt to utilize TLS. If an application uses TLS but not kTLS, Aludel employs patching techniques to make it compatible with kTLS and enable the utilization of its offloading capabilities.

Table 3 – Pros and Cons Table of the Runtime Patching Approach

PROS	**CONS**
Could use a Global approach to target every application that uses TLS	Probably harder than binary patching.
It may support a wider variety of applications	Need an application running with a lot of privileges

Table 4 – Pros and Cons Table of the Binary Patching Approach

PROS	**CONS**
It may be easier than runtime patching	Possible need to be specific to the application (Local Approach)
The binary could be patched and distributed without Aludel in other hosts	Usually the binary patching countermeasures are more complex to bypass
Less intrusive to the host OS	

We brainstormed to enumerate the possible tools and libraries to implement both approaches. The results of the brainstorm was summarized in Table 5

Table 5 – Tool Descriptions and Capabilities for Binary/Runtime Patching. Columns legend: BP - Binary Patching, RP - Runtime Patching, LA - Local Approach, GA - Global Approach

Tool	**Description**	**BP**	**RP**	**LA**	**GA**
PyDBG	Python debugger for analyzing and manipulating software execution.		✓	✓	✓
PyASM	Python library for low-level assembly operations and analysis.	✓	✓	✓	✓
HexEdit	A graphical tool for viewing and editing binary files in hexadecimal format.	✓		✓	
python-ptrace	Python module for controlling and observing processes, useful for debugging and reverse engineering.		✓	✓	✓
pygdb	Python interface for GDB (GNU Debugger) to facilitate debugging and exploration of software.	✓	✓	✓	✓
scapy	A Python library for crafting and decoding packets, which is useful for network protocol analysis.		✓		✓
pwntools	Python library and framework for exploiting and reverse engineering binary files.	✓		✓	
eBPF	Framework for fine-tuning and observing Linux kernel behavior through custom programs at critical points.		✓		✓

Implementing these program analysis techniques is rather complex. Usually, it involves a lot of knowledge of the object in analysis, such as in the case of Aludel, a specific knowledge about the kernel module or library like OpenSSL. But there are techniques and even previous works like (BATES et al., 2014) that show it is possible to use techniques such as Library Shiming to provide security through the use of patching using program analysis techniques.

A Reverse Proxy strategy was thought to be the first Proof of Concept (PoC) on

whether we could make an application without any kTLS support to take advantage of this module, we thought of using sockmap and splice (SITNICKI, 2023) to do low-level packet forwarding, but the simplest way to implement this PoC was just taking advantage of NGINX capabilities of Reverse Proxy, kTLS support and caching to implement the Aludel PoC.

4.3 Implementation

Addressing the intricacies of modifying behavior within closed-source software, which lacks accessible source code, poses a considerable challenge. To tackle this necessity, our approach was inspired by solutions like ModSecurity and Authelia. ModSecurity is a Web Application Firewall implementation that commonly utilizes a Reverse Proxy method to detect Web vulnerabilities. And Authelia focuses on single-sign-on to any application through a Reverse Proxy approach.

Reverse Proxies are situated between clients and destination servers and intercept requests before forwarding them. They are commonly employed for implementing caches, load balancers, and Web application firewalls.

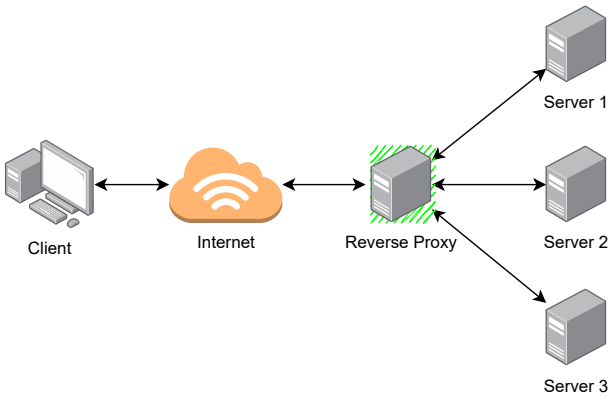

Figure 11 – In Reverse Proxy architecture, a server exists between the client and the destination servers. This server can have capabilities like WAF, load balancing, cache, etc.

Our initial step involved implementing Aludel using the reverse proxy method. Recognizing that NGINX supports kTLS with OpenSSL 3.0.0, we created a self-contained binary based on NGINX, incorporating all required libraries like — OpenSSL, Zlib, and PCRE—compiled into a single binary named Aludel PoC. This binary streamlines the process for network administrators, eliminating the need for manual configuration activation to use kTLS and preventing conflicts with the OpenSSL version on the host OS. This resulted in a 14MB binary capable of intercepting connections between clients and destination servers and employing kTLS seamlessly.

A common challenge in reverse proxy implementation is the inherent problem of introducing an extra hop before reaching the destination host and increasing latency. This is a known problem in Service Mesh architecture, where sidecar proxy optimization is a challenge still to be tackled. Initially conceived as a Proof of Concept, our objective was to explore the feasibility of bridging legacy applications that lack support for the latest TLS protocol versions and other optimizations. Aludel offers the additional advantage of simplifying kTLS usage, requiring only that the network administrator ensures the OS kernel and modules are loaded. With Aludel acting as the intermediary, the application seamlessly processes connections originating from clients through the Aludel Proof of Concept using kTLS.

4.4 Experiments

The setup for Aludel PoC experiments was very similar to the ones executed in the kTLS Exploratory Experiments. The same scenario for the hosts was set, a client host and a server host, precisely the same hardware used for the Exploratory Experiments with its specs in Table 2. In the case of Aludel PoC, instead of the bare-metal and containerized tests, we did only a bare-metal test, meaning that all service instances were hosted on the same server.

For the client host, the client applications were cURL, wrk, and the supporting scripts needed to run the tests.In the server host, a particular target binary was set. It was a specifically crafted NGINX version 1.11.1 (released on 31 May 2016), OpenSSL 1.0.2f (January 26, 2016), PCRE2 10.42 (Dec 12, 2022), and Zlib 1.3.1 (January 22, 2024). Some dependencies, like Zlib and PCRE2, have up-to-date versions. Still, the core functionalities we are testing are based on an old version of NGINX and OpenSSL. The choice of OpenSSL 1.0.2f was not by coincidence. We decided to target this version because of two motivations: it does not support kTLS by any chance, and second to WebTechSurvey (Web Technology Survey, 2024), this is the most used version of OpenSSL around the world by the time of this publication. In summary, this target binary was, in fact, an example of a possible legacy application that does not have any kTLS capability.

It is important to note that both binaries, the Aludel PoC and the target binary, resided in the same host and shared some resources. The same CPU affinity scenario from the Exploratory Experiments was in place, meaning that we isolated a CPU core only for the experiment's binaries to use.

Using the setup described in the previous section, we then ran our experiments. There were two versions of PoC that we tested v0.1 (also called Aludel RProxy) and v0.2 (also called Aludel Cache). Each PoC description and results will be explained in separate sections.

Like in the Exploratory Experiments with the Chelsio Board and Aludel PoC experiments, four metrics were collected and processed: CPU, Power, Throughput, and Latency. And just like the Exploratory Experiments, we have a scenario for different modes:

- **Without Aludel:** In this mode, the client accessed the legacy application directly without any server in the middle. The legacy application is based on OpenSSL 1.0.2f and has no kTLS capabilities.
- **Aludel with Software kTLS:** The software mode brings TLS connections close to the kernel and reduces context switches. No additional hardware is necessary in this case.
- **Aludel with Coprocessor mode:** Uses the Chelsio T6 coprocessor mode for cryptography. This mode is analogous to AES-NI.
- **Aludel with Inline mode:** Uses Chelsio T6 TOE with the cryptography offloading enabled.

Like the Exploratory experiments, the Aludel experiments used Scaphandre CollectD to gather CPU time and Power Consumption while we downloaded a big file (30G). And to collect and measure the throughput and latency on the client side, we used wrk with a rather smaller file (10MB), varying the number of connections between single, 8, and 16 connections.

4.4.1 Aludel Proxy

In the first iteration of Aludel PoC, which we called v0.1 codename Aludel Proxy, the system was a reverse proxy that had kTLS capabilities and would be in the middle of the client and the destination service (target binary), which is a legacy app. The Figure 12 depicts exactly the Aludel Proxy scenario. We receive a connection from the client, and Aludel is the point in the middle that terminates the TLS connection, starts another connection with the destination client, and then forwards the requests and responses between the communication of these two.

Through the use of Aludel Proxy we were capable of using kTLS to make the connection between the client and the server. This could represent a big thing if it means that the legacy application would operate using insecure protocols and cipher suites. Adding this extra layer would not only permit us to offload this connection close to the kernel but also make it safer by reducing the exposure of the legacy app by undesirable agents in the network. However, we had some performance degradation in general by using the Aludel PoC v0.1. This is a well-known problem since we add extra hops between the

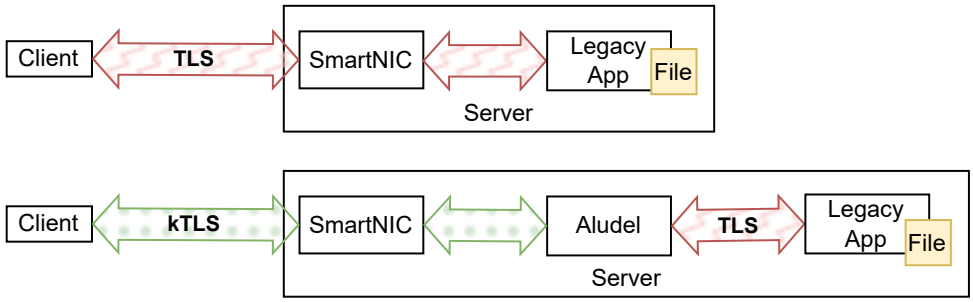

Figure 12 – Overview of Aludel Proxy data communication.

client and server, it is indeed expected that we will have some performance degradation, coming from the design that we took inspiration from the sidecar proxy pattern and represents a big challenge for vendors in the current literature as we can see from Istio page. The overall thing we got from this is that comparing an application without an extra hop that performs only traffic forwarding is not so great for the Aludel PoC v0.1 scenario. But we did not give up. It may seem unfair to compare Aludel with a scenario without extra hop between the client and server. Still, one of the Aludel premises is that we should consider that we would not only try to bring safety but also to get things different resources even for very outdated applications.

4.4.2 Aludel Cache

The (BALDWIN, 2020) indicates that one of the main gains we could have from kTLS is when we use it combined with file transfer. Hence, we get to use the `sendfile()`. Traditionally, the system calls `read()` and `write()` are used to transfer files from user space to kernel space. Using `sendfile()`, we avoid some of these operations and make this access more resource-efficient.

The main problem with Aludel Proxy is that it has no control over the file in the destination server. The same process of `read()` and `write()` buffer operations will still happen on the destination server. So, even with the kTLS and hardware offloading, the `sendfile()` operation is not fully utilized, as Aludel Proxy does not have control over the file.

So, to take advantage of `sendfile()` capabilities with kTLS, we thought that it could be a good idea to try to put a copy of the file that the destination service legacy app news so that Aludel could serve it and process some of the files in the kernel space using kTLS.

As Aludel PoC is based on the NGINX Web server, which has cache capabilities, we got this working with some minor configurations. We could see some surprises coming from this setup, and you will see the results we will show later immediately.

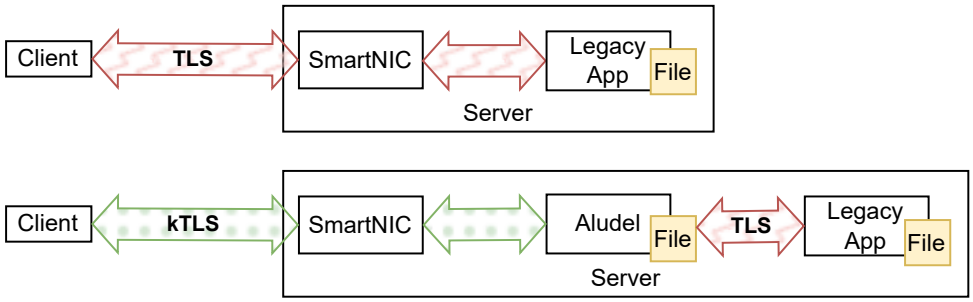

Figure 13 – Aludel Cache is capable of creating a cache copy where it can use `sendfile()` to reduce read operations.

4.5 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.

4.5.1 Aludel Proxy

CPU - It is significantly perceptible that Aludel Proxy performed worse than any offloading mode, software, and hardware. The main reason is the supposedly positive results from the Exploratory Experiments are related to the possibility of using `sendfile()` to avoid some copy operations, so apart from the forwarding operations, the destination server still uses the `write()` and `read()` operations to create buffers to transfer the files from different contexts and then to the socket as Figure 1. But, even with insignificant results we can notice that between the offloading results, the inline mode was slightly better than Software kTLS and Coprocessor mode.

Throughput - As seen in CPU time results, a similar degradation was found for throughput. For throughput, it was even more evident, reaching a 2x performance difference. Between the offloading methods, we could see that as the number of connections increased, the Inline mode performed better than the other offloading methods.

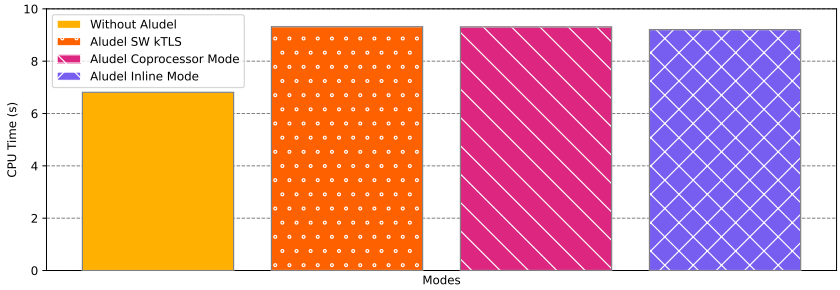

Figure 14 – Aludel Proxy CPU Time in bare-metal setup.

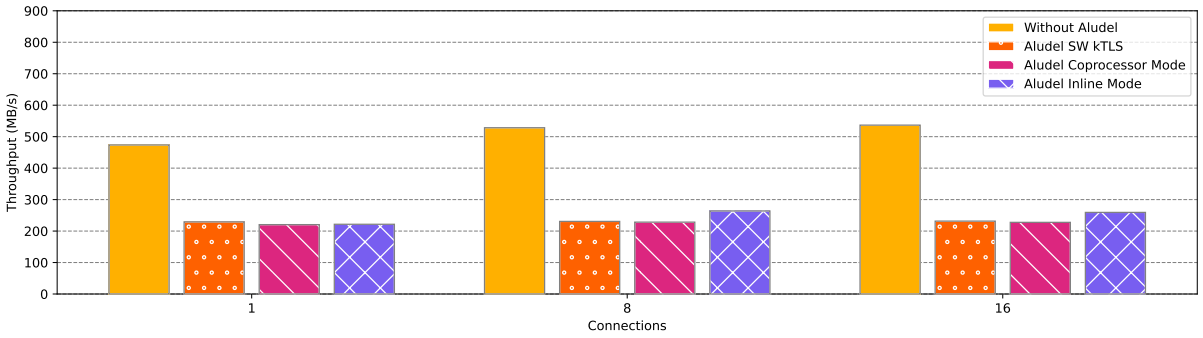

Figure 15 – Aludel Proxy download throughput performance in bare-metal settings, analyzing the influence of diverse offloading modes on download speeds.

Latency - As expected, the latency showed similar results to what we had for CPU and Throughput. In the case of latency, we could perceive that the degradation was proportional to the number of connections.

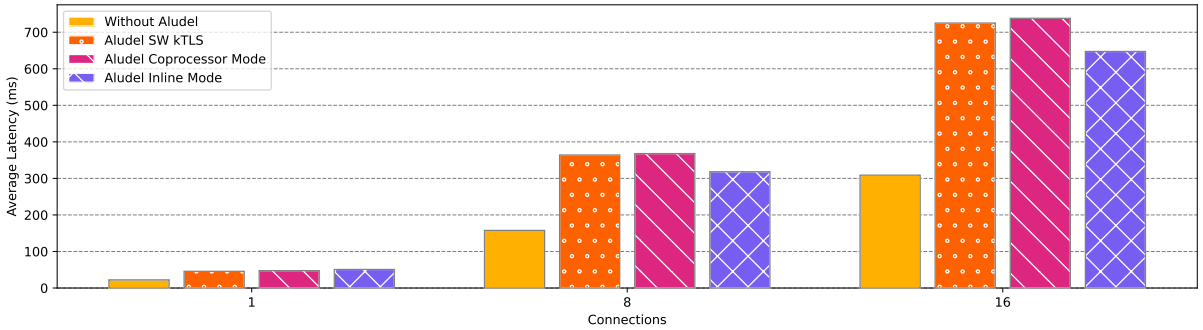

Figure 16 – Examination of latency in Aludel Proxy under bare-metal conditions, analyzing the impact of diverse offloading modes on response times.

Power Consumption - As per the Exploratory Experiments power consumption results for bare-metal, in Aludel Proxy, we got a different pattern related to the usage of offloading methods. For instance, in the Exploratory Experiments Figure 5, the Inline mode demonstrated to be very power consumption intensive while the others would be considerably lesser. For Aludel Proxy, this was not the case, this results leans towards that the main reason of Inline mode in Exploratory tests being so power intensive is because

the high IO necessity while using this mode as we are not really using `sendfile()` in Aludel Proxy.

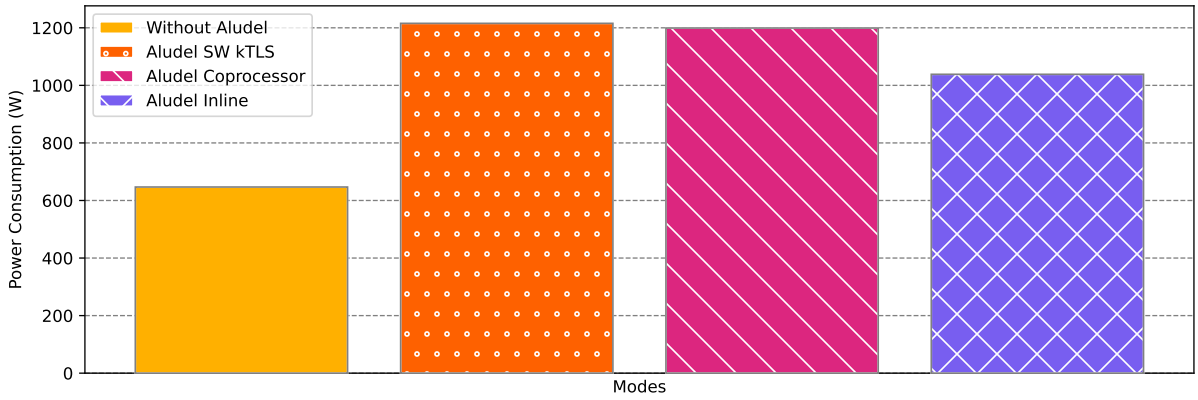

Figure 17 – Evaluation of power consumption in Aludel Proxy.

In the end, the experiments in Aludel Proxy served the purpose of showing that one of the main gains with kTLS resides in the `sendfile()` capabilities and also that when using it just for traffic forwarding, the results can be worse because you are adding an extra hop to the connection. It is not doing any optimization in the communication that is happening. The advantage of using Aludel Proxy is on the matter of having a legacy application to communicate with safer protocols. Also, it is supposed to be more efficient than just using another server to do things like this as the results with inline mode showed to us.

4.5.2 Aludel Cache

CPU - For Aludel Cache we could see from the get go that we had some performance increase. It is expected that the only time we ask the destination server for the file is the first request we have for this file, and then it will be cached in Aludel Cache and served as if the file was from Aludel. In this way we could take advantage of the `sendfile()` operation and get amazing results as shown in Figure 18. The CPU showed a reduction of more than 3x when using any offload method, highlighting again the inline mode for performing slightly better than the others.

Throughput - For throughput, we got a peculiar result. For the single connection scenario, only the Software kTLS mode performed better than without the Aludle Cache, and the Inline mode was particularly worse than the others. But for the consequent number of connections, in fact, the inline mode showed an impressive gain over the facts. This somewhat followed the same trend of throughput in the bare-metal results from the Exploratory Experiments, if we compare Figure 3 and Figure 19 we can see that in both cases, the hardware offloading was not a good option for a single connection, but after this the reached interesting results.

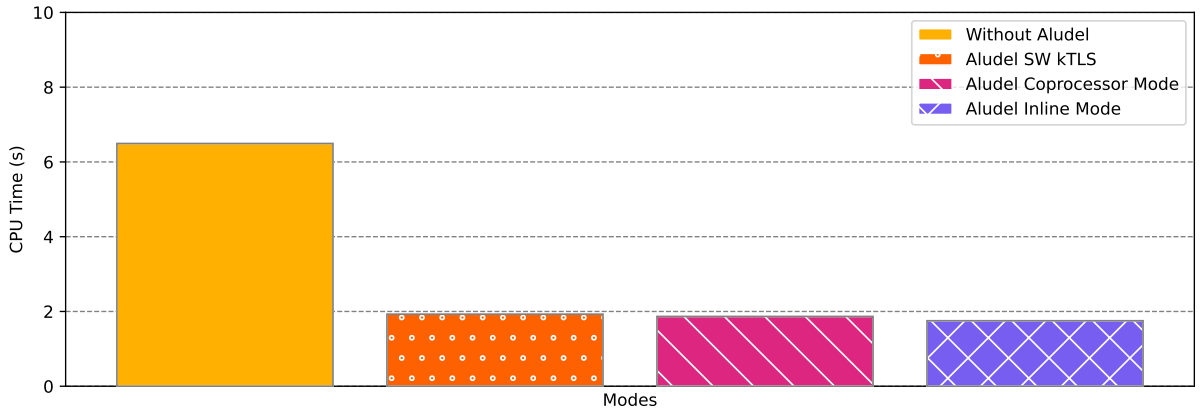

Figure 18 – CPU time analysis within a bare-metal setup, showcasing the performance of Aludel Cache.

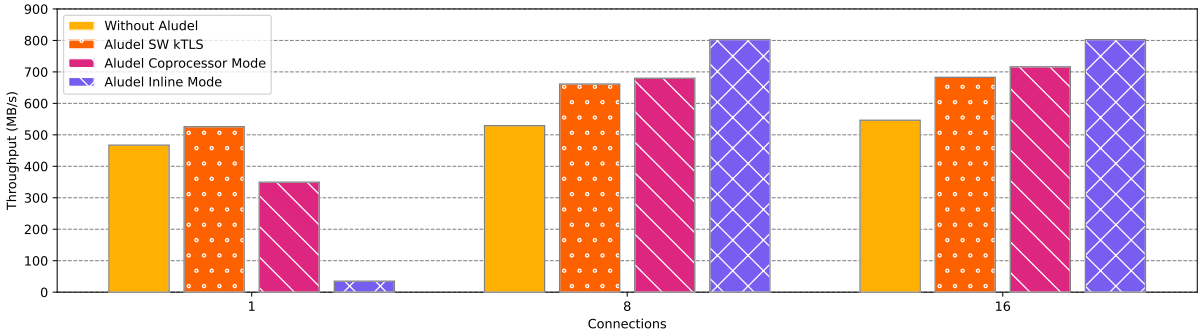

Figure 19 – Aludel Cache download throughput in bare-metal environments, highlighting how various offloading modes impact download speeds.

Latency - Latency showed a consistent result compared to throughput. We got a perceptive gain in throughput as we increased the number of connections. For the single connection scenario, the difference between throughput was irrelevant. and we could see that latency-wise, the hardware offload methods were way superior as the number of connections increased.

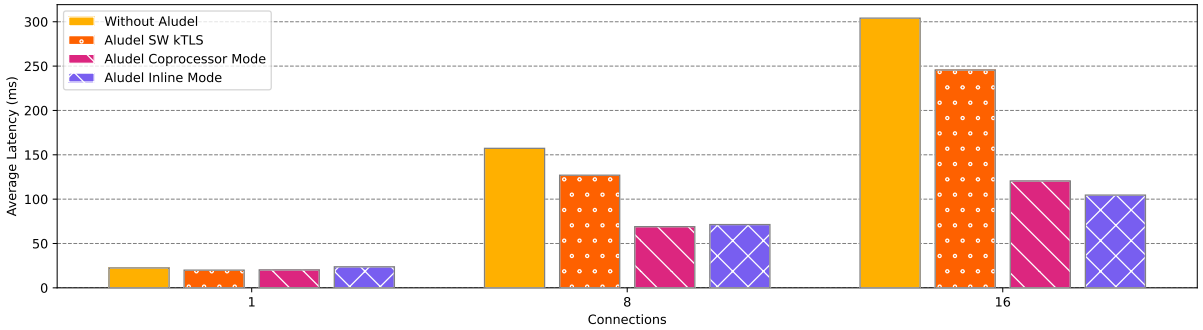

Figure 20 – Latency values of Aludel Cache within bare-metal setups, analyzing the effect of different offloading modes on response times.

Power Consumption - And for power, we can get interesting insights that resemble

the results from Power Consumption in Exploratory Experiments for bare-metal. The offload method cases consumed way more power than when compared to the scenario without Aludel. These results can be explained as the target binary being the only executable running on the target server CPU core. At the same time, with Aludel, you have the Aludel binary and the target binary running, while in the Exploratory Experiments, the only binary running is the kTLS-compatible Web server.

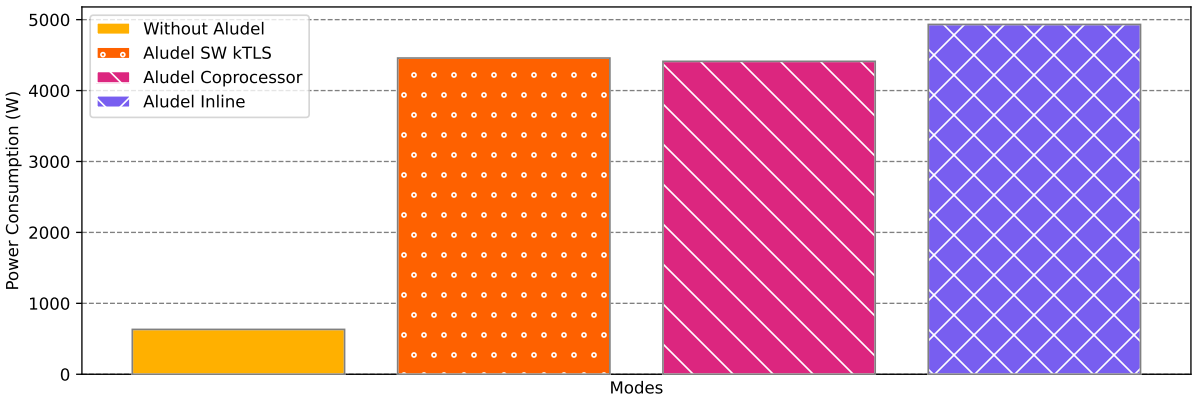

Figure 21 – Assessment of power consumption in Aludel Cache, performed in a bare-metal setting, emphasizing how different offloading modes influence power efficiency.

As we can see by the results, Aludel PoC v0.2 (Aludel Cache) performed way better than Aludel Proxy and could satisfy most of our premises, but mainly making a legacy application that uses an OpenSSL 1.0.2f version that is completely incapable of supporting kTLS to support it and even get the advantages of transparent hardware offload support, even to applications that would never find a way to be updated without their source code.

4.6 Observations

We managed to get Aludel to work and to be a middleware capable of providing kTLS support to closed-source applications. The results showed that a traffic forwarding-only solution could not take advantage of kTLS benefits acting in a reverse proxy mode. Still, they showed good numbers for a scenario using a cache where Aludel could access the file and use `sendfile()`. There is an important highlight to Inline mode: apart from being one of the most power inefficient, it was the more performant, so it also throws the same information from the Exploratory Experiments: Network Administrators should check on your strategy for offloading and make sure to choose the best options based on the resource efficiency requirements.

One of the limitations of this approach is that it is particularly focused on serving Web applications. We did not test any other applications other than HTTP. But we got interesting results, such as we could make legacy applications to support kTLS. We also

could check that the Aludel Proxy approach was not that interesting. Another conjecture is that when you already have an extra hop in the middle of the connections, making the Aludel Proxy an interesting approach as you already will face some communication degradation.

With what we have, we managed to mitigate one of the most objective flaws of kTLS, which is the difficulty of implementing it. But there is still room for improvement and that will be explained in Section 5.

5 Conclusions Remarks

5.1 Achievements

The work presented in Section 3 was proudly accepted at the IEEE NOMS 2024, under the title of Unlocking Security to the Board: An Evaluation of SmartNIC-driven TLS Acceleration with kTLS (NOVAIS; VERDI, 2024). Receiving good reviews from the peer reviewing and confirming the impact of showing the intricacies of using offloading with the kTLS technology.

5.2 Conclusions

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. Our experiments with Aludel showcased its capability to provide kTLS support to closed-source applications, even those lacking access to their source code. By offering a streamlined integration process without extensive code modifications, Aludel addresses a critical gap, enabling closed-source applications to benefit from improved security and performance in the ever-evolving landscape of network communications.

To further enhance the practicality of kTLS integration, future efforts should prioritize the development of mechanisms and compatibility layers that seamlessly integrate kTLS, even in scenarios where the application's source code is inaccessible instead of recurring with a solution like Aludel. This will contribute significantly to the broader adoption of kTLS in real-world applications, bolstering network security across diverse landscapes. Additionally, as kTLS continues to evolve, vendors must extend support to emerging encryption standards like TLS 1.3 and evolving protocols such as QUIC, ensuring the technology's continued relevance in contemporary networking scenarios.

What Aludel could achieve represents an indicator that there is space for improvements to provide better ease of usage for the kTLS, aiming for bringing compatibility to a wider range of applications and not only the ones that follow the constraints for the actual kTLS implementation.

5.3 Future Work

Exploring innovative methods to make kTLS more accessible to a broader range of network software presents an ongoing challenge. Subsequent research may focus on implementing better solutions than Aludel, maybe upgrading the Reverse Proxy architecture to use low-level packet forwarding like sockmap (SITNICKI, 2023), or eliminating the middleware from the equation by modifying the closed-source binary with techniques like binary patching or runtime patching to do kTLS integration across various application landscapes, ensuring its advantages are accessible to a broader community of users.

The experiment code, which was distributed as open source, could also be used to test other boards that support kTLS. The results will probably differ, and most of the other boards do not seem to have an Inline mode, so they may not be directly comparable. Also, as the technologies evolve, it is important to check if other cryptographic backends will try to bring compatibility to the kTLS. An example is that a lot of cloud-native technologies rely on cryptographic backends like BoringSSL and MBedTLS, making it interesting to compare if different cryptographic backends could bring different improvements in performance that would justify solutions like Aludel to evolve to increase kTLS adoption.

Bibliography

BALDWIN, J. Tls offload in the kernel. *FreeBSD Journal*, maio/jun. 2020. Cited 2 times on pages 46 and 54.

BATES, A. et al. Securing ssl certificate verification through dynamic linking. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2014. (CCS '14), p. 394–405. ISBN 9781450329576. Disponível em: <https://doi.org/10.1145/2660267.2660338>. Cited in page 50.

BISHOP, M. *HTTP/3*. RFC Editor, 2022. RFC 9114. (Request for Comments, 9114). Disponível em: <https://www.rfc-editor.org/info/rfc9114.> Cited in page 25.

CITRON, J. *Discord | Your Place to Talk and Hang Out*. 2024. Disponível em: <https://discord.com/>. Cited in page 39.

cURL Docs. *cURL Docs - Rate limiting*. 2024. Disponível em: <https://everything.curl.dev/usingcurl/transfers/rate-limiting>. Cited in page 35.

DOCKER. *Docker overview*. 2024. Disponível em: <https://docs.docker.com/get-started/overview/>. Cited in page 35.

DUROV, N.; DUROV, P. *Telegram Messenger*. 2024. Disponível em: <https://telegram.org/>. Cited in page 39.

ERAN, H. et al. *NICA: An Infrastructure for Inline Acceleration of Network Applications*. USENIX Association, 2019. ISBN 978-1-939133-03-8. Disponível em: <https://www.usenix.org/conference/atc19/presentation/eran.> Cited in page 31.

FORSTER, F. *collectd – The system statistics collection daemon*. 2024. Disponível em: <https://collectd.org/>. Cited in page 36.

FREITAS, E. et al. A survey on accelerating technologies for fast network packet processing in linux environments. *Comput. Commun.*, Elsevier Science Publishers B. V., NLD, v. 196, n. C, p. 148–166, dec 2022. ISSN 0140-3664. Disponível em: <https://doi.org/10.1016/j.comcom.2022.10.003.> Cited in page 41.

GLOZER, W. *wrk - a HTTP benchmarking tool*. 2024. Disponível em: <https://github.com/wg/wrk>. Cited in page 36.

GRANT, S. et al. 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*. New York, NY, USA: Association for Computing Machinery, 2020. (SIGCOMM '20), p. 681–693. ISBN 9781450379557. Disponível em: <https://doi.org/10.1145/3387514.3405895.> Cited in page 25.

HUNTER, J. *Matplotlib: Visualization with Python*. 2024. Disponível em: <https://matplotlib.org/>. Cited in page 38.

IYENGAR, J.; THOMSON, M. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC Editor, 2021. RFC 9000. (Request for Comments, 9000). Disponível em: <https://www.rfc-editor.org/info/rfc9000.> Cited in page 46.

KATSIKAS, G. et al. What you need to know about (smart) network interface cards. In: *International Conference on Passive and Active Network Measurement*. [S.l.: s.n.], 2021. p. 319–336. ISBN 978-3-030-72581-5. Cited in page 30.

Linux Kernel Organization. *Kernel TLS Offload - The Linux Kernel documentation*. 2024. Disponível em: <https://docs.kernel.org/networking/tls-offload.html>. Cited 2 times on pages 25 and 30.

LIU, J. et al. Performance characteristics of the bluefield-2 smartnic. *CoRR*, abs/2105.06619, 2021. Disponível em: <https://arxiv.org/abs/2105.06619.> Cited in page 30.

LIU, M. et al. Offloading distributed applications onto smartnics using ipipe. In: *SIGCOMM 2019 - Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication*. [S.l.]: Association for Computing Machinery, Inc, 2019. p. 318–333. ISBN 9781450359566. Cited in page 31.

LIU, M. et al. *E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers*. USENIX Association, 2019. ISBN 978-1-939133-03-8. Disponível em: <https://www.usenix.org/conference/atc19/presentation/liu-ming.> Cited in page 31.

MCKINNEY, W. *pandas - Python Data Analysis Library*. 2024. Disponível em: <https://pandas.pydata.org/>. Cited in page 38.

NEUGEBAUER, R. et al. Understanding pcie performance for end host networking. In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. New York, NY, USA: Association for Computing Machinery, 2018. (SIGCOMM '18), p. 327–341. ISBN 9781450355674. Disponível em: <https://doi.org/10.1145/3230543.3230560.> Cited in page 25.

NICHOLS, D. *Coloring for Colorblindness*. 2024. Disponível em: <https://davidmathlogic.com/colorblind/>. Cited in page 38.

NOVAIS, F.; VERDI, F. Unlocking Security to the Board: An Evaluation of SmartNIC-driven TLS Acceleration with kTLS. In: *IEEE/IFIP Network Operations and Management Symposium (NOMS 2024)*. Seoul, South Korea: IEEE, 2024. p. 6–10. Disponível em: <https://leris.dcomp.ufscar.br/wp-content/uploads/2024/01/IEEE_NOMS_2024___Unlocking_Security_to_the_Board___An_Evaluation_of_SmartNIC_driven_TLS_Acceleration_with_kTLS-1.pdf>. Cited 2 times on pages 27 and 61.

OETIKER, T. *RDDTool*. 2024. Disponível em: <https://oss.oetiker.ch/rrdtool/>. Cited in page 38.

OLIPHANT, T. *Numpy*. 2024. Disponível em: <https://numpy.org/>. Cited in page 38.

PAAR, C.; PELZL, J. *Understanding Cryptography - A Textbook for Students and Practitioners*. [S.l.]: Springer, 2010. I-XVIII, 1–372 p. ISBN 978-3-642-04100-6. Cited in page 29.

PEREZ, F.; GRANGER, B. *Jupyter*. 2024. Disponível em: <https://jupyter.org/>. Cited in page 38.

PISMENNY, B.; LESOKHIN, I.; LISS, L. Offload to network devices-rx offload. *Netdev 2.2*, 2017. Cited in page 25.

PISMENNY, B. et al. Tls offload to network devices. *Netdev 1.2*, 2016. Cited in page 32.

POETTERING, L. *systemd — Linux manual page*. 2024. Disponível em: <https://man7.org/linux/man-pages/man1/systemd.1.html>. Cited in page 37.

Red Hat. *Podman*. 2024. Disponível em: <https://podman.io/>. Cited in page 39.

RESCORLA, E. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC Editor, 2018. RFC 8446. (Request for Comments, 8446). Disponível em: <https://www.rfc-editor.org/info/rfc8446.> Cited 2 times on pages 29 and 46.

ROTT, J. K. *Intel® Advanced Encryption Standard Instructions (AES-NI)*. 2012. Disponível em: <https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html>. Cited in page 26.

ROULIN, A. *Advancing the State of Network Switch ASIC Offloading in the Linux Kernel*. Dissertação (Mestrado) — École Polytechnique Fédérale de Lausanne, 2018. Disponível em: <http://infoscience.epfl.ch/record/254829.> Cited in page 29.

SHEHABI, A. et al. *United States Data Center Energy Usage Report*. [S.l.], 2016. Cited in page 42.

SITNICKI, J. Speedrun through splicing sockets with sockmap. In: . Dublin: USENIX Association, 2023. Cited 2 times on pages 51 and 62.

STENBERG, D. 2023. Disponível em: <https://daniel.haxx.se/media/curl-user-survey-2023-analysis.pdf>. Cited in page 47.

STENBERG, D. *cURL - command line tool and library for transferring data with URLs (since 1998)*. 2024. Disponível em: <https://curl.se/>. Cited in page 36.

STEWART, R. R.; GURNEY, J. M. Optimizing tls for high–bandwidth applications in freebsd. *AsiaBSDCon 2015*, 2015. Cited in page 25.

SURESH, L. et al. Distributed resource management across process boundaries. In: *Proceedings of the 2017 Symposium on Cloud Computing*. New York, NY, USA: Association for Computing Machinery, 2017. (SoCC '17), p. 611–623. ISBN 9781450350280. Disponível em: <https://doi.org/10.1145/3127479.3132020.> Cited in page 25.

TSUJIKAWA, T. *ngtcp2*. 2024. Disponível em: <https://github.com/ngtcp2/ngtcp2>. Cited in page 46.

WASKOM, M. *Seaborn: Statistical Data Visualization*. 2024. Disponível em: <https://seaborn.pydata.org/>. Cited in page 38.

WATSON, D. Ktls: Linux kernel transport layer security. *Netdev 1.2*, 2016. Cited in page 25.

Web Technology Survey. *Web Technology Survey - OpenSSL Version Distribution*. 2024. Disponível em: <https://webtechsurvey.com/technology/openssl/versions>. Cited 3 times on pages 27, 47, and 52.

XING, T. et al. Towards portable end-to-end network performance characterization of smartnics. In: *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*. New York, NY, USA: Association for Computing Machinery, 2022. (APSys '22), p. 46–52. ISBN 9781450394413. Disponível em: <https://doi.org/10.1145/3546591.3547528.> Cited in page 29.

YANG, X. et al. Making quic quicker with nic offload. In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. New York, NY, USA: Association for Computing Machinery, 2020. (EPIQ '20), p. 21–27. ISBN 9781450380478. Disponível em: <https://doi.org/10.1145/3405796.3405827.> Cited in page 46.

ZHAO, J.; NEVES, M.; HAQUE, I. On the (dis)advantages of programmable nics for network security services. In: *2023 IFIP Networking Conference (IFIP Networking)*. [S.l.: s.n.], 2023. p. 1–9. Cited in page 32.