POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Toward High-Speed Tunneling Technologies: A New WireGuard Parallel Implementation for Linear Throughput Scaling

Supervisors

Prof. Fulvio RISSO

Ph.D. Student Federico PAROLA

Ph.D. Student Davide MIOLA

Candidate

Mirco BARONE

July 2024

Summary

Multi-cloud solutions offer a potential avenue for reducing the dominance of current hyperscalers. In this context, multiple clusters can be established across different providers or in various geographical locations and appropriately interconnected to enable interoperability. However, existing solutions aimed at achieving this goal (such as Submariner.io and Liqo.io) yield unsatisfactory results regarding network bandwidth. For instance, when technologies like Wireguard are employed, the maximum speed between two clusters is a few gigabits per second.

Wireguard, one of the most commonly used tunneling technologies in Linux, is known for its simplicity and excellent integration with the Linux kernel. Despite its widespread adoption, it struggles to provide high-speed connectivity between two sites when using a standard single-tunnel configuration. This poses a significant limitation when a secure, high-speed interconnection is required. Indeed, the ability to scale Wireguard performance with the number of available CPU cores is limited, even with an intrinsically parallel software architecture.

The primary aim of this thesis is to investigate the main features of Wireguard and the state of the art of the current solutions developed to enhance its throughput. Secondly, it aims to identify existing limitations and propose an improved architecture that facilitates effective scaling, achieving a nearly linear throughput increase depending on the number of involved CPU cores.

The thesis examines the architecture of a single tunnel setup, highlighting how, despite its ability to parallelize encryption and decryption stages, the presence of serial per-tunnel stages still limits the use of additional resources. It then shifts focus to a multi-tunnel architecture. However, the analysis reveals that merely leveraging multiple tunnels can result in no scaling at all due to a subtle "black hole" condition related to the NAPI poll functions when using the standard softirq-based NAPI. This limitation can be overcome by enabling the threaded NAPI on Wireguard interfaces. However, despite leveraging all our nodes' resources, this approach still shows far from linear performance improvement when increasing the number of allocated cores.

To further enhance performance, a modified architecture is proposed. This architecture handles all Wireguard stages inline for each flow in a signal processing context on a single core, eliminating the costs of task and cache synchronization. This improved architecture, tailored for multi-tunnel support, demonstrates almost a 2x performance improvement over a multi-tunnel deployment based on the vanilla Wireguard implementation and is

capable of supporting 18x the throughput of a single tunnel setup on our machines.

This approach is not a one-size-fits-all solution. Its main limitation currently lies in the inability to parallelize the encryption/decryption stages for a single flow, which could potentially penalize elephant flows. However, it provides an exciting starting point for further discussion and represents a first step towards a more scalable Wireguard architecture.

Table of Contents

List of Figures VII

1 Introduction 1
- 1.1 Main Problem, Goals and Prosed Solution . . . 1
- 1.2 Thesis Structure . . . 2

2 Background 3
- 2.1 Kubernetes Overview . . . 3
  - 2.1.1 Internal Architecture Overview . . . 4
- 2.2 Details About VPNs . . . 5
- 2.3 Liqo Overview . . . 6
- 2.4 Submariner Overview . . . 7
- 2.5 Related Works . . . 9

3 Linux networking 11
- 3.1 The path of a packet . . . 11
  - 3.1.1 Kernel and interrupts . . . 11
  - 3.1.2 Top-half and Bottom-Half handlers . . . 12
  - 3.1.3 The socket buffer . . . 13
  - 3.1.4 Receiving packets and RSS . . . 15
  - 3.1.5 NAPI . . . 16
  - 3.1.6 Driver poll method . . . 17
  - 3.1.7 Function `netif_receive_skb` . . . 18
  - 3.1.8 Transmission Path . . . 18
  - 3.1.9 Conclusions . . . 20

4 Wireguard 22
- 4.1 Overview and Terminology . . . 22
  - 4.1.1 Main features . . . 22
  - 4.1.2 Inner details and terminology . . . 23
  - 4.1.3 Versions . . . 25
  - 4.1.4 Current Adoption . . . 26
- 4.2 Internal Architecture . . . 26

4.2.1 Encapsulation . . . 26
4.2.2 Encapsulation Code . . . 28
4.2.3 Decapsulation . . . 29
4.2.4 Decapsulation Code . . . 30
4.3 Alternatives . . . 31
4.3.1 Ipsec . . . 31
4.3.2 OpenVpn . . . 33
4.4 Perfomance . . . 33

5 Known WireGuard Optimization Techniques 35
5.1 The role of MTU . . . 35
5.1.1 What is MTU . . . 35
5.1.2 The Impact of changing MTU in Wireguard Performance . . . 36
5.2 Offloading Techniques . . . 38
5.2.1 Introduction and Main Purpose . . . 38
5.2.2 TSO,GSO and LSO . . . 38
5.2.3 GRO and LRO . . . 41
5.2.4 Offloading Techniques in UDP . . . 42
5.2.5 Improving Wireguard with offloading . . . 42

6 Wireguard Evaluation 46
6.1 Testbed . . . 46
6.2 Single Tunnel Evaluation . . . 47
6.3 Multiple Tunnels Evaluation . . . 48
6.3.1 Results and the problem of the Napi . . . 49
6.3.2 Results with Threaded Napi . . . 51
6.3.3 Final Considerations . . . 52

7 Wireguard Inline 53
7.1 Limitations and Overhead of Wireguard . . . 53
7.2 Idea and Main Changes . . . 53
7.3 Wireguard Inline Code . . . 54
7.3.1 Transmission Path . . . 54
7.3.2 Receive Path . . . 56
7.4 Peformance Evaluation . . . 57
7.4.1 Efficiency Measure . . . 59

8 Conclusions and Future Work 61
8.1 Current Limitation . . . 62
8.2 Future Work . . . 62

Bibliography 65

List of Figures

2.1 High level vision of the network fabric established between two clusters . . 7
2.2 Submariner internal architecture 8
2.3 AWS Transit Gateway architecture with multiple tunnels 10

3.1 Top-half and bottom-half handler in Linux Kernel 12
3.2 Socket buffer internal structure 14
3.3 Packet reception in the Linux kernel 15

4.1 WireGuard processing on CPU cores on the encapsulation side in the single tunnel scenario. 27
4.2 WireGuard processing distribution on CPU cores on the decapsulation side in the single tunnel scenario. 30
4.3 AH in transport and tunnel mode 32
4.4 ESP in transport and tunnel mode 33

5.1 MTU and MSS 36
5.2 Performance of Wireguard increasing MTU 37
5.3 TSO 39
5.4 GSO 40
5.5 GRO 41
5.6 WireGuard-Go's Transmission Path Architecture including the improvement introduced by the Tailscale team 44

6.1 Testbed 46
6.2 Aggregate throughput and CPU usage for an increasing number of TCP flows in a single tunnel setup. 47
6.3 Per-core CPU usage on GW1 and GW2 in the single tunnel setup when handling 32 flows. 48
6.4 Aggregate throughput and CPU usage for an increasing number of TCP flows and corresponding WireGuard tunnels. 49
6.5 Per-core CPU usage on *gw2* when handling 8 flows spread over 8 tunnels. . 51
6.6 Aggregated Throughput and CPU usage on *gw1* and *gw2* comparing Standard NAPI and Threaded NAPI 52

7.1 WireGuard Inline processing distribution on CPU cores in a 2 tunnels setup (note that, even if not represented here, a single core might encapsulate multiple tunnels). 54
7.2 Comparison of CPU Usage and Throughput for WireGuard Inline and WireGuard Vanilla 58
7.3 CPU Usage per Core in GW1 with 16 Tunnels in Use 59
7.4 CPU Usage per Core in GW2 with 16 Tunnels in Use 59
7.5 60

8.1 Overhead and MTU size when using GRETAP interface over Wireguard interface 63

Chapter 1

Introduction

1.1 Main Problem, Goals and Prosed Solution

This thesis tackles a significant issue that emerges when two remote clusters are interconnected using technologies such as Liqo and Submariner - the challenge of inadequate throughput. These technologies depend on Wireguard, a simple and recent VPN mechanism, for their infrastructure. However, the limitations of Wireguard hinder it from achieving high throughput.

Attaining substantial throughput is a critical objective as it enables the reduction of the role played by hyperscalers. This is achieved by facilitating the interconnection of remote clusters developed using different technologies or on the networks of various hyperscalers. Indeed, multi-cloud solutions present a potential avenue to diminish the dominant position of current hyperscalers by distributing workloads across multiple cloud providers.

The initial aim is to achieve a throughput slightly less than a 40 Gigabit/s line-rate, with the subsequent goal of approaching a 100 Gigabit/s line-rate.

The thesis commences by analyzing the characteristics of Wireguard, pinpointing its limitations and bottlenecks within the architecture, and exploring known methods in literature for enhancing its throughput. Specifically, two methods are identified: altering the Maximum Transmission Unit (MTU) and applying Generic Segmentation Offload (GSO) and Generic Receive Offload (GRO) to the physical interface where Wireguard packets are dispatched. These methods are not part of the final solution due to several issues that arise when attempting to apply them. However, they remain viable options for enhancing the throughput of a single tunnel and can be combined with the solution proposed in the remainder of the work.

Parallelism is identified as the optimal technique to boost throughput. Consequently, the thesis proceeds by parallelizing Wireguard and addressing various issues related to this approach, allowing for a throughput near 50 Gbps on a 100G line.

In the final section, a novel architecture is described and evaluated. This architecture, optimized for parallelism, eliminates all the overhead of vanilla Wireguard. The processing of packets occurs in the same core they are received, within the same system call or soft

interrupt request. Utilizing this architecture allows for a throughput near 90 Gbps, almost reaching the initial goal.

The solution developed is merely a starting point. Various issues must be addressed, and numerous future opportunities to further enhance the speed remain open.

1.2 Thesis Structure

- **Chapter 2:** This chapter provides background information on Kubernetes and Intercluster Connectivity. It offers a brief overview of solutions to interconnect clusters, such as Liqo or Submariner. Additionally, it includes a section dedicated to VPNs and another section discussing existing works similar to the thesis.
- **Chapter 3:** This chapter presents an overview of the Linux networking stack, focusing on the lower levels of the stack. It describes important concepts like NAPI, RSS, and Interrupt, along with the journey of a packet in the stack.
- **Chapter 4:** This chapter is centered on Wireguard. It initially presents a general overview of its advantages and features. The inner architecture is then described, highlighting potential bottlenecks. The final part of the chapter is dedicated to alternative VPNs, such as Ipsec and OpenVPN.
- **Chapter 5:** This chapter introduces two techniques to enhance Wireguard performance. Firstly, it describes the effects of increasing the MTU value in a general context and specifically in Wireguard, studying its feasibility. The second part provides an overview of offloading techniques, describing which are currently applied and which could be applied to Wireguard in the future.
- **Chapter 6:** This chapter evaluates Wireguard performance. Initially, the performance of a single tunnel is tested. Then, the impact of parallelism on Wireguard is studied, highlighting problems (e.g., the Napi problem) and potential solutions.
- **Chapter 7:** This chapter delves into the changes made in the inline architecture of Wireguard. It describes the rationale behind these changes and the code modifications. The final part contains an evaluation comparing the results with the standard Wireguard version.
- **Chapter 8:** This chapter focuses on the problems of Wireguard inline and discusses future work.

Chapter 2

Background

2.1 Kubernetes Overview

Kubernetes [1] is an open-source platform designed to automate deploying, scaling, and operating application containers. It groups containers that make up an application into logical units for easy management and discovery.

The evolution of deployment methodologies has played a significant role in Kubernetes' creation and its popularity:

- **Traditional Deployment Era:** Initially, applications ran on physical servers. However, resource allocation issues arose when multiple applications ran on a single server. The solution was to run each application on a different physical server, but this was not cost-effective and led to underutilization of resources.

- **Virtualized Deployment Era:** Virtualization was introduced to allow multiple Virtual Machines (VMs) on a single physical server's CPU. This method improved resource utilization and provided better scalability and security.

- **Container Deployment Era:** Containers, similar to VMs but lighter, share the OS among applications. They are portable across clouds and OS distributions, providing benefits like environmental consistency, resource isolation, and high efficiency.

Kubernetes was developed to manage these containers in a production environment. It provides a framework to run distributed systems resiliently, taking care of scaling and failover for applications, among other things. Some key features of Kubernetes include:

- Service Discovery and Load Balancing: Kubernetes can expose a container using the DNS name or their own IP address. If traffic to a container is high, Kubernetes can load balance and distribute the network traffic to keep the deployment stable.

- Storage Orchestration: Kubernetes permits to automatically mount a storage system.

- Automated Rollouts and Rollbacks: It is possible to describe the desired state for a deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate.

- Self-Healing: Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to user-defined health check, and doesn't advertise them to clients until they are ready to serve.

- Secret and Configuration Management: Kubernetes permits to store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys.

- Horizontal Scaling: It is possible to scale an application up and down with a simple command, with a UI, or automatically based on CPU usage.

- Designed for Extensibility: Every user can add features to a Kubernetes cluster without changing upstream source code.

In addition to these, Kubernetes also supports IPv4/IPv6 dual-stack for the allocation of IPv4 and IPv6 addresses to Pods and Services.

Kubernetes has a large, rapidly growing ecosystem and services, support, and tools are widely available. This makes it a reliable choice for organizations looking to implement containerized applications. It's worth noting that Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system. It preserves user choice and flexibility where it is important and adds powerful primitives for building distributed systems.

2.1.1 Internal Architecture Overview

A Kubernetes cluster is a set of machines, known as nodes, that run containerized applications. These nodes host the Pods, which are the smallest and simplest units in the Kubernetes object model that is possible to create or deploy. A Pod represents a running process on the cluster and can contain one or more containers.

Every Kubernetes cluster has at least one worker node. The worker node(s) run the Pods, and thus, the components of the application workloads. The worker nodes receive instructions from the control plane, which manages the worker nodes and the Pods in the cluster.

The control plane consists of various components, including the Kubernetes Master, etcd, kubelet, kube-proxy, and others. The control plane's role is to make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (starting up a new pod when a deployment's replicas field is unsatisfied).

In production environments, the control plane usually runs across multiple computers for fault tolerance, and a cluster usually runs multiple nodes, providing high availability.

2.2 Details About VPNs

A *Virtual Private Network* (VPN) is a technique used to extend a private network (any network that is not the public Internet) across one or more other networks. These networks could be untrusted (not controlled by the entity implementing the VPN) or need to be isolated (making the underlying network invisible or unusable).

A VPN can provide access to a private network (which restricts or denies public access to some of its resources) to users who lack direct access, such as an office network that permits secure off-site access via the Internet. This is accomplished by establishing a connection between computing devices and computer networks using network tunneling protocols.

A VPN can be made secure over an insecure communication medium (like the public internet) by selecting a tunneling protocol that incorporates the necessary security features to ensure confidentiality and integrity. Such VPN implementations offer the advantage of lower costs and increased flexibility compared to dedicated communication lines, particularly for remote workers.

The operation of a VPN depends on the technologies and protocols it is built upon. A tunneling protocol is employed to transport network messages from one end to the other. The objective is to take network messages from applications (operating at OSI layer 7) on one side of the tunnel and replay them on the other side, effectively replacing the underlying network or link layers. Applications don't need to be altered to allow their messages to pass through the VPN, as the virtual network or link is made available to the OS.

The configurations of Virtual Private Networks can be categorized based on the objective of the virtual extension, which determines the appropriate tunneling strategies for different topologies:

- **Remote Access:**
  A host-to-network configuration is similar to connecting one or more computers to a network to which they cannot be directly connected. This type of extension grants that computer access to the local area network of a remote site, or broader enterprise networks, such as an intranet. Each computer is responsible for activating its own tunnel to the network it wishes to join. The joined network only recognizes a single remote host for each tunnel. This can be used for remote workers, or to allow individuals to access their private home or company resources without exposing them to the public Internet.

- **Site-to-site:**
  A site-to-site configuration connects two networks. This configuration extends a network across geographically separated locations. Tunneling is only performed between two devices (like routers, firewalls, VPN gateways, servers, etc.) situated at both network locations. These devices then make the tunnel accessible to other local network hosts that aim to reach any host on the other side. This is beneficial for maintaining stable connections between sites, like office networks to their headquarters

or datacenter. In this case, either side can be configured to initiate communication as long as it knows how to reach the other on the medium network. If both are known to each other, and the chosen VPN protocol is not restricted to a client-server design, the communication can be initiated by either of the two as soon as they notice the VPN is inactive or a local host is trying to reach another one known to be located on the other side.

2.3 Liqo Overview

Liqo [2] [3] is an open-source project that facilitates the creation of dynamic and seamless multi-cluster topologies in Kubernetes, accommodating heterogeneous infrastructures across on-premise, cloud, and edge environments. Liqo offers several features:

- **Peering:** In the context of Liqo, peering is defined as a one-way relationship between two Kubernetes clusters where resources and services are consumed. One cluster (the consumer) is given the ability to offload tasks to another cluster (the provider), but not the other way around. The consumer initiates an outgoing peering towards the provider, which is then subject to an incoming peering from the consumer. This setup provides maximum flexibility in asymmetric configurations, while also supporting bidirectional peerings through their combination. Furthermore, a single cluster can act as both a provider and consumer in multiple peerings.

- **Offloading:** Workload offloading is facilitated by a virtual node that is created in the local (consumer) cluster at the conclusion of the peering process. This node represents and aggregates the resources shared by the remote cluster. This method allows for the transparent expansion of the local cluster, with the new node and its capabilities seamlessly integrated into the standard Kubernetes scheduler for optimal workload execution. This approach is fully compatible with standard Kubernetes APIs, allowing for interaction with and inspection of offloaded pods as if they were running locally.

- **Liqo Network:** The network fabric is a subsystem of Liqo that transparently extends the Kubernetes network model across multiple independent clusters. This allows offloaded pods to communicate with each other as if they were all running locally. Specifically, the network fabric ensures that all pods in a given cluster can communicate with all pods in all remote peered clusters, with or without NAT translation. Given the support for arbitrary clusters with different parameters and components (e.g., CNI plugins), it is not possible to guarantee non-overlapping pod IP address ranges (i.e., PodCIDR). This may necessitate address translation mechanisms, although NAT-less communication is preferred when address ranges are disjoint.

  The network manager serves as the control plane of the Liqo network fabric. It runs as a pod and is responsible for negotiating connection parameters with each remote cluster during the peering process. It includes an IP Address Management

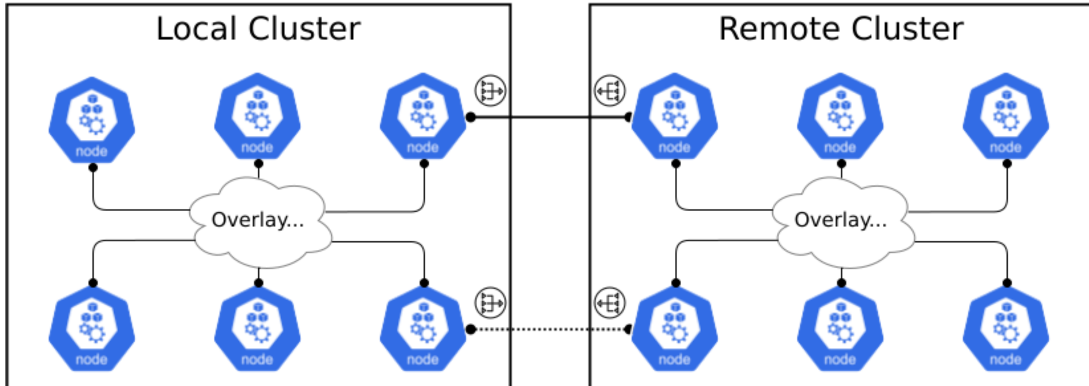

Figure 2.1: High level vision of the network fabric established between two clusters

(IPAM) plugin, which manages potential network conflicts through the establishment of high-level NAT rules (enforced by the data plane components).

The connection between peered clusters is established through secure VPN tunnels, created with WireGuard(further detail in Section 4), which are dynamically set up at the end of the peering process based on the negotiated parameters. The Liqo gateway, a component of the network fabric, sets up these tunnels. It runs as a privileged pod on one of the cluster nodes, populates the routing table appropriately, and configures the NAT rules required to resolve address conflicts using iptables.

The overlay network is used to route all traffic originating from local pods/nodes and directed to a remote cluster to the gateway, where it enters the VPN tunnel. The same process occurs on the other side, with traffic exiting the VPN tunnel entering the overlay network to reach the node hosting the destination pod.

Liqo employs a VXLAN-based setup, configured by a network fabric component that runs on all physical nodes of the cluster. This component is also responsible for populating the appropriate routing entries to ensure correct traffic forwarding.

- **Storage Fabric:** The Liqo storage fabric subsystem facilitates the seamless offloading of stateful workloads to remote clusters.

2.4 Submariner Overview

Submariner[4] is a comprehensive system composed of several key components that collaboratively ensure secure connectivity across multiple Kubernetes clusters. It establishes a secure and efficient connection between multiple Kubernetes clusters, flattening the networks between them and enabling IP accessibility between Pods and Services.

The main components of Submariner, which ensure secure connectivity across multiple Kubernetes clusters, whether on-premises or on public clouds, include:

- **Gateway Engine**: This component manages the secure tunnels to other clusters.

- **Route Agent**: This agent directs cross-cluster traffic from nodes to the active Gateway Engine.
- **Broker**: This component facilitates the exchange of metadata between Gateway Engines, enabling them to discover each other.
- **Service Discovery**: This feature provides DNS discovery of Services across clusters.

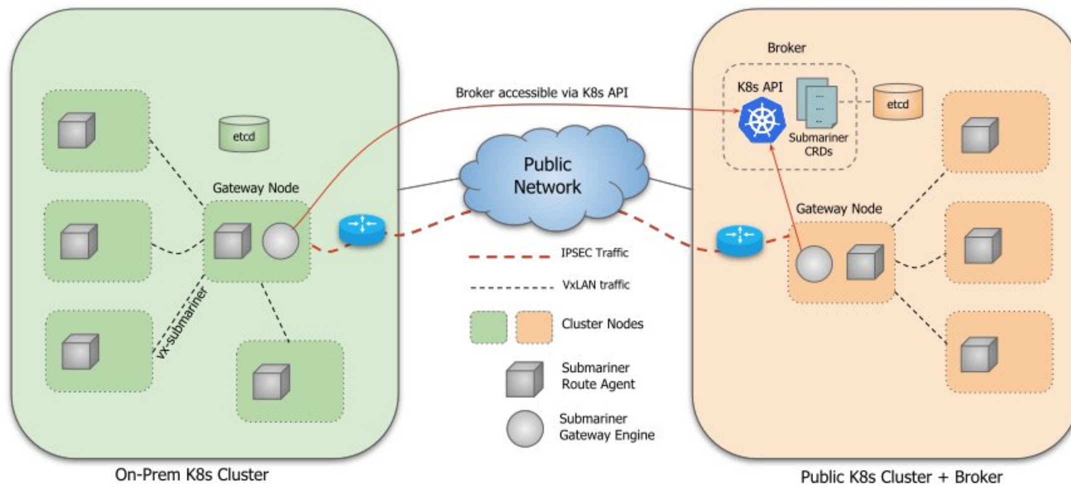

Figure 2.2: Submariner internal architecture

The Gateway Engine is deployed in each participating cluster and is tasked with establishing secure tunnels to other clusters. The Gateway Engine features a pluggable architecture for the cable engine component that maintains the tunnels. The available implementations include:

- An IPsec implementation using Libreswan, which is currently the default.
- An implementation for WireGuard, via the wgctrl library (further details in Section 4)
- An unencrypted tunnel implementation using VXLAN.

VXLAN connections are intentionally unencrypted. This is typically beneficial for environments where all participating clusters run on-premises, the underlying network fabric is controlled, and in many cases, already encrypted by other means. Another common use case is to leverage the VXLAN cable engine over a virtual network peering on public clouds (e.g., VPC Peering on AWS). In this scenario, the VXLAN connection is established over a peering link provided by the underlying cloud infrastructure, which is already secured. In both cases, the expectation is that connected clusters should be directly reachable without NAT.

Instances of the Gateway Engine run on specifically designated nodes in a cluster, and there may be more than one for fault tolerance. Submariner supports active/passive High Availability for the Gateway Engine component, meaning that there is only one active Gateway Engine instance at a time in a cluster. A leader election process is performed to determine the active instance, and the others remain in standby mode, ready to take over should the active instance fail.

2.5 Related Works

This thesis is dedicated to evaluating the constraints of WireGuard, with a particular emphasis on its scalability issues. While there are existing studies that suggest parallel architectures to mitigate these scalability challenges associated with tunnels, none have yet proposed something similar to the single-core, Inline architecture applied to WireGuard in this study.

A comparable architecture has been present in IPsec since its inception (refer to Section 4.3.1 for more details). In fact, IPsec is inherently associated with a single-core architecture. However, WireGuard and IPsec have distinct architectural differences. The primary distinction is that IPsec encapsulates packets at the IP level, while WireGuard operates at a different level, encapsulating IP packets into UDP. Given their different applications and usability, this thesis has chosen to focus exclusively on WireGuard, which is also the primary solution used in Liqo.

The two solutions discussed in this section are primarily related to IPsec, not WireGuard:

In 2020, Amazon[5] implemented a strategy to enhance the throughput of AWS Site-to-Site VPN, which traditionally has a maximum limit of 1.25 Gbps per IPsec tunnel. This strategy involved the use of **AWS Transit Gateway**, a service that simplifies the process of connecting multiple Virtual Private Clouds (VPCs) and on-premises networks.

The crux of this strategy is the use of multiple IPsec tunnels and a routing method known as Equal-Cost Multi-Path (ECMP) routing (see section 8.2 for further details about ECMP).

In the context of AWS Transit Gateway, ECMP allows the distribution of network traffic across multiple VPN tunnels, effectively increasing the overall throughput beyond the 1.25 Gbps limit of a single tunnel. This is achieved by using a 5-tuple hash (consisting of the protocol number, source IP address, destination IP address, source port number, and destination port number) to direct packets along one of the available tunnels.

However, it's important to note that while ECMP can increase overall throughput, a single VPN tunnel still has a maximum throughput of 1.25 Gbps. This means that the throughput increase is achieved by distributing the load across multiple tunnels, rather than increasing the capacity of a single tunnel.

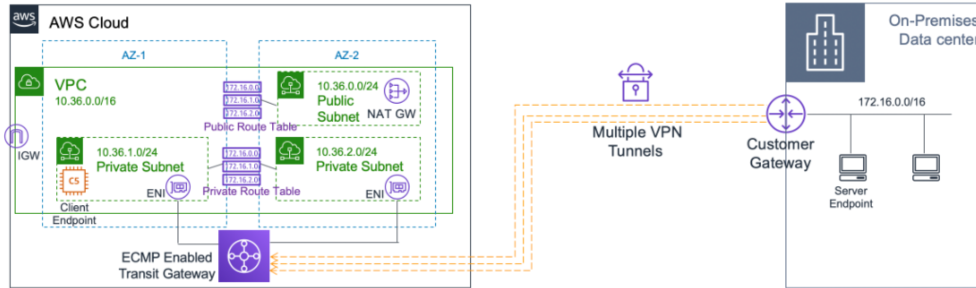

Figure 2.3: AWS Transit Gateway architecture with multiple tunnels

A similar solution, named **Aviatrix High Performance Encryption**[6, 7], was proposed and patented by Aviatrix in November 2023. This technique establishes multiple tunnels between two virtual routers, thereby allowing all CPU cores to be utilized for performance scaling with the CPU resources. This allows for surpassing the limit imposed by IPsec, which is only capable of using a single core.

Chapter 3

Linux networking

The Linux network stack is a component of the kernel responsible for managing and transferring network packets. This includes movement between the network interface and the application (and vice versa), as well as from one network interface to another (for bridging and forwarding purposes). This holds true irrespective of whether the interfaces are physical Network Interface Cards (NICs) or virtual devices that exist solely in software.

3.1 The path of a packet

This section aims to explain various networking concepts related to Linux kernel and to detail the journey of a packet within the stack, both when it is received and when it is transmitted.

3.1.1 Kernel and interrupts

The kernel is predominantly an event-driven software. This means that it reacts or is invoked when a specific set of events occur. Unlike many software systems, modern operating systems do not operate on an “event loop”. Instead, the kernel code is executed only when an interrupt is fired. In such instances, only the portion of the code that serves the interrupt is executed.

The processing of hardware interrupts is serialized. This means that while an Interrupt Service Routine (ISR) is being executed, other pending interrupts are enqueued and wait their turn. This process is crucial as it prevents conflicts and ensures smooth operation.

Interrupt Requests (IRQs) invariably trigger a context switch, which comes with an associated cost. A context switch refers to the process of storing and restoring the state (context) of a process or thread so that execution can be resumed from the same point at a later time. This allows multiple processes to share a single CPU. The context switch is a fundamental part of a multitasking operating system.

While user-to-system context switches are typically more prevalent, system-to-system context switches can also occur, particularly in the case of software interrupts.

Each interrupt is linked to an Interrupt Service Routine (ISR), a function that executes upon receipt of the interrupt. This necessitates a context switch from the currently executing code, whether it's user or system code. The cost of context switching can become prohibitive when the number of interrupts is large.

To manage this, techniques like interrupt coalescing are used. These techniques allow the system to delay interrupts and process multiple events at once, striking a balance between reaction time and overhead.

Under heavy load conditions, polling could be a more efficient option. Polling involves continuously checking if the device has any information to relay (for example, by reading the register of the device). However, this method can waste resources under low to medium load conditions.

A mixed approach is often the most effective. This involves adaptively switching between interrupt and polling modes according to the actual workload. This approach is used by the New API (NAPI) in the Linux kernel, which helps to improve the system's performance and efficiency.

3.1.2 Top-half and Bottom-Half handlers

The management of interrupts in the kernel is typically split into two steps: the **Top Half Handler** (`hardirq`) and the **Bottom Half Handler** (`softirq`).

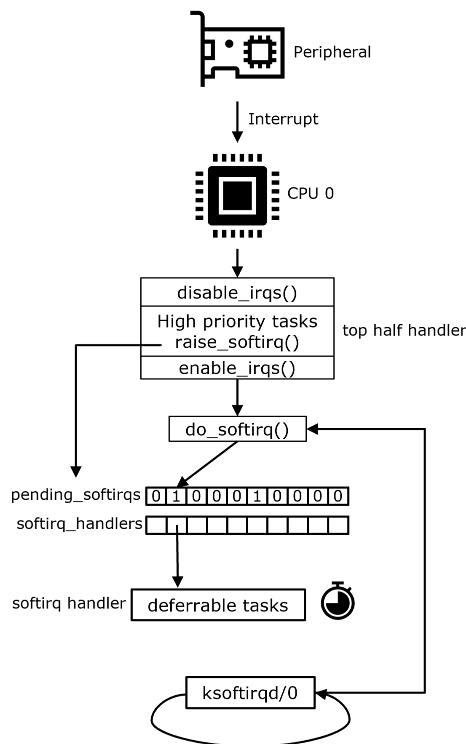

Figure 3.1: Top-half and bottom-half handler in Linux Kernel

The Top Half Handler is executed upon the reception of an interrupt. It performs operations that cannot be delayed, as any delay could potentially result in the loss of data. During the execution of the Top Half Handler, interrupts and preemption are disabled to ensure the integrity of the operations. This process needs to be as fast as possible to minimize the disruption to the system.

When a device is initialized, it registers one or more top half handlers with the `request_irq()` function. This registration process includes indicating the number of the interrupt and the handler function to execute when the interrupt is fired. The execution of the Top Half Handler is architecture-specific, usually a function named `do_IRQ()` retrieves the handler from a global table using the interrupt number and executes it.

The Bottom Half Handler, on the other hand, performs tasks that require more time but can be delayed. During its execution, interrupts are enabled, and preemption is disabled. This handler is implemented by `softirqs`. It's important to note that only one `softirq` of a certain type can be executed on a CPU at a time, but multiple `softirqs` of the same type can be executed concurrently on different CPUs. The execution of the Bottom Half Handler is scheduled on the local CPU with the `raise_softirq()` method, which takes the `softirq` type as a parameter. This method sets a bit in a global per-CPU bitmap to mark the `softirq` as pending. This is usually called in the Top Half Handler.

The softirq is executed asynchronously in different spots with the `do_softirq()` method. This method scans the bitmap of pending `softirqs` and executes the corresponding handlers.

It's interruptible but non-preemptible. Since interrupts are enabled, a `softirq` could be re-scheduled before handling has ended. To handle such cases, the method scans the bitmap multiple times.

To prevent the softirq from monopolizing the CPU, the scan is performed for a maximum of **MAX_SOFTIRQ_RESTART** times and for a maximum time of **MAX_SOFTIRQ_TIME**.

If `softirqs` are still pending after this, a low priority per-CPU kernel thread `ksoftirqd/n` is scheduled and will handle the remaining `softirqs`.

3.1.3 The socket buffer

The **socket buffer**, or `sk_buff`, is a data structure that maintains the data and metadata of a packet during its journey through the network stack. The primary objective of this structure is to provide an efficient and uniform way for any kernel component to access the packet, its metadata (such as the interface it was received from, timestamp when available), and the most useful fields (like IP addresses, TCP/UDP ports).

The problem is that the format in which the packet is saved by the NIC in memory depends on the NIC's architecture, and it usually varies across different cards. The kernel cannot parse data differently based on the format defined by the manufacturer. Furthermore, some metadata, such as hardware timestamp, may not be available in all NICs.

This necessitates a "data normalization" phase to store all frames in the same "canonical"

format. This process helps avoid different packet parsing in the kernel for frames coming from different link layers (for example, Ethernet vs WiFi) or with different characteristics (like VLAN-tagged frames).

The `sk_buff` also retains kernel internal information and the value of most useful fields to expedite the processing of subsequent modules.

Every socket buffer is composed of two entities as depicted in figure 3.3: the `sk_buff` data structure that stores the metadata of the packet, and a buffer where the data of the packet is stored. The head and end pointers of the `sk_buff` structure point to the limits of the buffer of the packet. The data and tail pointers delimit the packet data inside the buffer.

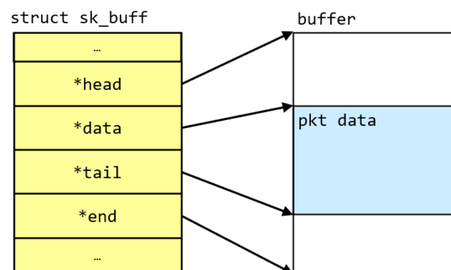

Figure 3.2: Socket buffer internal structure

The size of the buffer is larger than the packet itself. The data and tail can be moved inside the buffer to create or remove space when pushing and popping headers, without the need to copy the packet or allocate new memory.

Socket buffers are usually organized in circular lists to be queued and processed in batches. The next and prev pointers point to nearby elements in this list.

Socket buffer code

The `sk_buff` structure contains several fields, which are related to different network layers

```
struct sk_buff {
    ...
    struct net_device   *dev; // The input or output device
    ktime_t             tstamp;  // The timestamp of reception
    __be16              protocol;  // The L3 protocol of the packet
    __u16   transport_header; // Pointers to the various headers in the buffer
    __u16   network_header;
    __u16   mac_header;
    ...
}
```

3.1.4 Receiving packets and RSS

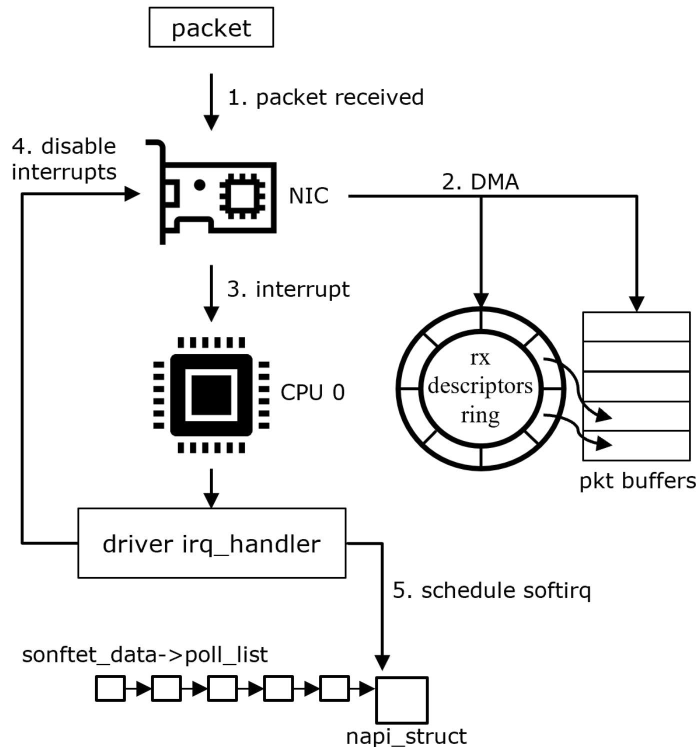

Figure 3.3: Packet reception in the Linux kernel

Upon initialization, the device driver allocates specific memory areas to be used for exchanging packets with the NIC. The structure of these buffers is unique to each driver, but they typically include a buffer area used to store packet data through Direct Memory Access (DMA), and a ring of descriptors used to store metadata about the packet.

This metadata includes the address of the packet data in the buffer area, its size, and additional metadata provided by the NIC, such as a timestamp (if hardware timestamping is supported and enabled) and the Receive Side Scaling (RSS) hash.

Modern NICs support multiple receive and transmit descriptor queues (multi-queue). This capability allows a NIC to distribute different packets to different queues, thereby distributing the processing load among multiple CPUs.

The NIC achieves this distribution by applying a filter to each incoming packet. This filter assigns each packet to one of a small number of logical flows. Packets assigned to each flow are then steered to a separate receive queue. Each of these queues can, in turn, be processed by a separate CPU. This mechanism is generally known as “Receive-side Scaling” (RSS) [8].

The primary goal of RSS and other similar scaling techniques is to increase performance uniformly across the system. The filter used in RSS is typically a hash function that operates over the network and/or transport layer headers. For example, a common

approach is to use a 4-tuple hash over the IP addresses and TCP ports of a packet.

Some advanced NICs take this a step further by allowing the steering of packets to queues based on programmable filters. This provides even greater flexibility and control over packet processing, enabling more sophisticated load balancing strategies.

If the NIC supports multiple hardware queues, a copy of these memory areas is made for each queue. Packets are then distributed through RSS. The NIC transfers one or more packets into the buffer using DMA and signals the presence of new data with an interrupt.

The hardware interrupt handler, which is registered by the driver, handles the interrupt raised by the NIC. It disables hardware interrupts for the device and schedules the execution of the softirq on the local CPU with the `__napi_schedule()` method.

3.1.5 NAPI

NAPI [9] is a key event handling mechanism used by the Linux networking stack. Despite its name, which once stood for "New API", it no longer represents any specific acronym.

In its basic operation, a device notifies the host about new events via an interrupt. Upon receiving this interrupt, the host schedules a NAPI instance to process these events. This is the primary mode of operation for NAPI, and it's designed to handle the majority of network events efficiently and effectively.

However, NAPI also supports an alternative mode of operation known as busy polling. In this mode, the device is polled for events via NAPI without waiting for an interrupt. This can be particularly useful in high-load situations where waiting for an interrupt could lead to significant delays.

NAPI processing typically occurs in the context of a software interrupt. This allows the system to handle network events at a low level, close to the hardware, which can help to reduce latency and improve performance.

However, there is also an option to use separate kernel threads for NAPI processing. This can be beneficial in certain situations, such as when dealing with a large number of simultaneous network events, or when the system has multiple CPUs that can be used to parallelize the processing of network events. This options was used for Wireguard in section 6.3.2

A structure of type `napi_struct` is used to schedule a packet processing task. This structure is initialized by the driver upon device initialization with the `netif_napi_add()` method. This method receives a driver-specific poll function that is bound to the NAPI structure and will be used to process packets in the `softirq`.

The `softnet_data` structure plays a crucial role in handling per-CPU receive (rx) and transmit (tx) software queues. This structure is allocated once for every CPU when the system starts up. It's designed to manage the packet processing tasks that need to be scheduled and executed.

One of the key fields in the `softnet_data` structure is `poll_list`. This field contains a queue of elements of type `napi_struct`.

Each `napi_struct` element represents a packet processing task that needs to be scheduled. This queue serves as a task list for the CPU, ensuring that packet processing tasks

are executed in an orderly and efficient manner.

The `__napi_schedule()` method is a critical part of this process. This method appends the `napi_struct` of the current device to the `poll_list` of the `softnet_data` structure of the local CPU. In other words, it adds the packet processing task associated with the current device to the task list for the CPU.

Once the task has been added to the list, the `__napi_schedule()` method then schedules the execution of a `softirq` of type **NET_RX_SOFTIRQ** that is responsible for processing incoming network packets.

The `net_rx_action()` method is responsible for executing `softirqs` of type **NET_RX_SOFTIRQ**. This method executes the driver-specific poll method for every `napi_struct` in the local `poll_list`. To ensure fair handling of all devices, each device has a maximum number of packets that it can process before yielding control to another device. This value is stored in the `weight` field of the `napi_struct` structure.

If a device driver is unable to process all packets within a single budget, the `napi_struct` is moved into a `repoll` list. To prevent the softirq from running for too long, a global processing limit is enforced. This limit spans all devices and is enforced in terms of both packets (`netdev_budget`) and time (`netdev_budget_usecs`). If there are still `napi_struct` structures to handle (either in the `repoll` list or new ones/leftovers in the `poll_list`), a new softirq is scheduled with the `__raise_softirq_irqoff(NET_RX_SOFTIRQ)` method.

3.1.6 Driver poll method

The driver poll method is the function associated to the `napi_struct` in charge of processing packets. It identifies the next descriptor in the RX ring and allocates a new socket buffer for the packet. Some drivers copy the entire packet into a new memory buffer, while others simply make the `skb` point to the original DMA buffer. The driver then copies additional metadata, which can vary based on the device, provided by the NIC into the `skb`.

The `eth_type_trans()` method is executed to determine the Layer 2 features of the packet. This includes the destination of the packet (this host, another host, broadcast, or multicast), which is saved in the `pkt_type` field of the socket buffer, and the Layer 3 protocol.

The `skb` is then sent to the upper layer of the network stack. The system could group multiple packets and process them at once using Generic Receive Offload (GRO). Ultimately, the `__netif_receive_skb_core()` function is called.

The driver poll method iterates until there are no more packet descriptors or the budget of the device has been consumed. If there are no more descriptors, hardware interrupts for the device are re-enabled.

3.1.7 Function `netif_receive_skb`

The `netif_receive_skb()` function is a critical component of the Linux networking stack. It applies Receive Packet Steering[8] (RPS), if enabled, which is essentially a software implementation of Receive Side Scaling (RSS). Unlike RSS, which selects the queue and hence the CPU that will run the hardware interrupt handler, RPS selects the CPU to perform protocol processing above the interrupt handler. This is achieved by placing the packet on the desired CPU's backlog queue and waking up the CPU for processing.

RPS has several advantages over RSS. Firstly, it can be used with any Network Interface Card (NIC). Secondly, software filters can easily be added to hash over new protocols. Lastly, it does not increase the hardware device interrupt rate, although it does introduce inter-processor interrupts (IPIs). Thanks to RPS, `netif_receive_skb()` can enqueue the socket buffer (`skb`) to the `softnet_data` of another CPU and wake it with an inter-CPU interrupt (`input_pkt_queue`).

In addition, `netif_receive_skb()` optionally timestamps the packet if required by upper layers and not already timestamped. It also optionally delivers the packet to network taps using the `deliver_skb()` method.

Network taps are systems used by packet sniffers to capture traffic. Every Layer 3 protocol handler, including both network taps and traditional handlers like IPv4, is represented by a `packet_type` structure. This structure contains the ethertype of the protocol handled (a wildcard in case of network taps) and a function to handle the packet. Network taps spanning all devices are stored in the `ptype_all` list, while device-specific ones are in the `dev->ptype_all` list.

`netif_receive_skb()` also handles Traffic Control (TC). TC is a system used to implement Quality of Service by managing queuing in the transmit direction. In the receive path, it is possible to classify the traffic and apply actions according to the result.

If the device is connected to the port of a Linux Bridge, `netif_receive_skb()` passes the packet to the bridge. The packet is handled with the `br_handle_frame()` function. The return value of the function determines whether the packet must proceed in the network stack (`RX_HANDLER_PASS`) or has been consumed by the bridge (for example, forwarded) (`RX_HANDLER_CONSUMED`).

Finally, `netif_receive_skb()` delivers the packet to the appropriate L3 protocol handler with `deliver_skb()`. Protocol handlers are stored in the `ptype_base` global array. Upon system initialization, every handler adds itself to the proper list with the `dev_add_pack()` method.

3.1.8 Transmission Path

Queueing Disciplines

`Queueing disciplines` serve as the cornerstone of `Traffic Control` within the Linux Kernel. A queueing discipline stipulates the algorithm employed to enqueue and dequeue packets within the Linux networking stack. Predominantly, there are two categories of queueing disciplines: `classless` and `classful`.

Classless queueing disciplines adopt a uniform approach towards all packets. Illustrations of classless queueing disciplines encompass:

- `pfifo_fast`: This embodies a straightforward First In, First Out queue and is the default choice in Linux.
- `tbf` (Token Bucket Filter): This utilizes a token bucket to regulate the pace at which packets are dequeued.

Conversely, **classful queueing disciplines** facilitate the division of traffic into classes using classifiers (also referred to as filters), and manage them in diverse manners. Instances of classful queueing disciplines comprise:

- `htb` (Hierarchical Token Bucket): This represents a classful variant of the token bucket.
- `prio`: This enforces strict priority queuing.

Queueing disciplines can be structured in hierarchies, with the egress of a queueing discipline being either a device or another queueing discipline. This provision enables the implementation of intricate traffic management strategies.

The `clsact` (classify action) queueing discipline is a distinct type of queueing discipline that abstains from queuing but merely classifies packets and executes actions based on the outcome. This permits the implementation of advanced packet processing strategies, such as traffic shaping or filtering.

Function `dev_queue_xmit()`

The `dev_queue_xmit()` function is a pivotal element in the transmission pathway of the Linux networking stack. It is utilized by both Layer 3 protocol handlers and Layer 2 subsystems, such as the bridge, to transmit a packet. This function accepts a `skb` as a parameter. The caller is responsible for setting all the necessary information for transmitting the packet, including the output device and L2 addresses.

If available, the `clsact` queuing discipline is applied.

Each transmit (tx) hardware queue of the device is linked to a structure of type `netdev_queue`. For multi-queue devices, the suitable tx queue is chosen with the `netdev_core_pick_tx()` method.

If **Transmit Packet Steering** (XPS) [8] is enabled, the queue is selected based on the current CPU from a map that associates every core to a set of available queues. Transmit Packet Steering is a mechanism designed for intelligently determining which transmit queue to use when transmitting a packet on a multi-queue device.

Otherwise, a hash function is applied on the skb. In both scenarios, if the packet is bound to a L4 socket, the queue is stored in the socket structure (`struct sock`) and will be used for all subsequent packets generated by the socket. The `netdev_queue` structure

contains a field of type `Qdisc` representing the queuing discipline used to manage packets of that queue.

The packet is then passed to the queuing discipline with the `enqueue()` virtual method. Each queuing discipline provides its own implementation of this method. The queuing discipline is then executed with the `qdisc_run()` method to identify the next packet (or packets) to send.

Function `ndo_start_xmit()`

Every driver in the Linux networking stack may perform device-specific operations according to the supported features and architecture of the device. These operations are common across most drivers:

- **Checking for sufficient descriptors:** The driver checks whether there are enough descriptors available to send the packet. Multiple descriptors might be needed for a single packet, especially in cases involving offloading features or fragmented (`skbs`). If there aren't enough descriptors available, the driver signals the upper layers to stop sending packets using the `netif_tx_stop_queue()` function and returns `NETDEV_TX_BUSY`.
- **Mapping the packet buffer:** The driver maps the packet buffer to make it accessible by the device through Direct Memory Access (DMA). This allows the device to read the packet data directly from memory, which can significantly improve performance.
- **Filling the descriptors:** The driver fills the next available descriptors in the transmit (tx) ring with information about the packet. This includes details such as the size of the packet, the location of the packet data in memory, and any offloading features that should be used.
- **Notifying the device:** Once the packet data has been prepared and the descriptors have been filled, the driver notifies the device that new data is available for transmission. This is typically done by writing to a specific register on the device.
- **Handling packet transmission:** Usually, the device notifies the driver of a packet sent with the same interrupt used for reception. The poll method executed in the `NET_RX_SOFTIRQ` can either schedule the cleanup of the `skbs` in the `NET_TX_SOFTIRQ` with the `dev_kfree_skb_irq()` or directly free resources.
- **Re-enabling transmission:** If transmission was previously stopped due to a lack of descriptors, the driver can optionally re-enable transmission once sufficient resources are available. This is done using the `netif_wake_queue()` function.

3.1.9 Conclusions

The Linux networking stack is indeed a very complex and feature-rich system. It's designed to handle a wide range of networking tasks, from basic data transmission and reception to

more advanced features like traffic shaping, routing, and network security. This complexity displays the versatility and power of the Linux networking stack, but it can also make the system challenging to understand and manage.

One of the key characteristics of the Linux networking stack is its continuous evolution. The open-source nature of Linux means that developers around the world are constantly working on new features and improvements. These enhancements can range from performance optimizations and bug fixes to entirely new protocols and services.

However, this complexity and continuous evolution can also make the Linux networking stack difficult to tune and forecast its behavior. The performance and behavior of the networking stack can be influenced by a wide range of factors, including hardware characteristics, system configuration, network conditions, and the specific workloads being run.

Chapter 4

Wireguard

4.1 Overview and Terminology

WireGuard [10, 11] is a modern and simple VPN that employs cutting-edge cryptography. Its design aims to outperform IPsec by being faster, simpler, leaner, and more versatile, all while avoiding IPsec's complexity. WireGuard also aspires to offer significantly better performance than OpenVPN.

It is designed for a wide range of applications and can operate on anything from embedded interfaces to supercomputers, making it suitable for various scenarios.

It was initially launched for the Linux kernel and has since become cross-platform, with support for Windows, macOS, BSD, iOS, and Android, making it widely deployable.

Although WireGuard is currently undergoing intensive development and evolution, it is, as of this writing, ready for use in a production environment.

4.1.1 Main features

- **Easy to use**
  WireGuard is designed to be straightforward to configure and deploy. A VPN connection is established simply by exchanging public keys, with WireGuard transparently handling everything else. There's no need to manage connections, worry about state, or oversee daemons. In fact, WireGuard offers an interface that is remarkably simple and robust.

- **Cryptography Details**
  WireGuard employs cutting-edge cryptographic techniques, including the Noise protocol framework, Curve25519, ChaCha20, Poly1305, BLAKE2, SipHash24, HKDF, and other secure, trusted constructions. It opts for conservative and sensible choices that have undergone review by cryptographers.

- **Minimal Attack Surface**
  WireGuard is designed with a focus on simplicity and ease of implementation. It is

intended to be easily implemented in a minimal amount of code, making it readily auditable for security vulnerabilities. Comprising approximately 4000 lines of code, WireGuard stands in stark contrast to behemoths like IPsec or OpenVpn. Auditing these massive codebases can be a daunting task, even for large teams of security experts. In contrast, WireGuard is designed to be comprehensively reviewable by single individuals.

4.1.2 Inner details and terminology

WireGuard securely encapsulates IP packets over UDP. When a WireGuard interface is added, it's configured with a private key and the public keys of various peers, enabling packet transmission across it. Issues of key distribution and pushed configurations fall outside the scope of WireGuard; these are matters best addressed at other layers. In this respect, WireGuard's model closely resembles that of SSH. Both parties possess each other's public keys, allowing them to start exchanging packets through the interface immediately.

To better understand the following sections, two terms need to be clarified:

- **Device**: This term is synonymous with 'interface'. It refers to the WireGuard interface that is created on a machine, which can potentially exist independently and not be associated with another interface.

- **Peer**: This term describes the association between an interface on one machine and an interface on another machine. A single device can have multiple peers associated with it. In the following chapter, the term 'tunnel' is used interchangeably with 'peer'

Cryptokey Routing

Central to WireGuard is a concept known as **Cryptokey Routing**.

This works by associating public keys with a list of tunnel IP addresses permitted inside the tunnel. Each network interface possesses a private key and a list of peers, with each peer having a public key. These public keys, which are short and simple, are used by peers for mutual authentication. They can be shared for use in configuration files via any out-of-band method, much like sharing an SSH public key for shell server access.

In WireGuard, peers are strictly identified by their public key, a 32-byte Curve25519 point. This establishes a straightforward mapping between public keys and a set of permitted IP addresses. The interface itself has a private key, a UDP port on which it listens, and a list of peers.

When an outgoing packet is transmitted on a WireGuard interface, the peer table is consulted to determine the public key to use for encryption.

Conversely, when the interface receives an encrypted packet, it will only accept it after decryption and authentication if its source IP matches the public key used in the secure session for decryption in the table.

More broadly, any packets arriving on a WireGuard interface will have a reliably authenticated source IP, in addition to the guaranteed perfect forward secrecy of the transport. Note that this is only possible because WireGuard operates strictly at layer 3. Unlike some common VPN protocols, such as L2TP/IPsec, using authenticated identification of peers at a layer 3 level results in a much cleaner network design.

In other words, when sending packets, the list of allowed IPs functions as a sort of routing table, and when receiving packets, it acts as a sort of access control list.

It is crucial that peers can send encrypted WireGuard UDP packets to each other at specific Internet endpoints.

Each peer in the cryptokey routing table can optionally predefine a known external IP address and UDP port for that peer's endpoint. This is optional because if it's not specified and WireGuard receives a correctly authenticated packet from a peer, it will use the outer external source IP address to determine the endpoint.

Since a public key uniquely identifies a peer, the outer external source IP of an encrypted WireGuard packet is used to identify the remote endpoint of a peer. This allows peers to roam freely between different external IPs, such as between mobile networks.

Send and Receive flow

When a packet is locally generated (or forwarded) and is ready to be transmitted on the outgoing WireGuard interface:

1. The plaintext packet arrives at the WireGuard interface.
2. The system searches for a match between the destination IP address of the packet and a peer. If no match is found, the packet is dropped, the sender is informed by a standard ICMP 'no route to host' packet, and -ENOKEY is returned to user space.
3. The symmetric sending encryption key and nonce counter of the secure session associated with the identified peer are used to encrypt the plaintext packet using ChaCha20Poly1305.
4. A header, containing various fields explained in section 5.1.2, is prepended to the encrypted packet.
5. This header and encrypted packet are sent together as a UDP packet to the Internet UDP/IP endpoint associated with the peer, resulting in an outer UDP/IP packet containing a header and encrypted inner-packet as its payload. The peer's endpoint is either pre-configured or learned from the outer external source IP header field of the most recent correctly-authenticated packet received. If no endpoint can be determined, the packet is dropped, an ICMP message is sent, and **-EHOSTUNREACH** is returned to user space.

When a UDP/IP packet reaches a specific UDP port of the host, which is the listening UDP port of the interface associated with the WireGuard interface:

1. A UDP/IP packet containing a specific header and an encrypted payload is received on the correct port.
2. Using the header, WireGuard identifies that it is associated with a specific peer secure session, checks the validity of the message counter, and attempts to authenticate and decrypt it using the secure session's receiving symmetric key. If it cannot identify a peer or if authentication fails, the packet is dropped.
3. Once the packet has been authenticated correctly, the source IP of the outer UDP/IP packet is used to update the endpoint for the peer.
4. After the packet payload is decrypted, the interface receives a plaintext packet. If this is not an IP packet, it is dropped. Otherwise, WireGuard checks if the source IP address of the plaintext inner-packet routes correspondingly in the cryptokey routing table. If no match is found, the packet is dropped.
5. If the plaintext packet has not been dropped, it is inserted into the receive queue of the WireGuard interface.

4.1.3 Versions

Wireguard is more than just the version implemented in the Linux kernel. It is divided into several subprojects and repositories. This section aims to present two important user-space implementations. User-space implementations are advantageous because they allow Wireguard to be used in different contexts compared to the Linux kernel. However, they can be slower because the packets are not confined to the kernel space and must be copied to the user space to be processed by the module.

- **Wireguard-Go:**
  This is the version of Wireguard written in Golang. Section 5.2.5 explains the architecture details and discusses Tailscale's improvements in this version.

  Go, also known as Golang, is an open-source programming language created by Google in 2007. It is designed to be efficient, easy to learn, and provides support for modern hardware architectures. Go is known for its simplicity, built-in support for concurrency, and robust standard library. Many tech giants like Google, Netflix, Twitch, Ethereum, Dropbox, Kubernetes, Docker, Heroku, and more use Go in their tech stacks.

- **Wireguard-rs:**
  This is the Rust user-space version of Wireguard. This version also adopts a TUN interface in its underlying architecture.

  Rust is a modern, multi-paradigm, high-performance programming language that first appeared in 2015. It was created by Mozilla Research employee Graydon Hoare as a personal project in 2006, and Mozilla began sponsoring the project in 2009. Rust

is designed to be blazingly fast and memory-efficient because it has no runtime or garbage collector. Rust's rich type system and ownership model guarantee memory and thread safety, enabling you to eliminate many classes of bugs at compile time. In addition, Rust provides first-class support for concurrent programming.

There is also another version of Wireguard in Rust developed by Cloudflare, which is known as BoringTun.

4.1.4 Current Adoption

Owing to its features and simplicity, WireGuard is adopted by leading solutions that enable the interconnection of remote clusters, such as Liqo and Submariner. The existence of multiple versions and the open-source nature of the project facilitate its adoption in various contexts.

Several VPN solutions are built on top of WireGuard. For instance, Tailscale is based on WireGuard-go (Further details on section 5.2.5). Similarly, Netbird and MullVad, which are other VPN solutions, also rely on WireGuard.

4.2 Internal Architecture

Wireguard[12] operates by creating a virtual network device. The device can handle multiple tunnels, each one towards a different peer. The routing table on the host is in charge of forwarding packets that need encapsulation on the Wireguard interface, which then encrypts the packets, locates the proper peer and adds the external headers of the tunnel. When a wireguard-encapsulated packet is received it is decapsulated, decrypted, and the inner, original packet is reinjected into the network stack by the WireGuard interface.

The next two sections provide more details on the processing involved in these two directions, focusing on parallelization aspects.

4.2.1 Encapsulation

Traffic needing encapsulation can originate from two sources. The first source is an application running locally within the current network namespace. In this scenario, the packets will originate from the application socket. The second source is another host, in case this namespace is performing forwarding. In this scenario, packets will be received from an interface, either physical, if the traffic comes from another physical host, or virtual, if traffic comes from a container or VM running on the same host. In both cases the routing layer of Linux will decide to forward the packet on the Wireguard interface, based on the destination address or other policies. Once the packet is transmitted on the Wireguard interface, Wireguard code kicks in. Encapsulation is composed of three main steps, which operate in different execution contexts and are subject to different parallelization opportunities:

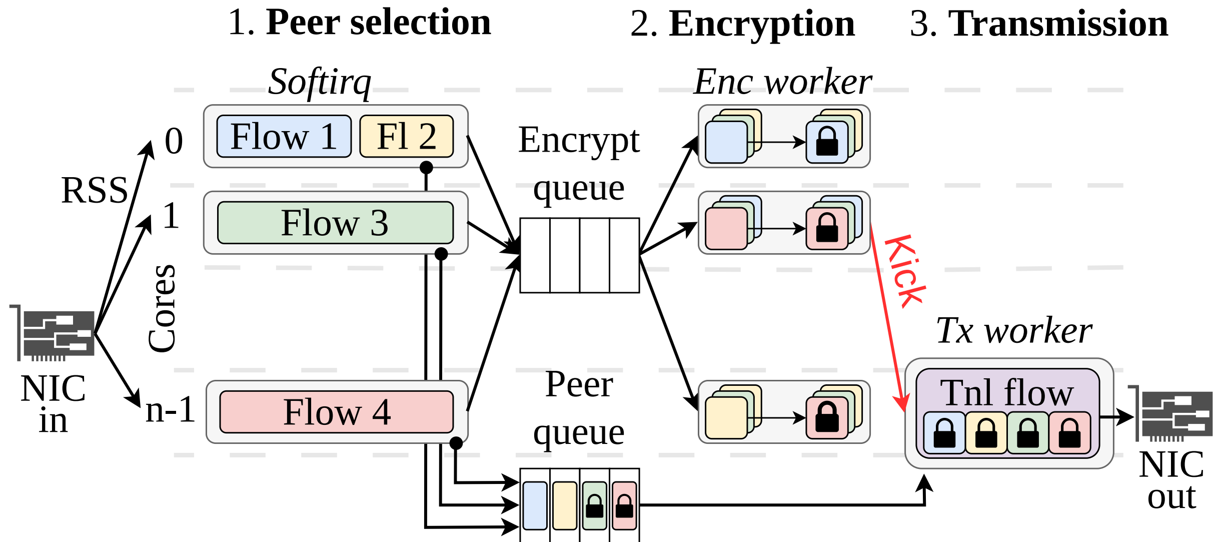

Figure 4.1: WireGuard processing on CPU cores on the encapsulation side in the single tunnel scenario.

1. **Peer selection:**
   This process occurs in the same context in which the original packet was processed. It might be a softirq if the packet comes from an interface, or a syscall if the packet originates from an application socket. Once the peer is selected and the packet nonce is properly set, the packet is tagged as unencrypted and is enqueued in two different queues. The first queue is a per-device multi-producer multi-consumer (MPMC) ring buffer. Its purpose is to hold all the packets awaiting encryption directed to all the peers associated with the wg device. This queue is drained by the encryption workers running, by default, on all the CPU cores of the machine. The second is a per-peer queue and is drained in a serial manner by a single CPU core when transmission occurs. Its purpose is to preserve the order of transmission. After enqueuing, an encryption worker is woken on a CPU chosen in a round-robin manner. The parallelism level of this phase is one CPU core per original flow.

2. **Encryption:**
   This step occurs in the CPU core that has been selected to execute the worker. The worker pulls packets from the ring buffer until the buffer is empty, encrypting them and marking them accordingly. It optionally wakes the per-peer TX worker if not already running. This phase is potentially done by all the CPU cores of the node.

3. **Transmission:**
   This step is performed on a dedicated kernel worker thread assigned to each peer. This worker executes always on the same CPU core, selected during the handshake phase. The worker's task is to pull packets from the serial queue until it is empty or until an unencrypted packet is encountered. These packets are then encapsulated

and transmitted on the physical NIC, following all necessary routing steps. The parallelism level of this phase is 1 CPU core per peer.

4.2.2 Encapsulation Code

The function in Listing 4.1 can be regarded as the core of the transmission path. It is also used in the receive path; on the contrary, it is completely bypassed in the inline version described in Chapter 7.

Listing 4.1: function wg_queue_enqueue_per_device_and_peer() in file queueing.h

```
static inline int wg_queue_enqueue_per_device_and_peer(
    struct crypt_queue *device_queue, struct prev_queue *peer_queue,
    struct sk_buff *skb, struct workqueue_struct *wq)
{
    int cpu;

    atomic_set_release(&PACKET_CB(skb)->state, PACKET_STATE_UNCRYPTED);
    /* We first queue this up for the peer ingestion, but the consumer
     * will wait for the state to change to CRYPTED or DEAD before.
     */
    if (unlikely(!wg_prev_queue_enqueue(peer_queue, skb)))
        return -ENOSPC;

    /* Then we queue it up in the device queue, which consumes the
     * packet as soon as it can.
     */
    cpu = wg_cpumask_next_online(&device_queue->last_cpu);
    if (unlikely(ptr_ring_produce_bh(&device_queue->ring, skb)))
        return -EPIPE;
    queue_work_on(cpu, wq, &per_cpu_ptr(device_queue->worker, cpu)->work);
    return 0;
}
```

The first parameter the function receives is a **per-device queue** (`struct crypt_queue`) that is shared among all peers. The Wireguard device is associated with three different `crypt_queues`: **encrypt_queue**, **decrypt_queue**, and **handshake_queue**. The first one is exploited in the transmission path.

Listing 4.2: multicore_worker and crypt_queue from the file device.h

```
struct multicore_worker {
    void *ptr;
    struct work_struct work;
};

struct crypt_queue {
    struct ptr_ring ring;
    struct multicore_worker __percpu *worker;
    int last_cpu;
};
```

`crypt_queue` is implemented as a ring buffer because has inside the field `ring` of type `struct ptr_ring`, which is basically a circular buffer that can contain a certain number of skbs. During Encapsulation its primary purpose is to hold all the packets awaiting encryption. It is shared among all the online CPUs in the machine, can be filled by all the peers associated with the specific device, and can be drained by one of the cores of the machine during encryption. Because of this multi-producer and multi-consumer scenario, the `ptr_ring` is protected by two spinlocks. Inside the `struct crypt_queue`, there is a field named `worker` that defines a number of multicore workers equal to the number of existing CPUs. Each of these workers is basically a function: for the `encrypt_queue` this field is initialized with the function `wg_packet_encrypt_worker` that is responsible to encrypt the packets during the encryption phase.

The second parameter of the function `wg_queue_enqueue_per_device_and_peer()` is a queue specific for each peer of type `prev_queue` that is basically an skbs list. In the transmission path, its purpose is to hold each packet that the peer has to transmit. Packets in the list can have different states; in fact, each skb has a field called `cb->state` which can be **UNCRYPTED**, **CRYPTED**, or **DEAD**.

The third parameter is the linked list of skbs the device should put in the two previous queues.

The fourth parameter is a work queue. Its primary purpose is to allocate a task (e.g., a function) to a specific CPU.

In particular, the function **wg_queue_enqueue_per_device_and_peer()**:

- Changes the state of the received list of packets to **UNCRYPTED**.
- Put the list in the per-peer queue. If this operation fails, the error **-EINVAL** is returned.
- Thanks to the function `wg_cpumask_next_online` choose a CPU in round-robin fashion, and the (list of) packet(s) is put in the per-device queue. If this operation fails, the error **-EPIPE** is returned.
- Starts a worker thread that executes the function `wg_packet_encrypt_worker`, in the CPU chosen before. Its purpose is to encrypt all the packets in the `encrypt_queue` until it is completely empty.

4.2.3 Decapsulation

Three asynchronous steps take part in the decapsulation process:

1. **Reception:**
   The NIC directs traffic to different CPU cores through Receive Side Scaling (RSS) or another core selection mechanism, such as ntuple filters. These techniques allow for traffic steering with a maximum granularity of flow level. Given that a single tunnel between two wg devices corresponds to a single UDP flow, the processing of this step (Reception) is confined to a single core per tunnel(or peer), even if it encapsulates

multiple flows. Traffic is identified as WireGuard-encapsulated and directed to an in-kernel WireGuard socket. Upon receipt by the socket, the peer is identified and the appropriate key pair is selected for decryption. Similar to encapsulation, the packet is enqueued in both a per-device MPMC queue for decryption and a per-peer queue for serial reception. A decryption worker is selected in round-robin. The parallelism level for this phase is one CPU core per tunnel/peer.

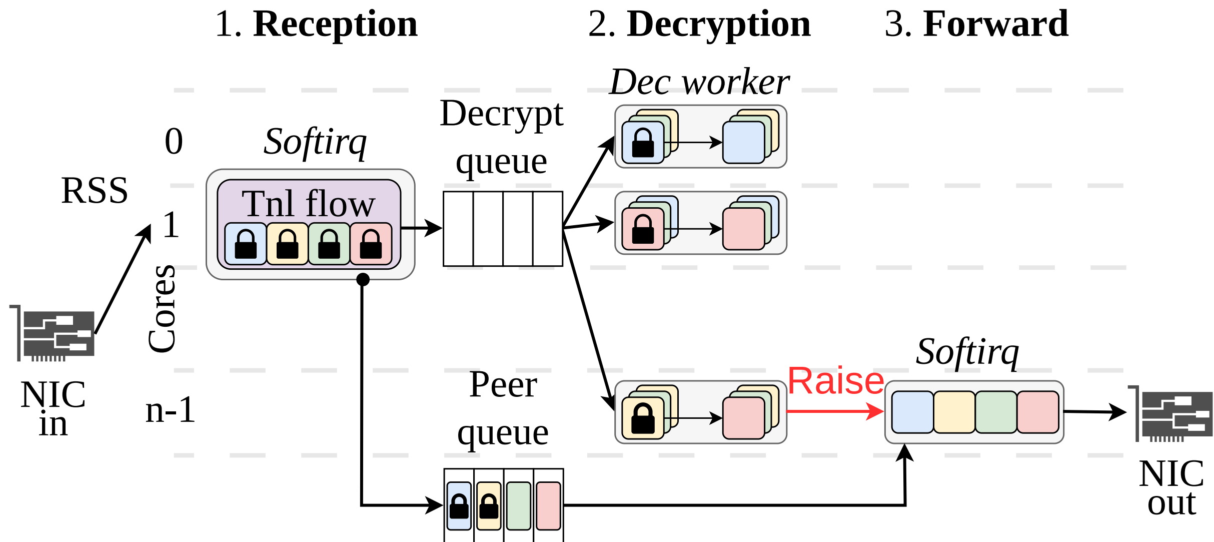

Figure 4.2: WireGuard processing distribution on CPU cores on the decapsulation side in the single tunnel scenario.

2. **Decryption:**
   This step is symmetric to the encapsulation phase, with the selected worker draining the per-device queue, decrypting each packet and updating its status accordingly.

3. **Forwarding (to application or on another interface):**
   Each peer is associated with a NAPI structure which establishes the appropriate callback function executed for packets received on the Wireguard interface. If the NAPI callback of a peer is not already running, the worker schedules it after packet decryption, by raising a softirq on the current CPU core. The function drains packets from the per-peer queue until it is empty or an encrypted packet is found, handing them to the upper layers of the network stack where they can be forwarded to an application or a different interface. The parallelism of this phase is 1 CPU core per peer.

4.2.4 Decapsulation Code

The core function is **wg_queue_enqueue_per_device_and_peer** (Listing 4.1). The parameters passed to this function differ from the previous analysis done in (Section 4.2.2)

in the following ways:

- **decrypt_queue:** The first parameter is now **decrypt_queue** (of type **struct crypt_queue**) associated with the current device. The worker within the queue is linked to the function **wg_packet_decrypt_worker**. This function's role is to drain the packets from the queue and decrypt them until the queue is empty. As previously noted, this worker is scheduled on a CPU selected in a round-robin manner.
- **rx_queue:** This is a per-peer list of skbs, which prevents out-of-order reception.
- The third parameter is the skb, which should be decrypted.
- The last parameter is a **workqueue**, which is essentially a list of tasks awaiting execution.

The behavior of the function remains consistent with the previous description (Section 4.2.2).

4.3 Alternatives

4.3.1 Ipsec

IPsec, which stands for Internet Protocol Security, is more than just a simple VPN technology. It's an extension of the IP protocol suite that provides security to IP and the protocols in the upper layers. IPsec was designed by the Internet Engineering Task Force (IETF) and it provides security at the network level. It also includes protocols for managing the key exchange.

One of the most common uses of IPsec is to provide a Virtual Private Network (VPN) service, which secures communications over public networks.

IPsec operates in two distinct modes:

- **Transport Mode:**
  This mode is mainly used by endpoints and not by gateways. In this mode, IPsec only protects the payload of the IP packets that come from the upper level, leaving the Layer 3 (L3) header unprotected. This mode is typically used when there's a need to protect data in a host-to-host scenario. All hosts involved in the communication need to be aware of the IPsec protocol.
- **Tunnel Mode:**
  In this mode, IPsec can protect the entire IP packet. This mode is more commonly used to implement VPNs because it allows for intercommunication between gateways, typically between two routers. It's more complex than Transport Mode, but only the gateways need to have the IPsec software. In this scenario, a new IP header is added, resulting in an inner IP header (the original IP header) and an outer IP header (the new IP header).

IPsec includes two important protocols:

- **Authentication Header (AH):**
  is a protocol that authenticates the source host and provides integrity to the payload only. It creates a message digest using a hashing algorithm.

- **Encapsulating Security Payload (ESP):**
  is a protocol used to provide encryption, authentication, and confidentiality. It's more commonly used compared to AH. An ESP trailer is added to the payload, and both the payload and trailer are encrypted. Afterward, the ESP header is added. The ESP header, payload, and trailer are then used to create the Authentication Data, which is added at the end of the ESP trailer.

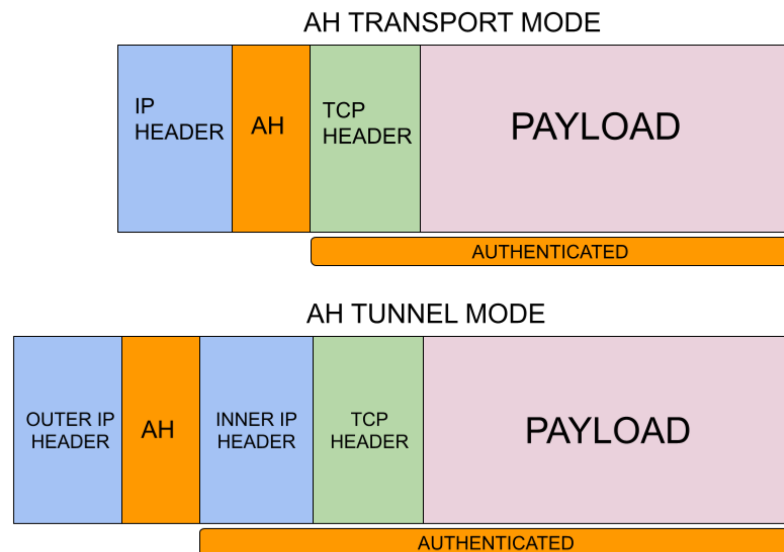

Figure 4.3: AH in transport and tunnel mode

The two figures 4.3 and 4.4 illustrate the different configurations of generic packets using both AH and ESP in transport and tunnel mode.

Although IPsec, including its multicore versions, is a viable option for encryption, this thesis steers in a different direction. The decision to concentrate on WireGuard was influenced by performance tests that revealed WireGuard to be faster than IPsec. Furthermore, the thesis's primary aim is to enhance the Liqo use case, which is dependent on WireGuard.

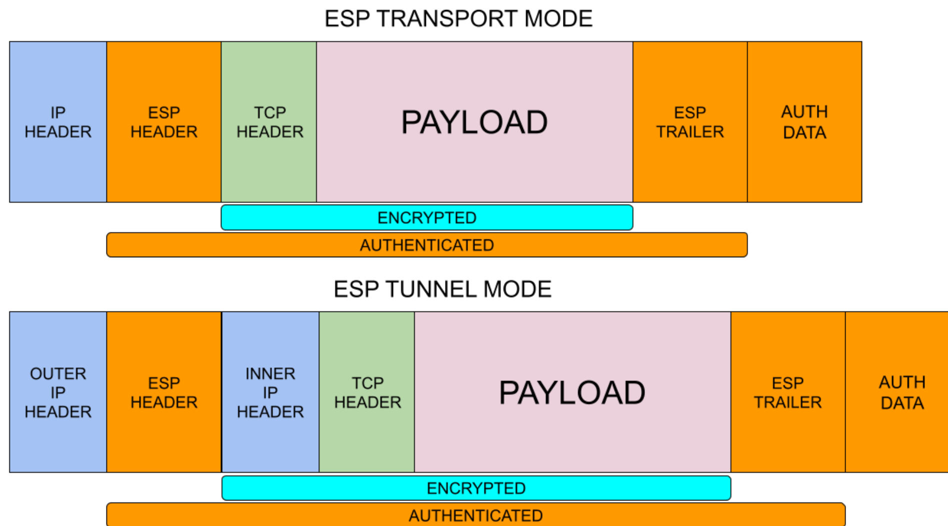

Figure 4.4: ESP in transport and tunnel mode

4.3.2 OpenVpn

OpenVPN is an open-source VPN project that creates secure connections over the Internet using a custom security protocol based on SSL/TLS. It's not just an open-source project but also a tunneling protocol, the name of the company behind the site, and the commercial products that support the open-source work.

Secure Socket Layer (SSL) is an encryption-based Internet security protocol. Netscape developed it in 1995 to ensure privacy, authentication, and data integrity in Internet communications. SSL is the predecessor to the modern Transport Layer Security (TLS) encryption used today.

OpenVPN's code is available for audits, meaning anyone can find and fix vulnerabilities. This transparency increases the security and allows for continuous inspection and improvement of the code.

4.4 Perfomance

The rationale behind WireGuard's parallelism is to enable superior performance, particularly when a single tunnel is involved, compared to other VPN technologies. The WireGuard whitepaper [10] presents some performance results, with tests conducted using a 1Gbps link. Specifically, it states that WireGuard outperformed OpenVPN and both IPsec modes, with less CPU usage than its competitors. According to the author, WireGuard is significantly faster than OpenVpn because it is implemented in kernel space. This eliminates the scheduler's added latency and overhead, as well as the need to copy packets between user space and kernel space multiple times. Moreover, it is claimed that WireGuard's cipher suite is simpler and incurs less overhead than IPsec. This simplicity

and efficiency contribute to WireGuard's overall performance advantage.

An independent study [13] conducted in 2020 by Lukas Osswald, Marco Haeberle, and Michael Menth compared various VPN protocols to validate the claims made in the WireGuard whitepaper. The study concluded that IPsec, with AES-based encryption, is an excellent choice for performance within a virtualized environment, specifically when CPU pinning is not utilized. In non-virtualised environments, WireGuard outperforms IPsec by approximately 30 percent. Furthermore, WireGuard surpasses OpenVPN in all tested scenarios.

Despite parallelism being an excellent choice to enhance the throughput of a single tunnel, there are other serial stages in computations that do not permit Wireguard to scale, so the maximum throughput reachable depends on the CPU and is around a few Gigabits/s. This problem is investigated deeply in Section 6

Chapter 5

Known WireGuard Optimization Techniques

There are multiple strategies to increase Wireguard's performance that stand orthogonal to the solution this thesis proposes. This chapter specifically introduces two primary methods: one pertains to the adjustment of the MTU, and the other involves the application of offloading techniques.

5.1 The role of MTU

The more straightforward way to enhance Wireguard's performance is to operate on the MTU by increasing it

5.1.1 What is MTU

The Maximum Transmission Unit (MTU) represents the largest Protocol Data Unit (PDU) that can be forwarded over a network without fragmentation. In the context of the widely utilized TCP/IP stack, the MTU refers explicitly to the maximum size of an IP packet, including headers, that can be dispatched to the lower layers. When Ethernet is utilized at the L2 layer of the stack, an Ethernet frame can accommodate up to 1500 bytes. So this is typically the standard value for the MTU, although it can vary depending on the specific technology implemented at the Link Layer level.

When a packet is encapsulated within an Ethernet frame for transmission, the actual size of the packet traversing the network is determined by the MTU plus the Ethernet Header. Each Ethernet frame consists of several fields: destination and source MAC addresses (each 6 bytes), EtherType (2 bytes), and concludes with a Frame Check Sequence (FCS)—a 32-bit cyclic redundancy check used to detect any data corruption during transit, consequently, the frame at Level 2 measures 1518 bytes. This size increases to 1522 bytes if IEEE 802.1Q, a standard used to support VLANs, is implemented, as it adds 4 bytes to the Ethernet Header.

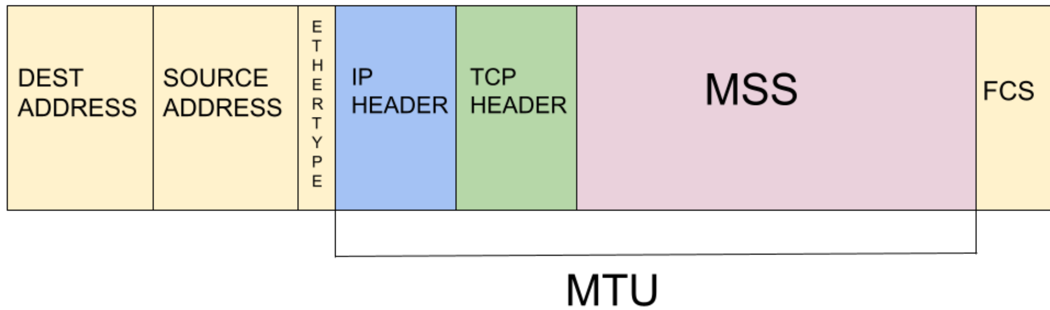

Figure 5.1: MTU and MSS

When the packet at the IP level exceeds the MTU, it must be fragmented. The IP protocol has a feature known as fragmentation, which allows for this behavior, as described in RFC 791 [14]. Generally speaking, fragmentation should be avoided [15]. The motivation is primarily because reassembling the fragments can be costly, especially when they arrive out of order. Additionally, losing a fragment equates to losing all the information of the packet. Another reason is that fragmentation is not a commonly used feature, making it more prone to errors and exploitable through various attacks. The source of the traffic plays a crucial role in preventing packet fragmentation. By setting the IP header's Don't Fragment (DF) flag to a value of 1, the source can ensure that packets that exceed the MTU in a hop are dropped. The source can also calculate the MTU of a specific path using the Path MTU Protocol [16], which enables the selection of the minimum MTU of all the hosts in a network path.

It is important to distinguish between the MTU and the Maximum Segment Size (MSS), which refers to the maximum size of the payload at the TCP level. The MSS depends on the MTU value and can be calculated by subtracting the L3 and L4 headers to the MTU value. For instance, in the case of IPv4, where the standard header is 20 bytes and the TCP header is 20 bytes, an MTU of 1500 would result in an MSS of 1460 bytes. In contrast, for IPv6, where the header size is 40 bytes (excluding all extension headers), the MSS would be 1440 bytes.

5.1.2 The Impact of changing MTU in Wireguard Performance

In Wireguard, the MTU of the device interface is determined by the MTU of the physical interface to which the packets are forwarded. Precisely, if the MTU of the physical interface is X, then the MTU of the Wireguard interface can be calculated by subtracting all the overhead that Wireguard adds from X [10]. The overhead of WireGuard is as follows:

- 20-byte IPv4 header or 40-byte IPv6 header
- 8-byte UDP header
- 4-byte type

- 4-byte key index
- 8-byte nonce
- N-byte encrypted data = MTU of Wireguard Interface
- 16-byte authentication tag

So, assuming X=1500 because the technology adopted at L2 is Ethernet, the worst-case scenario (IPv6) results in 1500-(40+8+4+4+8+16), leading to an MTU of the Wireguard interface of 1420 bytes. In the case of IPv4, this value is 1440 bytes.

Increasing the MTU of physical interfaces, and consequently, of the Wireguard interface, leads to an increase in throughput. This increase in throughput is possible because the CPU's overhead in computation is per single packet and is not affected by its size. As a result, it is possible to transmit more data with the same CPU usage. So, the same amount of data is transmitted encapsulated in fewer packets, reducing the CPU overhead per byte sent. This reduction is particularly evident in the number of times the stack is traversed, the number of interrupts used, and the number of data structures allocated (e.g., skbuffs). Additionally, the header's overhead is reduced, as the identical quantity of header bytes can encapsulate a larger PDU.

Figure 5.2 is derived from a simple setup involving two physical machines, Machine A and Machine B, directly interconnected with a link operating at a speed of 40Gbps. An iperf3 client is instantiated on one machine, while the other hosts an iperf3 server. The throughput between the two machines is measured by varying the MTU of the physical interface, with a granularity of 500 bytes, and repeating the iperf3 test. Since IPv4 is used for each test, the MTU of the Wireguard interface is obtained by subtracting 60 bytes from the physical value.

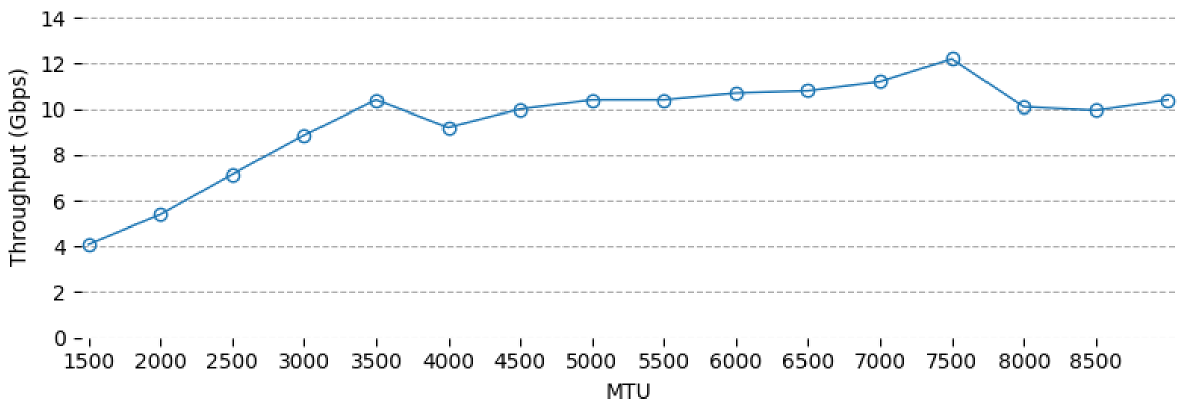

Figure 5.2: Performance of Wireguard increasing MTU

The figure vividly demonstrates the direct correlation between MTU and throughput. As the MTU increases, the throughput also rises, eventually surpassing the initial value

by more than double. However, this upward trend is not indefinite. Once the MTU value hits 3500 bytes on the x-axis, a saturation point becomes apparent. This saturation is a direct result of the characteristics of the Network Interface Cards (NICs) involved in the measurement. The unique two-peak profile of the throughput is a clear indication of the influence of the buffer size of the NIC driver on the measured throughput.

However, increasing the MTU is unsuitable for most use cases. For instance, when interconnecting geographically distant data centers over a network controlled by different ISPs, the MTU must be kept to the standard value to avoid packet fragmentation. This situation is typical because data centers are often in different regions or states, with no dedicated connection between them,and the ISPs' networks use a standard MTU. In addition, increasing MTU can generally also affect latency, especially for applications with real-time requirements.

This approach remains valid and usable when the network is entirely under the control of the entity managing the data centers, as well as on virtual links. For example, Amazon AWS allows Jumbo frames with an MTU of 9001 bytes. The challenge with these scenarios is that a VPN technology such as Wireguard can become redundant when complete network control exists.

For these reasons, even though increasing the MTU can lead to a throughput increase and remains valid in some use cases, the approach is not feasible in all the complex situations that may arise. A different approach is necessary, one that can also be combined with varying the MTU when this is permitted

5.2 Offloading Techniques

5.2.1 Introduction and Main Purpose

An alternative method to obtain the same advantages garnered by augmenting the MTU value is rooted in offloading techniques. Generally speaking, within the context of networking, offloading a specific function onto the NIC implies that the function is not carried out by the stack but rather executed by the driver or the hardware of the NIC. This approach allows for the postponement of specific operations that may be costly, and if enabled in hardware, it can save CPU cycles. The upcoming sections provides a general overview of offloading techniques, with a particular focus on their implementation within the Linux kernel.

5.2.2 TSO,GSO and LSO

When a system needs to send large chunks of data out over a computer network, the chunks first need to be broken down into smaller segments, with dimensions equal to MSS, and sent over the stack. This process is called segmentation.

TSO stands for TCP Segmentation Offload and is a technique that delegates the task of TCP data segmentation to the NIC rather than handling it in the operating system

(specifically, in the kernel network stack). This feature is enabled by default on most modern NICs.

The NIC is responsible for dividing large data chunks into TCP segments and adding TCP and IP headers. This way, large data chunks traverse unaltered all the stack layers, and they are divided into packets just at the NIC level, reaching the dimension of the egress interface's MTU. In Linux, the activation of TSO is contingent on the support for partial checksum offload and scatter/gather. If the Tx checksum offload for a given device is disabled, TSO is also disabled.

Checksum offload is another technique in which the NIC calculates the checksums for IP, UDP, and TCP prior to transmission over the network, saving CPU cycles. Checksums are bit sequences that play a crucial role in maintaining data integrity. Scatter/Gather is a technique that enables the transmission or reception of multiple memory buffers via DMA, which may not be contiguous.

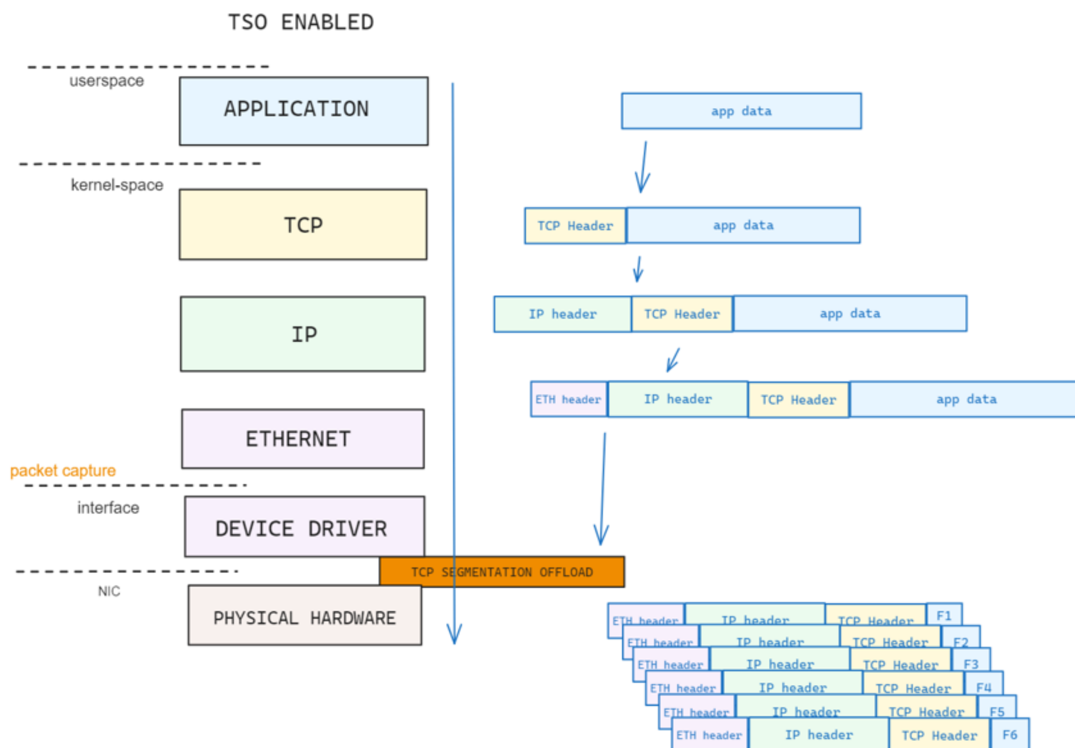

Figure 5.3: TSO

TSO offers several advantages, notably a reduction in CPU usage and overhead. This reduction is achieved by sending the same amount of data to the NIC using fewer interrupts, which is particularly beneficial when transmitting large volumes of data. Fewer packets traverse the stack (due to packet aggregation into large bursts), and the overhead associated with individual packets is reduced. All these factors lead to an increase in throughput. There are also some drawbacks: TSO can pose challenges for interactive applications that

often use small packets and require low latency (a packet may have to wait for other packets before being sent to the NIC). In addition, programs like tcpdump cannot see each packet when TSO is enabled because they capture packets before that segmentation happens and only see large, aggregated packets.

TSO is sometimes indicated also as LSO (Large Send Offload). Specifically, LSO is more generic and not only related to TCP.

GSO [17] stands for Generic Segmentation Offload. It is similar to TSO and is used when the driver or the NIC cannot offload. The segmentation phase is delayed as much as possible. Ideally, it happens in the driver, but more frequently, it is performed in the lower part of the stack. It supports not only TCP but also UDP. This technique proves beneficial when dealing with older NICs or when the hardware is virtual, such as when interconnecting virtual machines (i.e., veth). It is easy to manage because it is implemented in software, and software is more straightforward to extend than hardware.

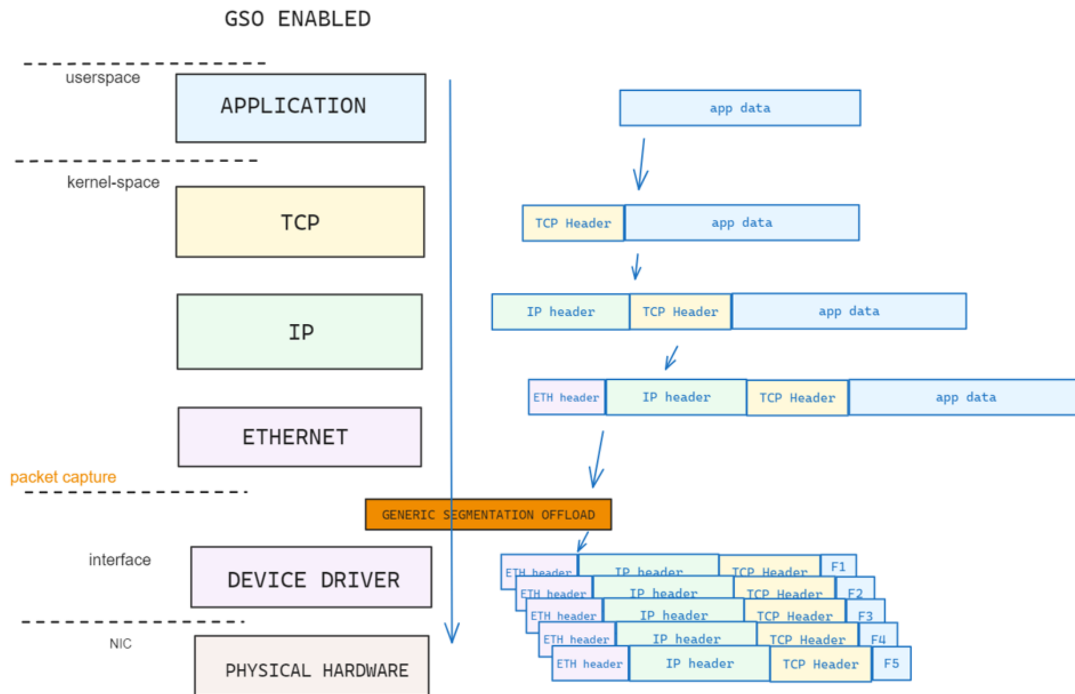

Figure 5.4: GSO

The two figures 5.4 and 5.3 illustrate that the various headers at different stack levels are created only once for the large payload. When the payload is segmented into chunks, the header is replicated (almost identically, except for the last chunk produced and for specific fields) and attached to each chunk. After replication, minor modifications are necessary to adapt the header to the various chunks. Specifically, the length field must be set in each header that needs it, and the TCP sequence number for each packet must be correctly set. The final operation involves recomputing all checksums. Segmentation is performed as late as possible to maintain the benefits of traversing the network stack a

minimal number of times. If neither GSO nor TSO is used, TCP fragments the data, and each packet created at the TCP level must traverse the stack.

5.2.3 GRO and LRO

Generic Receive Offload[18] (GRO) is the counterpart of Generic Segmentation Offload (GSO) on the receive-side. GRO coalesces packets before the upper layers of the stack process them, and it is invoked as early as possible to maximize its benefits. Coalescing is done using flows (e.g., the TCP 5-tuple). For each network device, a list of flows being coalesced is maintained. The GRO function attempts to match an incoming packet with one of the existing flows. A new flow can be created if there is no match. Alternatively, the packet goes to the upper layers as if no GRO is enabled. Specifically, if there is a match with an existing flow and the received packet is the next in sequence, the packet is coalesced.

Large Receive Offload (LRO) [19] is a feature that can only be used in TCP, not when IP forwarding is enabled due to checksum errors. Since LRO tries to coalesce all the incoming packets, it can aggregate packets that belong to different flows, making it less efficient compared to GRO.

On the other hand, GRO, which supports both UDP and TCP, is more rigorous than LRO; in fact, it compares the various fields of a packet. The MAC headers must be identical; only a few TCP or IP headers can differ.

GRO and LRO's advantages are similar to those of GSO/TSO (fewer packets traverse the stack, less CPU usage).

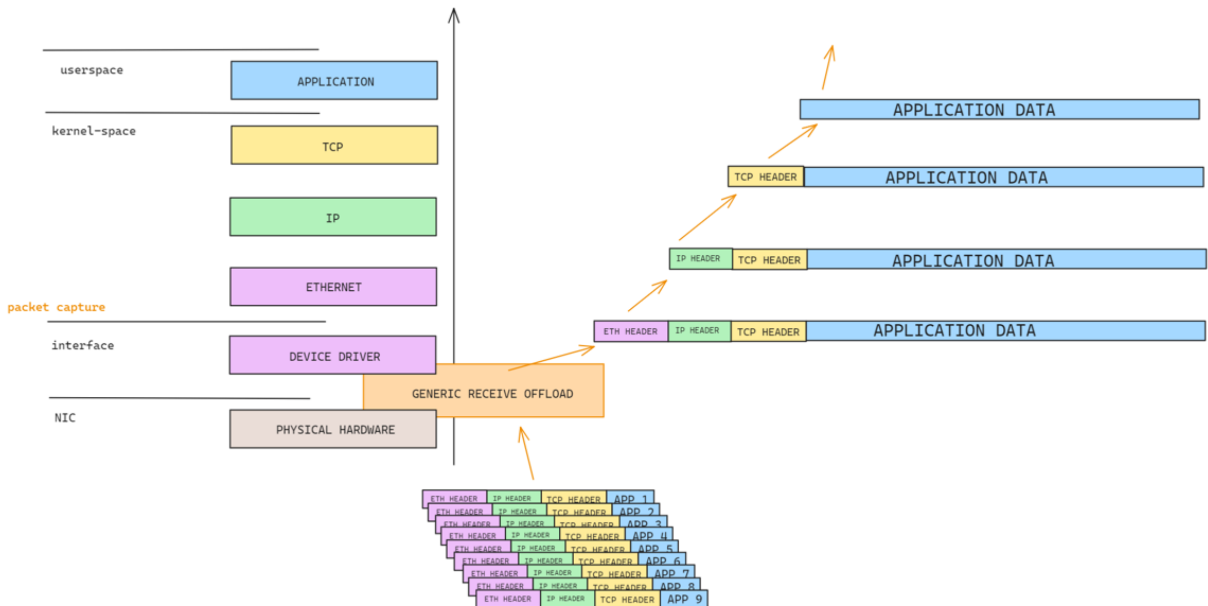

Figure 5.5: GRO

5.2.4 Offloading Techniques in UDP

Since Wireguard is encapsulated in UDP, the offloading techniques related to this protocol are crucial as they allow more data to be sent and received by the physical interface.

UDP Generic Segmentation Offload (GSO) is an offloading technique introduced by Willem de Bruijn in the Linux Kernel v4.18 [20]. It is based on the socket options SOL_UDP/UDP_SEGMENT and aims to replicate what TCP Segmentation Offload (TSO) does in the context of UDP. This technique was introduced due to the increasing adoption of the QUIC protocol.

UDP Fragmentation Offload (UFO) is a feature of the Linux kernel in which the network stack offloads the IP fragmentation functionality of large UDP datagrams to hardware. The requirements for UFO are similar to those for TSO.

UFO is currently deprecated and no longer used, having been replaced by UDP Segmentation. In UFO, an application configures a socket to bypass the MTU limit explicitly, builds large packets, and relies on IP for fragmentation. The fragments can take different paths over the network, which could pose a problem, especially at the receiver end, as highlighted in paragraph 5.1.1.

In UDP GSO, the application transmits multiple payloads concatenated. The splitting occurs at some point (in the NIC or in the driver). Each packet receives the same network layer header, UDP source, and destination ports. All but the last segment have the same UDP header, but the last may differ in length and checksum. Segmentation constructs packets identically to multiple MTU-sized calls, making it transparent to both the network and the receiver.

Paolo Abeni introduced UDP GRO in the Linux kernel in v5.0. It performs a socket lookup for each incoming packet and aggregates datagrams directed to UDP GRO-enabled sockets with a constant L4 tuple.

5.2.5 Improving Wireguard with offloading

From its inception, WireGuard incorporated GSO and GRO in its interface.

WireGuard's `net_device` is flagged as supporting hardware-based generic segmentation offload. This implies that instead of transmitting MTU-sized packets to the driver, it receives a considerably large "super-packet," typically around 65 kilobytes. Thanks to the function `skb_gso_segment()`, it's possible to divide this "super-packet" into MTU-sized chunks and assemble these packets into a linked list. Each distinct linked list of packet groups is then placed in the per-device queue (ring buffer), resulting in a ring of linked lists.

Listing 5.1: skb_gso_segment() usage example

```
1 static netdev_tx_t wg_xmit(struct sk_buff *skb, struct net_device *dev)
2 {
3     // omitted code
4
5     if (!skb_is_gso(skb)) {
```

```
        skb_mark_not_on_list(skb);
    } else {
        struct sk_buff *segs = skb_gso_segment(skb, 0);

        if (IS_ERR(segs)) {
            ret = PTR_ERR(segs);
            goto err_peer;
        }
        dev_kfree_skb(skb);
        skb = segs;
    }

    // omitted code
}
```

Listing 5.1 presents a small snippet of code associated with the `wg_xmit()` function. This function is a callback registered in the WireGuard device and represents the initial function triggered in the transmission path when the interface receives a new packet to encapsulate and transmit. Among the various operations performed by this function, one notable operation is segmenting the received skb. This process results in a linked list of skbs that can be collectively processed in the subsequent steps of the path. This efficient handling of packets contributes to the overall performance and speed of the WireGuard protocol.

On the receiving end, Generic Receive Offload (GRO) integrates seamlessly into this model, with clusters of packets from GRO being decrypted in bundles. This allows WireGuard to leverage the generally high single-threaded performance of modern processors while still benefiting from the overall speed increase provided by parallelism[12].

The case of Tailscale and Wireguard-go

In 2022 and 2023, the Tailscale team dedicated significant effort to optimizing Wireguard-go.

Tailscale is basically a VPN service that establishes encrypted point-to-point connections using the open-source WireGuard protocol in particular the version written in Go, allowing only devices on a private network to communicate with each other [21].

Wireguard-go internally employs a TUN device to simulate a physical device. In Linux TUN/TAP [22] facilitates packet reception and transmission for user-space programs. It can be viewed as a simple Point-to-Point or Ethernet device that, instead of receiving packets from physical media, gets them from a user-space program and, rather than sending packets via physical media, writes them to the user-space program. Specifically, in Wireguard-go, a packet destined for tunneling is transmitted to the TUN interface. The packet is then brought into user-space, encrypted, and reinjected into the kernel stack via a UDP socket. The receive path operates similarly: packets are received by the UDP socket, brought into user-space, decrypted, and then written to the TUN, allowing them to be received by the TUN as if it were a physical device.

In the initial implementation of Wireguard-go, no offloading techniques were exploited in either the TUN device or the physical interface. The first improvement made by the

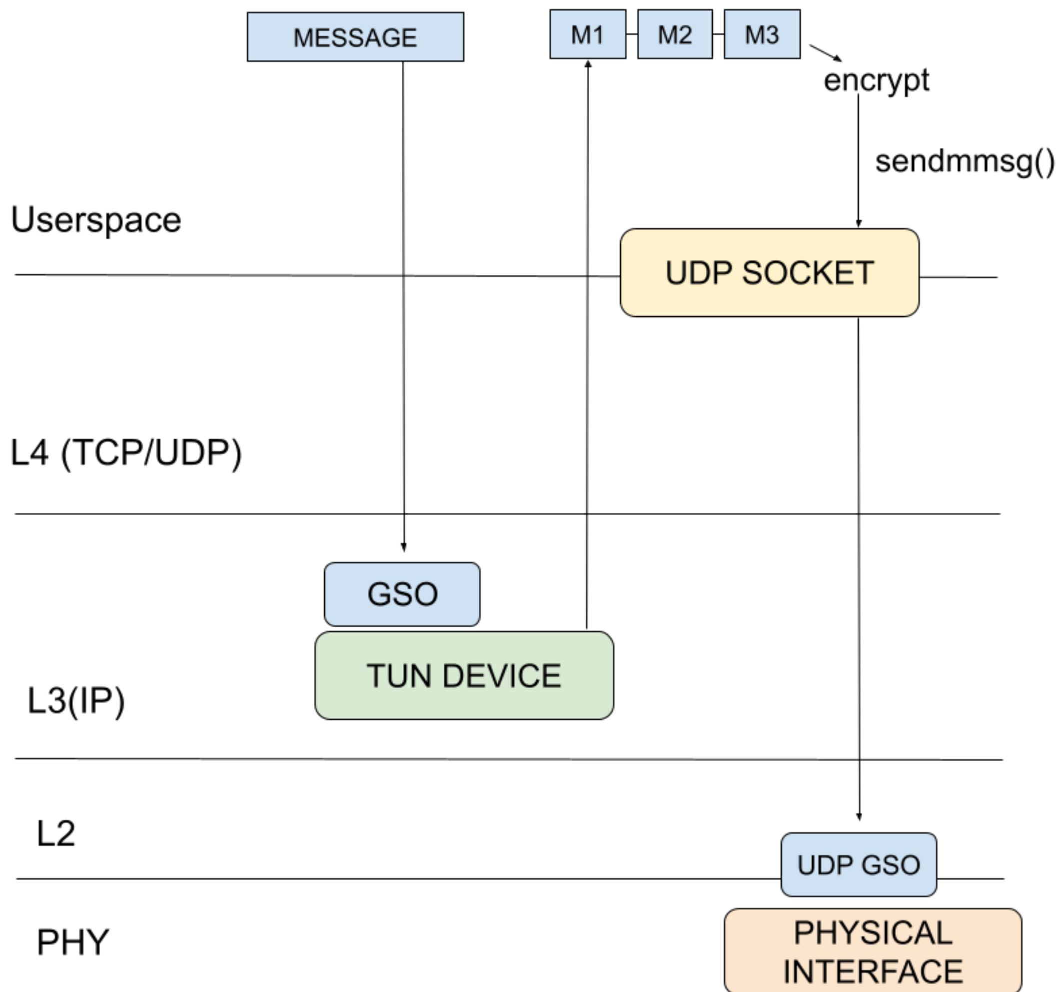

Figure 5.6: WireGuard-Go's Transmission Path Architecture including the improvement introduced by the Tailscale team

Tailscale team [23] was to enable TSO/GSO and GRO techniques on the TUN device. This allowed segmentation and coalescing to be handled directly by the interface code rather than by TCP, resulting in fewer packets traversing the stack. Thanks to this improvement, a vector of packets produced by the GSO (in the transmission path) arrives in user-space. The second optimization made by Tailscale was to leverage `sendmmsg()` instead of `sendmsg()` when transmitting out of the UDP socket. Consequently, the reverse direction begins with `recvmmsg()` instead of `recvmsg()` at the UDP socket, potentially returning multiple packets, which are candidates for coalescing just before writing to the TUN driver. Coupled with TSO and GRO, this can reduce I/O system calls on both ends

of the pipeline.

The `sendmmsg()` system call is an extension of `sendmsg()` that allows the caller to transmit multiple messages on a socket using a single system call.

Tailscale asserts that with TCP segmentation offload, generic receive offload, and the `mmsg()` system calls, significant throughput performance improvements can be achieved in wireguard-go, and consequently in the Tailscale client (up to a 2.2x improvement in wireguard-go in the best case). They also claim to have surpassed Wireguard's kernel implementation.

Following these changes, Tailscale's team continued the journey to improve Wireguard-go [24]. They identified the bottleneck as the time spent, in terms of CPU usage, sending UDP-encapsulated packets. This time was more significant than the time spent encrypting the same packets, which also held true in the reception phase.

The issue was that despite using `sendmmsg()`, the batch of packets was not transmitted through the stack as a single entity but was divided into single chunks by the UDP layer. In other words, UDP segmentation was still performed in the UDP layer, leading to the cost of traversing the stack being paid for each chunk. The Tailscale team recognized that the solution was to implement UDP GSO and UDP GRO, described in section 5.2.4 in the physical device to address this problem and further improve throughput.

In addition, they also enhanced the function used to calculate the checksum during GSO in the TUN interface. With all these modifications, they claimed to achieve a throughput of 10Gbps for Wireguard-go.

UDP GSO and GRO in Wireguard

One potential approach could be to enhance the kernel version of Wireguard with the improvements discovered by the Tailscale team. Specifically, throughput is expected to increase also in the kernel version when UDP GSO and GRO are implemented in the physical interface.

However, over the past year, not much effort has been made to incorporate this enhancement into the kernel version. The creator of Wireguard is aware of the modifications made by the Tailscale team but has not found a solution to integrate them into the main version of Wireguard. Wireguard still sends each UDP-encapsulated packet into the stack without grouping it, and it is still not capable of receiving a large packet from the physical interface.

Although this improvement could be orthogonal to the new Wireguard architecture proposed in this thesis and could potentially combine the benefits with that version, the decision made in this thesis is not to focus in this direction due to the high difficulty involved. A small attempt was made in 2018 during a Google Summer of Code, but it did not lead to any significant progress [25]

Chapter 6

Wireguard Evaluation

6.1 Testbed

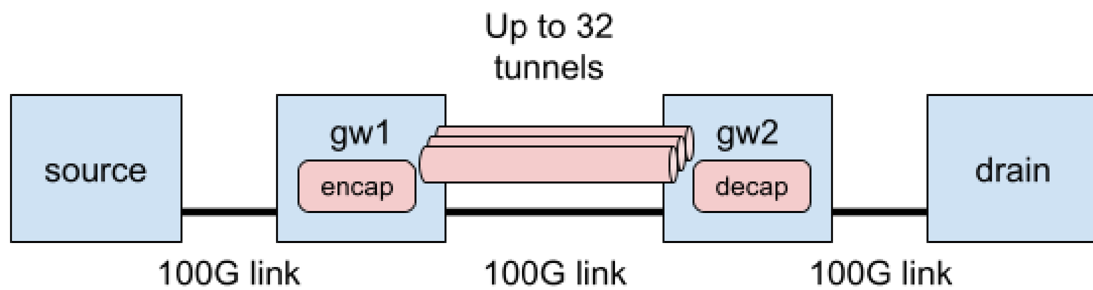

Figure 6.1: Testbed

The testbed involves four machines, as depicted in Figure 6.1. One machine acts as the traffic source, while another serves as the drain or final receiver. Between them, two other machines, *gw1* and *gw2*, are solely involved in forwarding the traffic and operate as VPN gateways. These machines are interconnected with 32 different Wireguard tunnels.

This setup is implemented as an experiment on CloudLab involving sm220u machines [26]. Each link operates at a speed of 100Gbit/s and utilizes the Mellanox ConnectX-6 DX as the Network Interface Card (NIC). Every machine is equipped with two Intel Xeon Silver 4314 16-core processors, each running at 2.40 GHz.

iperf3 is used to exchange TCP data between the source and drain. Multiple iperf3 client-server pairs are used to generate different flows to test multiple tunnels. *gw1* and *gw2* are configured so that each flow belongs to a different tunnel. Since the iperf3 test is unbalanced in a single direction, with the bulk of data moving from source to drain and only a few ACKs flowing in the opposite direction, this allows for a focus solely on the impact of the encapsulation procedure on GW1 and decapsulation on GW2, as the processing of ACKs is negligible. Hyperthreading and idle states on all nodes are disabled

to avoid inconsistent measurements. All tests are repeated ten times, and the average is presented.

The actual setup differs from the logical setup described here but does not affect the results obtained. Employing two machines for the source and the drain was challenging due to logistical reasons. Therefore, they were consolidated into a single node with two distinct namespaces, *N1* and *N2*. *N1* is connected to *gw1* through a physical interface, and the same applies to *N2* and *gw2*.

6.2 Single Tunnel Evaluation

The analysis begins by examining the maximum performance achievable in the single tunnel scenario, typically used to interconnect two clusters as in the Liqo[2] use case. For this purpose, an increasing number of TCP flows between the source and the drain machine are generated in each test, and all flows are encapsulated in a single tunnel established between *gw1* and *gw2*.

To evaluate the maximum throughput achievable, steering rules are configured on *gw1*'s NIC so that the reception of different flows coming from the source occurs on different cores and doesn't overlap with the tunnel's TX worker (i.e., when `n_flows <= n_cores - 1`).

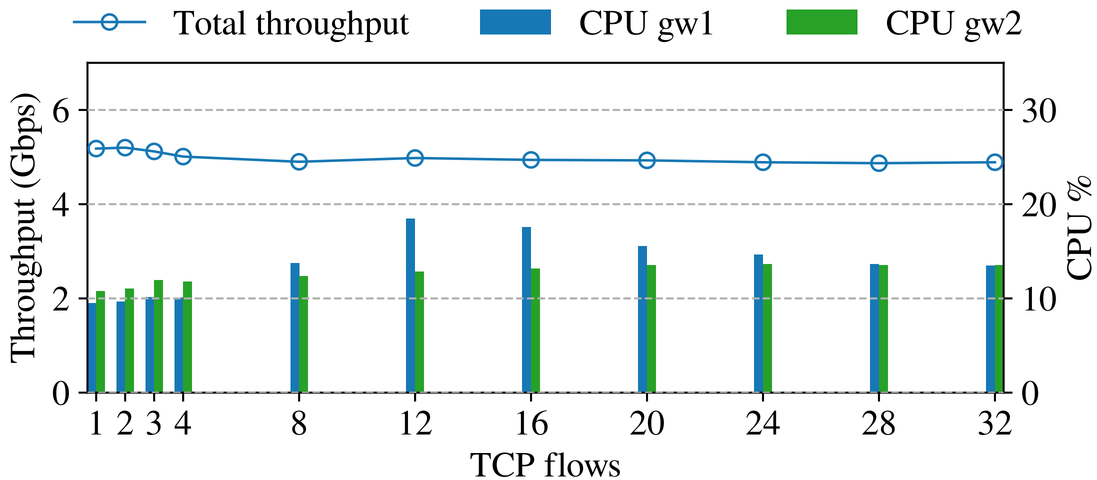

Figure 6.2: Aggregate throughput and CPU usage for an increasing number of TCP flows in a single tunnel setup.

Figure 6.2 provides a visual representation of the analysis. The x-axis shows the number of flows encapsulated in the same tunnel, while the y-axis represents the throughput. The blue line depicts the achieved throughput as the number of sessions varies. Conversely, the bars represent the CPU consumption in both machines, *gw1* and *gw2*. . Figure 6.2 also indicates that the global throughput remains consistent, unaffected by the number of

sessions encapsulated within the same tunnel. This behavior happens because while multiple flows allow leveraging multiple cores in the **Peer-Selection** stage of **Encapsulation**, the per-peer stages (**Transmission** for **Encapsulation**, **Reception**, and **Forwarding** for **Decapsulation**) are still limited to a single core. In this experiment, the bottleneck is represented by the **Reception** stage on *gw2*, which pushes the core up to 90%, limiting all other stages.

For the same reasons, the global CPU consumption remains the same, varying the number of flows.

Figure 6.3 presents the CPU usage per single core when a single tunnel manages 32 flows. The CPU usage in *gw1* is depicted in blue, while the CPU usage in *gw2* is shown in green.Note that a similar behavior is detected when other flow counts are involved.

For *gw1*, core 1 is the most utilized core. It is responsible for transmitting the packets because it instantiates the *tx_worker*. The other cores receive packets from the source and encrypt them in a distributed manner. They also decrypt ACKs from the drain received by core 20. As previously mentioned, the impact of ACKs is negligible compared to the direct flow.

In *gw2*, core 0 represents the real bottleneck, as its occupation is 100%. It is involved in receiving packets from the source. Like in *gw1*, the other cores are responsible for decrypting packets and encrypting ACKs from the drain. Core 1 is involved in transmitting the ACKs. The fact that core 0 of *gw2* is fully occupied limits the maximum throughput achievable in the single tunnel setup to the values seen in Figure 6.2.

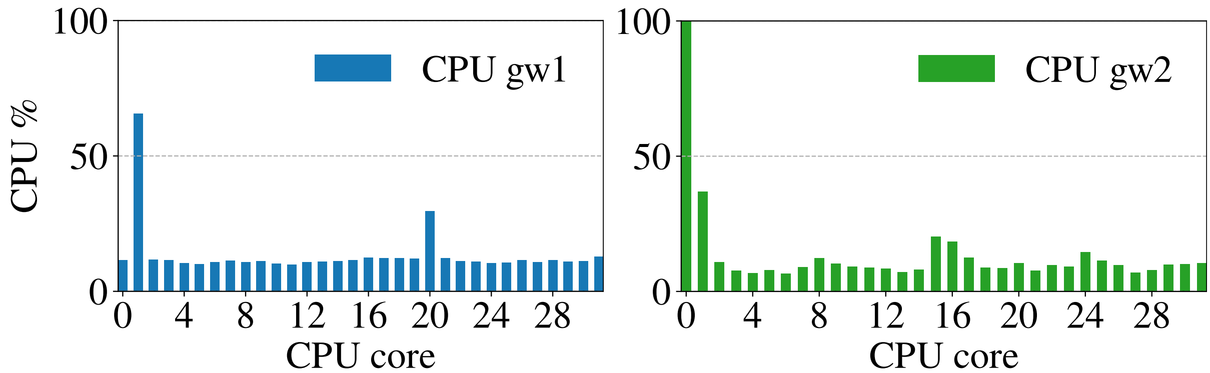

Figure 6.3: Per-core CPU usage on GW1 and GW2 in the single tunnel setup when handling 32 flows.

6.3 Multiple Tunnels Evaluation

Multiple tunnels can be used to parallelize the per-peer stages in Wireguard, and traffic can be distributed uniformly across these tunnels. This result can be achieved by creating multiple Wireguard interfaces in *gw1* and *gw2* and coupling them.

The previous tests are repeated, assigning each iperf3 flow to a different tunnel to assess the maximum throughput achievable with this solution. As in the previous test, steering

rules are configured on the NICs of *gw1* and *gw2* so that each distinct flow is processed on a different core, and there is no overlapping between flows related to the reception of packets and Wireguard tx_workers. With this configuration, it is possible to scale up to 16 tunnels, after which our servers equipped with 32 cores require overlapping (with 16 tunnels, on gw1, 16 cores perform flow reception and 16 cores tunnel transmission).

6.3.1 Results and the problem of the Napi

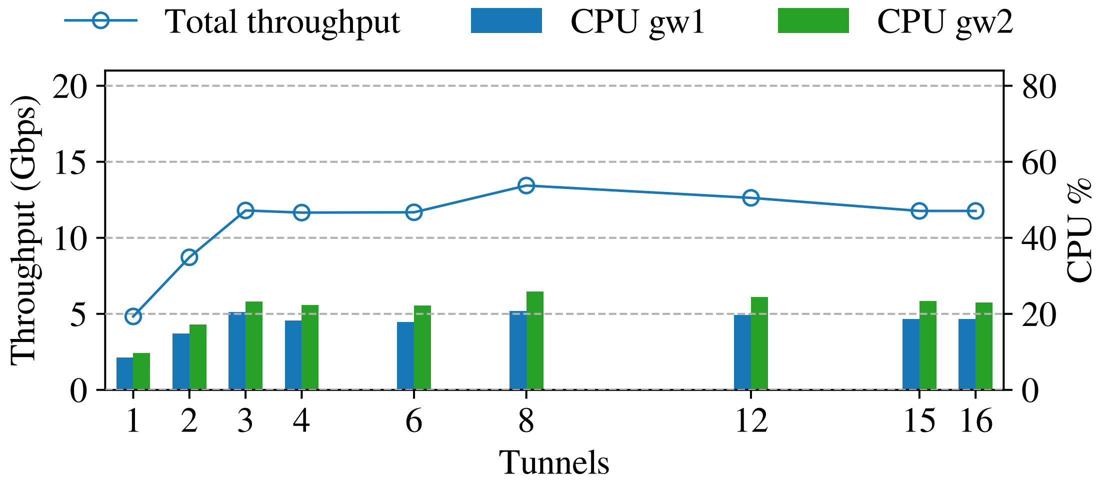

Figure 6.4: Aggregate throughput and CPU usage for an increasing number of TCP flows and corresponding WireGuard tunnels.

Figure 6.4 illustrates the throughput achieved when multiple tunnels are utilized simultaneously. Specifically, the y-axis depicts the throughput(on the left) and the percentage of cpu usage(on the right), while the x-axis represents the number of evaluated tunnels. The blue and green bars correspond to the CPU usage in *gw1* and *gw2*, respectively.

The data clearly show that this scenario offers limited scalability. As the number of tunnels increases to 3, a linear trend in throughput can be observed. However, beyond this point, the throughput stabilizes at approximately 12/14 Gbit/s, regardless of the increasing number of tunnels, with minor observable fluctuations.

Upon further analysis, it was possible to trace the root of this behavior to the manner in which the NAPI(detailed better in section 3.1.5) processes packets received by the Wireguard interface.

Each peer (tunnel) is associated with a single NAPI context and a matching NAPI `poll()` callback in charge of draining its queue. Each peer, in particular, contains a field of type `napi_struct` named *napi*. This field is initialized in the `wg_peer_create()` function. This struct napi defines a callback, `wg_packet_rx_poll()`, which is invoked when receiving packets from the driver is necessary. The primary purpose of this callback is to poll the packets that are already in the queue associated with the peer and have been decrypted by the encryption worker.

As per NAPI design, each NAPI context can only be scheduled on a single core at a time.

By default, the NAPI callback is executed inside a softirq, an execution context bound to a specific core (cannot be migrated), which doesn't terminate until there's work to do (i.e., until the `napi_poll()` has processed all the packets in the peer queue).

If not already running, a decryption worker schedules the `napi_poll()` on the softirq of the current core, otherwise, processing proceeds in the running context. This implies that the `napi_poll()` is bound to a specific core until it drains its peer's queue.

Listing 6.1: wg_queue_enqueue_per_peer_rx function

```
static inline void wg_queue_enqueue_per_peer_rx(struct sk_buff *skb, enum
    packet_state state)
{
    /* We take a reference, because as soon as we call atomic_set, the
     * peer can be freed from below us.
     */
    struct wg_peer *peer = wg_peer_get(PACKET_PEER(skb));

    atomic_set_release(&PACKET_CB(skb)->state, state);
    napi_schedule(&peer->napi);
    wg_peer_put(peer);
}
```

To better explain,the function `wg_queue_enqueue_per_peer_rx()` is invoked after the packet is encrypted. This function is responsible for changing the state of the current packet. The state, which is passed as a parameter, can be either CRYPTED or DEAD. In the first case, the packet was correctly encrypted, while in the second case, there was an issue during encryption. Following this, the function `napi_schedule()` is called. This function kicks the NAPI of the specific peer receiving the traffic.

Since `napi_poll()`s of different tunnels move to different cores over time, it is likely to have multiple functions scheduled on the same core. This, however, reduces the amount of CPU available to each function to drain its queue, making it more likely not to complete and be stuck on the core. As more `napi_poll()`s are randomly scheduled on the core the problem worsens, reducing the likelihood of recovery unless the traffic rate is reduced. This behavior can turn the core into a "black hole".

Figure 6.5 depicts this behavior with 8 tunnels by showing the CPU usage for each core on the *gw2* machine.

All cores are involved in decryption operations, which produces a base 10-20% CPU usage, and cores 1 to 8 are involved in receiving traffic of the 8 tunnels pushing usage to around 40%.

However, the bottleneck is represented by core 30 where all NAPI contexts are concentrated, saturating the capacity of the core. It is important to notice how the bottleneck core can change among different tests, being the result of the overlapping of probabilistically moving `napi_poll()` functions, however, every test consistently ended up in the "black hole" condition after some seconds.

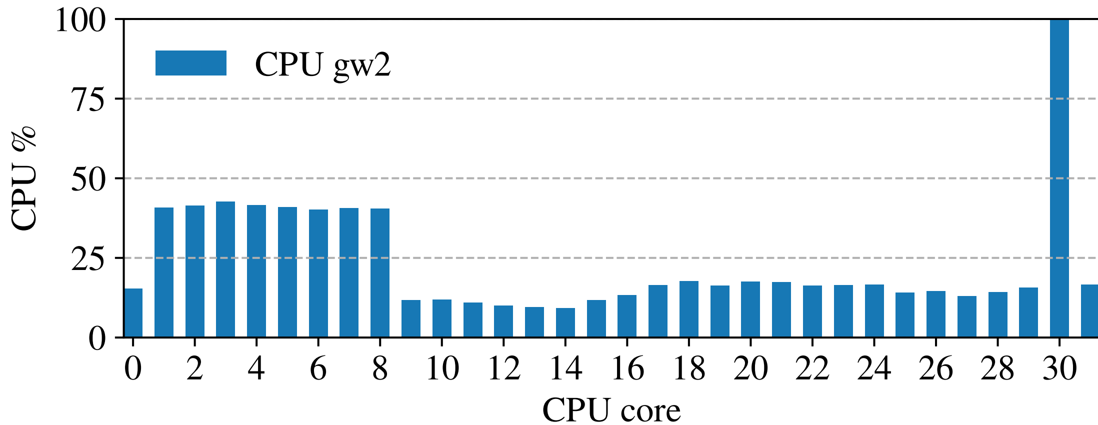

Figure 6.5: Per-core CPU usage on *gw2* when handling 8 flows spread over 8 tunnels.

6.3.2 Results with Threaded Napi

A simple solution is to switch to *threaded NAPI* by changing a flag associated with the WireGuard interface.

With *threaded NAPI*, the poll functions run in the context of preemptible threads, which can be moved among cores by the scheduler, dynamically avoiding overlapping.This also means that packets are not always drained in the same core where encryption occurs, but in our experiments this didn't have a detectable impact on performance.

Figure 6.6 presents Wireguard's performance in terms of throughput and CPU usage when multiple tunnels are used in parallel. It compares two scenarios: one using the *standard NAPI* (also depicted in Figure 6.4) and another adopting the *threaded NAPI*.

With the adoption of *threaded NAPI*, the aggregate throughput approaches a value near 50Gbps, and the CPU usage is nearly maximized. In this scenario, performance scales with the number of tunnels, and Wireguard can exploit almost all resources available on the two gateways. However, the scaling is still far from linear, and this version cannot saturate the 100 Gbps links.

The red line represents the linear case derived from the throughput that Wireguard with threaded NAPI achieves with one tunnel. As can be seen, the blue line, representing Wireguard with threaded NAPI, deviates significantly from the red one, and this deviation increases with the number of tunnels tested.

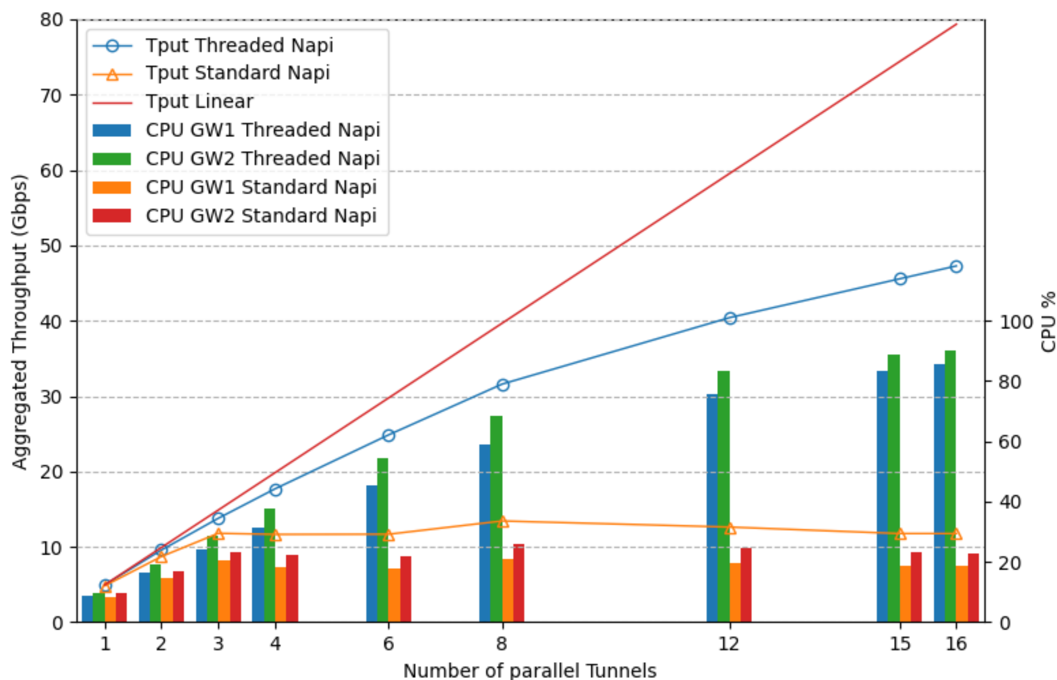

Figure 6.6: Aggregated Throughput and CPU usage on *gw1* and *gw2* comparing Standard NAPI and Threaded NAPI

6.3.3 Final Considerations

Using *threaded NAPI* combined with parallelism (multiple instances of Wireguard operating simultaneously) can achieve high aggregated throughput. This result can be obtained by simply creating a specific number of linked interfaces on the two machines and changing a flag in each interface. However, there is potential for further improvement. The next chapter suggests practical changes to the internal architecture of Wireguard to exceed the results outlined in this section.

Nevertheless, using *threaded NAPI* with parallelism is a practical compromise. It allows for achieving high, although not optimal, throughput without requiring any modifications to the inner architecture of the module. The only requirement is to set an appropriate flag on each involved interface. Importantly, there is no need for additional code or installing additional modules, providing a straightforward solution.

Chapter 7

Wireguard Inline

Simply adding tunnels in parallel does not result in linear scaling. To overcome the limitations identified in section 6, the overheads stemming from Wireguard's split architecture are entirely eliminated.

7.1 Limitations and Overhead of Wireguard

In Wireguard to allow scaling parallelizable stages (en/decryption) while keeping other stages serial, the processing is performed in a pipeline of different contexts (i.e., syscalls/softirqs and workers), potentially executed on different cores. This behavior, however, entails synchronization overheads needed to exchange packets among contexts through rings. Additionally, contexts running on different cores lead to cache misses when accessing the same packet, while contexts on the same core require additional context switches.

Handling core-context mapping is currently not possible, since WireGuard does not provide a mechanism to set the affinity of TX and en/decryption workers, nor to set the number of the latter, which defaults to the number of machine cores. This limitation, for example, leads to en/decryption workers overlapping with all other tasks running on the gateway.

On the other hand, leveraging multiple tunnels already provides a way to achieve multi-core scalability, allowing a simpler processing model for individual tunnels.

As a result, a new version of the WireGuard module, called **WireGuard Inline**, has been developed.

7.2 Idea and Main Changes

The main idea behind WireGuard Inline is to handle a packet's whole lifecycle, including encryption and decryption, in a single context on a single core, relying solely on multiple tunnels to scale on multiple cores.

This objective is achieved by eliminating the per-device ring buffer queue, which is used in packet encryption and decryption, and removing all the workers associated with these

two phases. This change is made in both Encapsulation and Decapsulation.

Additionally, in the encapsulation path, both the per-peer queue and the worker associated with transmission are removed. This is feasible because the main purpose of the per-peer queue is to maintain the order of transmission. However, within the same flows, this order is necessarily maintained when encryption is performed inline, which is a consequence of the removal of encryption workers.

As a result, packets are encrypted and transmitted over the tunnel in the same softirq or syscall context in which they are received. In the decapsulation path, the per-peer queue is retained to comply with the NAPI mechanism, which requires a list of packets to poll. Still, the NAPI is scheduled on the same softirq that receives UDP traffic from the tunnel.

It's evident that these modifications can lead to a decrease in performance, in terms of throughput, for a single tunnel, as parallelism in decryption/encryption is no longer exploited. In fact, a single flow exchanging a lot of traffic will not be able to scale on multiple cores. However, at the same time, it is advantageous with many parallel tunnels, because the operations of one tunnel do not affect the others. Furthermore, it's unnecessary to change cores in the various phases of packet processing. This implementation is associated with reduced CPU usage because only one core is exploited for all operations.

Moreover it is possible to move all computation by setting where softirqs are handled (i.e., changing RSS rules) or, during encapsulation, controlling where the syscall is executed.

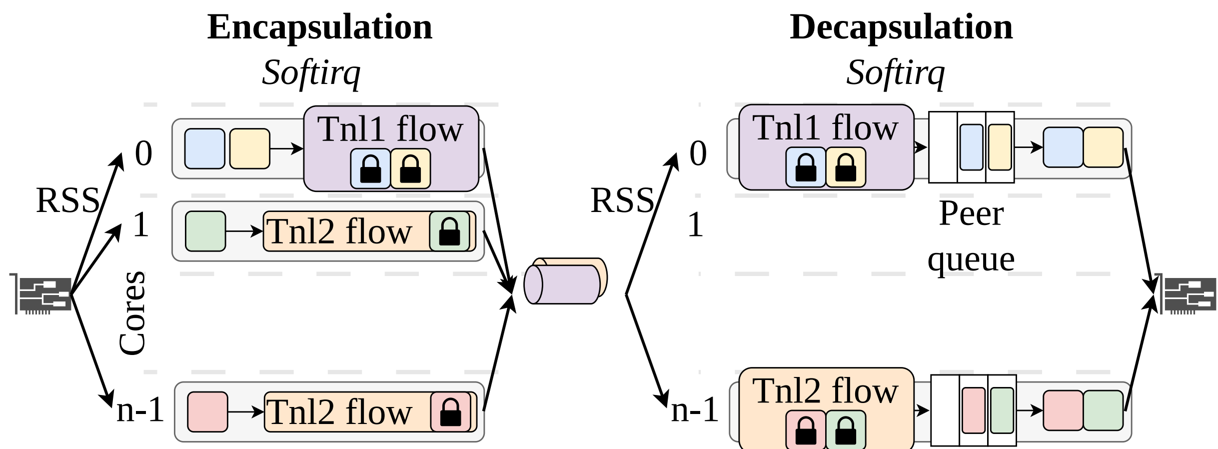

Figure 7.1: WireGuard Inline processing distribution on CPU cores in a 2 tunnels setup (note that, even if not represented here, a single core might encapsulate multiple tunnels).

7.3 Wireguard Inline Code

7.3.1 Transmission Path

Listing 7.1: Implementation of the wg_packet_create_data() function in the send.c file incorporating modifications for inline WireGuard

```
static void wg_packet_create_data(struct wg_peer *peer, struct sk_buff *
    first)
{
    struct noise_keypair *keypair;
    enum packet_state state;
    struct sk_buff  *skb, *next;
    rcu_read_lock_bh();
    if (unlikely(READ_ONCE(peer->is_dead)))
        goto err;
    keypair = PACKET_CB(first)->keypair;
    skb_list_walk_safe(first, skb, next) {
        if (likely(encrypt_packet(skb,keypair))) {
            wg_reset_packet(skb, true);
            state= PACKET_STATE_CRYPTED;
        }
        else {
            state = PACKET_STATE_DEAD;
            break;
        }
    }
    if (likely(state == PACKET_STATE_CRYPTED))
        wg_packet_create_data_done(peer, first);
    else
        kfree_skb_list(first);

    wg_noise_keypair_put(keypair, false);
    wg_peer_put(peer);
    if (need_resched())
        cond_resched();

    rcu_read_unlock_bh();
    return;
err:
    /* Error handling code */
}
```

The code remains unchanged from the original version until the function `wg_packet_create_data()` is reached. The original purpose of this function is to call the function `wg_queue_enqueue_per_device_and_peer()`, already seen in Listing 4.1, with the appropriate parameters to enqueue the packets in the per-device ring and the per-peer queue.

In the updated version of the function, depicted in Listing 7.1, two variables are declared at the outset: a keypair variable of type `noise_keypair`, which is populated with the keypair of the first skb in the received list, and a state variable of type `enum packet_state`.

The received linked list of skbs is then traversed, and each element is encrypted by calling the `encrypt_packet` function. In the standard version, this function was originally

invoked by the encrypt worker.

If the encryption of at least one element in the list fails, the state variable is updated to **PACKET_STATE_DEAD**. However, if no errors occur by the end of the loop, this variable is set to **PACKET_STATE_CRYPTED**. Note that a list of skbs is involved. This is because, as previously explained, the Wireguard interface implements GSO. Consequently, the list is composed of various chunks, each obtained by segmenting a large packet.

Upon successful completion, the function `wg_packet_create_data_done()` is called. This function is responsible for transmitting the packets toward the physical interface; this operation is done in the same core. From this point onward, the remainder of the path mirrors the original version.

7.3.2 Receive Path

Listing 7.2: Implementation of the wg_packet_consume_data() function in the receive.c file incorporating modifications for inline WireGuard

```
static void wg_packet_consume_data(struct wg_device *wg, struct sk_buff *
    skb)
{
    __le32 idx = ((struct message_data *)skb->data)->key_idx;
    struct wg_peer *peer = NULL;
    enum packet_state state;

    rcu_read_lock_bh();
    PACKET_CB(skb)->keypair =
        (struct noise_keypair *)wg_index_hashtable_lookup(
            wg->index_hashtable, INDEX_HASHTABLE_KEYPAIR, idx,
            &peer);
    if (unlikely(!wg_noise_keypair_get(PACKET_CB(skb)->keypair)))
        goto err_keypair;

    if (unlikely(READ_ONCE(peer->is_dead)))
        goto err;
    state=likely(decrypt_packet(skb, PACKET_CB(skb)->keypair)) ?
                PACKET_STATE_CRYPTED : PACKET_STATE_DEAD;

    if (unlikely(!wg_prev_queue_enqueue(&peer->rx_queue, skb)))
    {   goto err;}
    rcu_read_unlock_bh();
    atomic_set_release(&PACKET_CB(skb)->state, state);
    napi_schedule(&peer->napi);

    return;

err:
    /* error handling code */
}
```

The primary modifications take place in the `wg_packet_consume_data()` function. The original purpose of this function is to call the `wg_queue_enqueue_per_device_and_peer()` function, previously seen in Listing 4.1, with the appropriate parameters to enqueue the packets in both the per-device ring and the per-peer queue.

In the inline version of the function, a variable named state is used to indicate the state of the packet. In this scenario, only one packet is received by the interface because, as previously observed in section 5.2.5 Wireguard does not implement UDP GSO. The function calls the `decrypt_packet()` function on the skb received as a parameter. In the standard version, this function is executed by the decryption worker on a CPU that is selected in a round-robin manner.

If the decryption is successful, the state variable is set to **PACKET_STATE_CRYPTED**; otherwise, it is set to **PACKET_STATE_DEAD**.

The packet is then placed in the `rx_queue`, and its state is altered based on the value of the state variable. No additional modifications are made in the receive path. With the call to `napi_schedule()`, a napi poll function is instantiated, if not already active, on the same core where this computation has occurred.

7.4 Peformance Evaluation

The tests for the inline version are conducted using the same setup as described in Section 6.1. Specifically, during these tests, how the interrupts from source or drain machines are distributed among the various cores of *gw1* and *gw2* is controlled.

For WireGuard Inline, each gateway has two softirqs for each tunnel: one receives non-tunneled traffic from the source/drain, and the other receives tunneled traffic. Both are scheduled on the same core, ensuring each tunnel occupies just one core and distributing the tunnels across the available cores.

In contrast, for WireGuard Vanilla, interrupts are managed as outlined in Section 6.1. Each flow is associated with a unique core, and we try to avoid overlap with the TX Workers. However, when the number of tunnels reaches 16, some degree of overlap becomes inevitable. It is important to note that in the Vanilla version, the process of encryption/decryption remains distributed among all cores. Therefore, the NAPI instances can also be scheduled on all available cores.

WireGuard Inline, on the other hand, can utilize all 32 cores of the machines, instantiating 32 tunnels without any overlap between them.

Figure 7.2 depicts the test results. It is possible to observe a trend in the throughput for WireGuard Inline, represented in green, that, while not strictly linear, is not far from linearity. The throughput reaches approximately 90 Gigabit/s when 32 tunnels are utilized, and all available cores are engaged.

The data regarding WireGuard Vanilla, previously presented in blue in Figure 6.6 using only 16 parallel tunnels, are expanded in this Figure to include tests with 32 tunnels. In particular, the throughput, depicted in blue, ceases to increase when more than 16 tunnels are in use. This behavior is attributed to CPU utilization. In fact, with 16 tunnels,

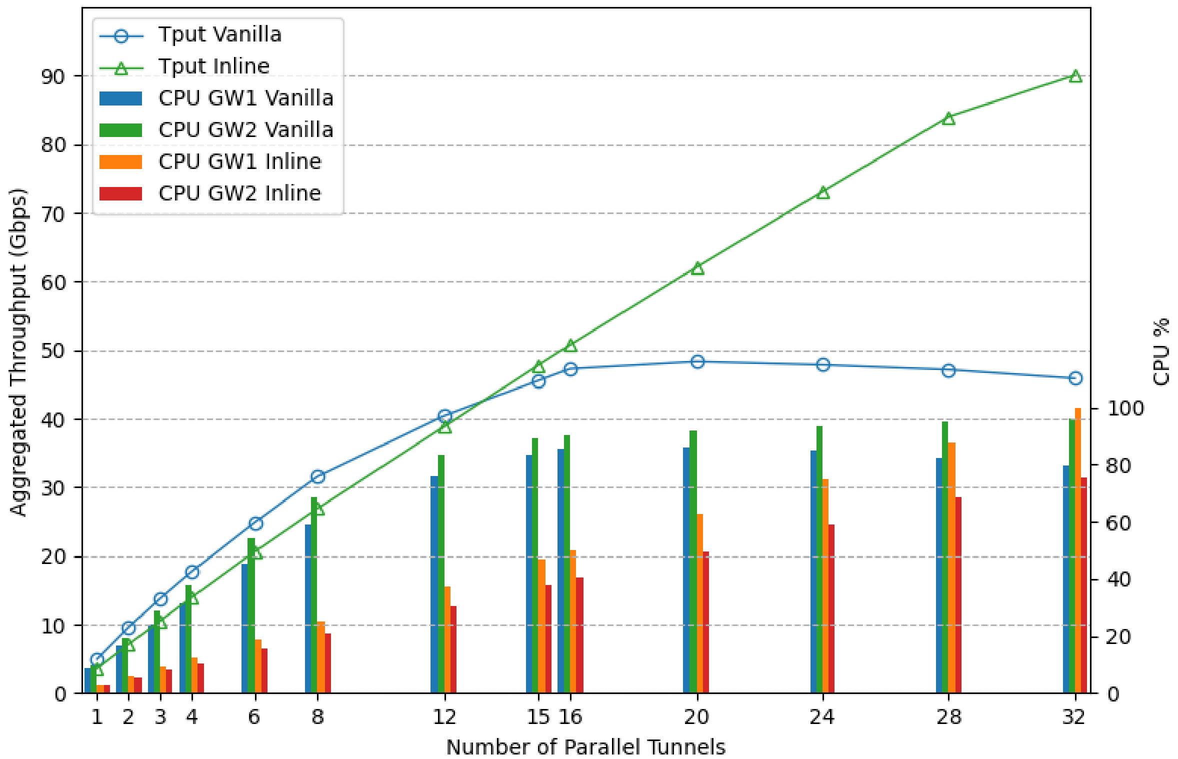

Figure 7.2: Comparison of CPU Usage and Throughput for WireGuard Inline and WireGuard Vanilla

WireGuard Inline uses less than half of the total available CPU (on both machines GW1 and GW2), whereas WireGuard Vanilla nearly exhausts all the available resources.

When fewer than 12 tunnels are leveraged, the Vanilla version of WireGuard achieves a higher throughput, albeit at a much higher CPU cost (more than 2x). This behavior highlights how the modifications introduced in Wireguard inline decrease the throughput for a single tunnel, as parallelism in encryption/decryption is no longer exploited.

Figure 7.3 illustrates the core occupancy of *gw1* when 16 tunnels are exploited for both WireGuard Vanilla (depicted in green) and WireGuard Inline (depicted in blue). As can be observed, WireGuard Vanilla tends to occupy all the machine cores, leading to resource exhaustion. In contrast, WireGuard Inline, which performs encryption and decryption on the same cores where the packets are received, only occupies the core associated with the tunnel. This results in an occupation of half of the machine's resources. In other words, WireGuard Inline has sufficient CPU to scale beyond 16 tunnels.

The same pattern is observed in *gw2* in figure 7.4. Since transmission is the bottleneck for the Inline version, WireGuard Inline's occupancy is further reduced. In fact, each core

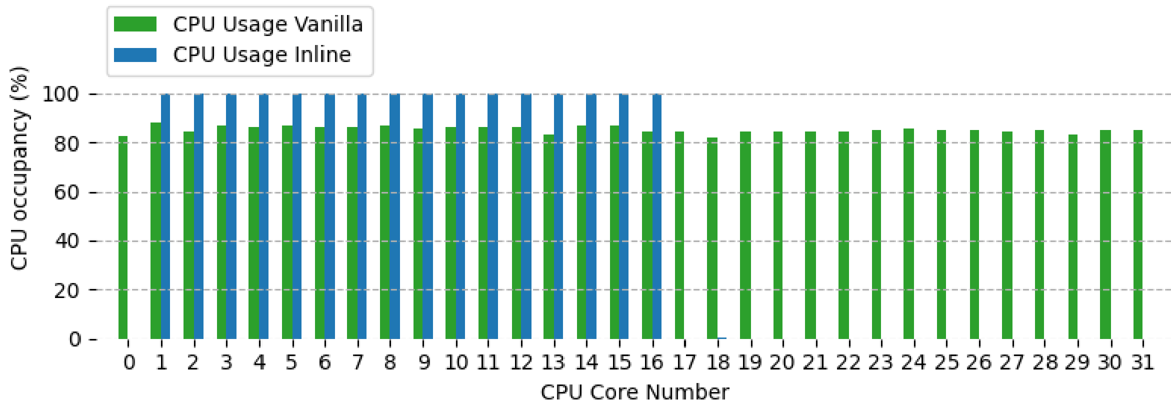

Figure 7.3: CPU Usage per Core in GW1 with 16 Tunnels in Use

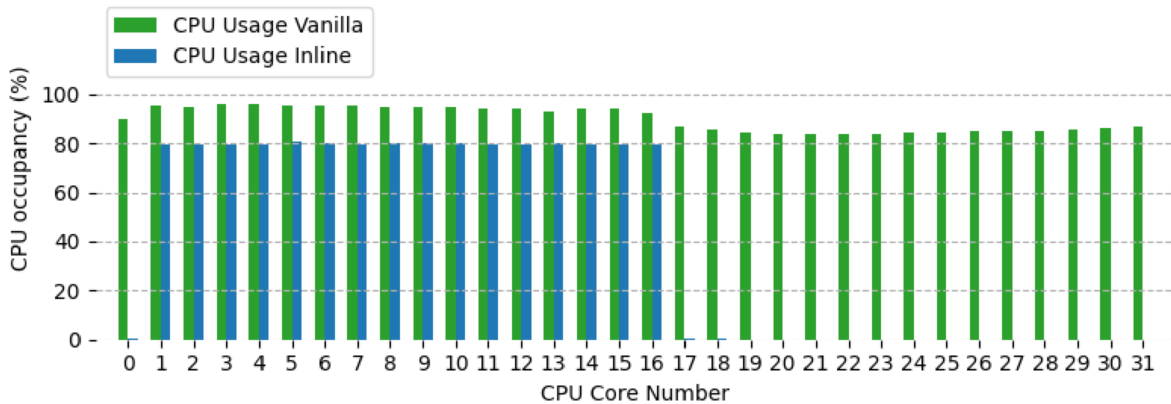

Figure 7.4: CPU Usage per Core in GW2 with 16 Tunnels in Use

associated with a tunnel is 80% occupied.

7.4.1 Efficiency Measure

Throughput and CPU usage can be compared using a single metric that encapsulates both components and measures efficiency. This metric, termed "throughput per core," is defined as the throughput associated with a single core when a certain number of tunnels are exploited. It is expressed in Gigabit/s (per core).

The calculation of this metric is done according to the formula

$$\text{Throughput per core} = \frac{\text{Throughput}}{\left(\frac{\%\text{CPU}}{\%\text{CPU}_{1\text{core}}}\right)} = \frac{\text{Throughput}}{\text{Number of cores occupied}}$$

in wich the measured global throughput associated to a certain number of tunnels is

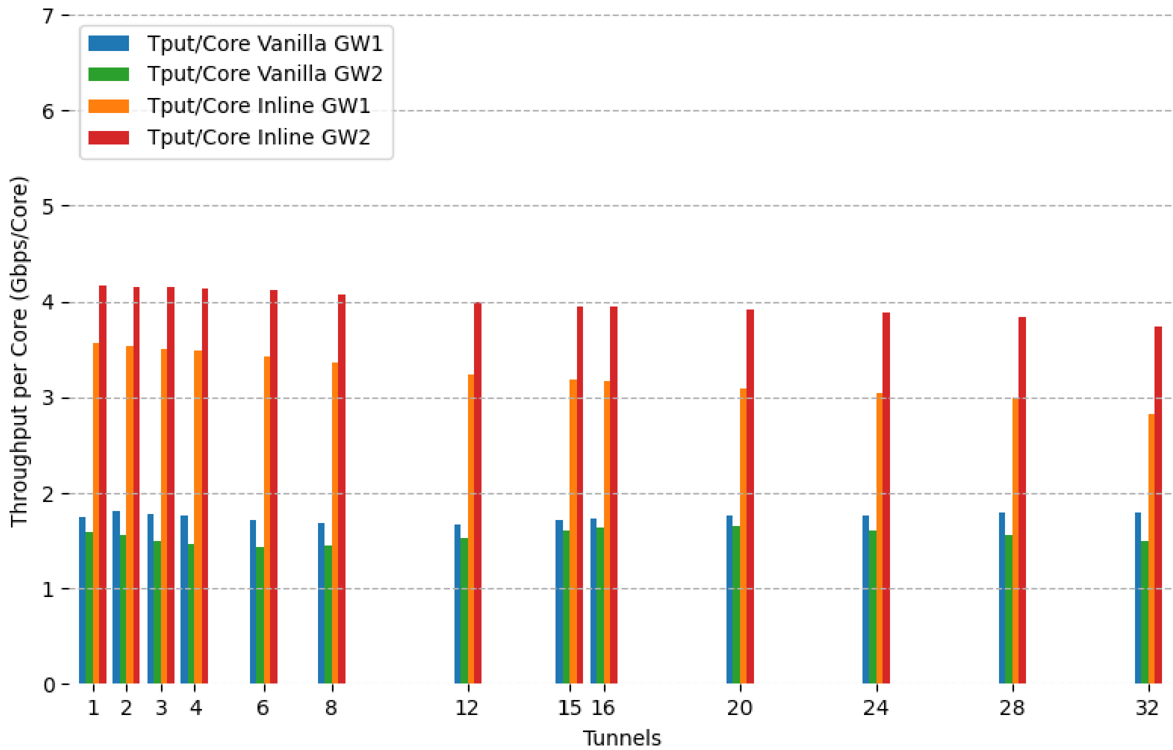

Figure 7.5

divided by the number of cores occupied by those tunnels.

Specifically, the number of occupied cores is determined by dividing the globally measured CPU percentage by a constant, 3.125%. This constant is specific to the setup and represents the CPU percentage equivalent to one core. Given that our machine has 32 cores, a single core occupies a CPU percentage equal to $\frac{1}{32} = 3.125\%$.

A higher value for this metric is preferable, as it indicates either high throughput or low CPU usage. This metric was calculated on both machines, GW1 and GW2, while exploiting an increasing number of parallel tunnels.

Figure 7.5 illustrates the metric's results. Specifically, the metric was evaluated on both *gw1* and *gw2* for both the Vanilla and Inline versions of WireGuard. Even though WireGuard Vanilla's absolute throughput is higher for a small number of tunnels, its efficiency is approximately half that of WireGuard Inline. However, as the number of tunnels increases, the values for WireGuard Inline show a slight decrease, suggesting that the throughput is not perfectly linear.

Chapter 8

Conclusions and Future Work

This work was inspired by the challenge faced by various technologies, such as Liqo and Submariner, which were used to interconnect remote clusters. These technologies struggled to achieve substantial throughput, often peaking at just a few Gigabits per second. The underlying architecture, which relied on Wireguard, was identified as the root of the problem. Consequently, Wireguard was selected as the technology to optimize in order to improve the performance observed in the cases mentioned above.

The thesis initially analyzed Wireguard's performance and the issues that prevented scalability with its current architecture despite its inherent parallelism. This lack of scalability was attributed to the presence of several serial stages that acted as bottlenecks. Specifically, the thesis demonstrated that it was impossible to increase throughput in a single tunnel scenario by only increasing the number of encapsulated flows because the bandwidth was divided among the various flows.

Then, the thesis examined the established methods to enhance Wireguard's throughput, highlighting their orthogonality to the approach adopted in the subsequent work. It concluded that increasing the MTU is beneficial as it allows for throughput enhancement, but its applicability is limited to a few specific use cases. Techniques such as Generic Segmentation Offload (GSO) and Generic Receive Offload (GRO) (with their associated variants) that enabled the offloading of packet segmentation and coalescing to the NIC were deemed interesting. However, their full integration into the kernel version of Wireguard posed significant challenges.

Parallelism, or the instantiation of multiple tunnels, was recognized as the preferred method for scaling. However, it was demonstrated that merely parallelizing Wireguard Vanilla did not enhance performance. This poor performance was caused by another issue related to the superimposition of different `napi_poll()` functions on the same core. A preliminary mitigation was proposed to address this issue: enabling threaded napi on all Wireguard interfaces. While suboptimal, this initial solution did not require code modification; only a flag needed to be set. It is immediately available and allows for significant performance improvement, achieving around 50Gbps in the testbed used. It represents a good compromise, offering high throughput with slightly reduced efficiency.

Subsequently, a novel Wireguard architecture optimized to exploit parallelism was proposed. This architecture was designed to eliminate all overhead associated with having multiple instances of Wireguard in parallel. In particular, all queues that allowed parallelism in the context of a single tunnel were removed, resulting in a version that could process each packet in the context of a single softirq. This version demonstrated impressive throughput, surpassing Wireguard Vanilla when several tunnels (>12) were utilized. In the chosen setup, it was able to achieve a throughput close to 90 Gbps, nearly doubling the performance of the vanilla version. It also exhibited improved efficiency compared to Wireguard Vanilla.

As stated in the introduction, this is merely a starting point. There are some limitations associated with this architecture and several open points that could be considered future opportunities.

8.1 Current Limitation

There are some limitations to the proposed architecture.

- The current version was implemented in kernel space by modifying the kernel module and reinstalling the kernel on *gw1* and *gw2*. If the goal is to distribute this version, there are two ways:
  The first is to distribute a custom kernel that will be installed on the machine. This approach is not within the reach of an average user. In addition, recompiling the kernel is a slow procedure that takes time. Furthermore, companies often rely on managed versions of the kernel and are not willing to change that version because they would lose all the support.
  The second is to push the modifications to the current kernel version. This means a public version requires 2-3 years to be usable. This is because patches need to be accepted and merged into the kernel, and then the kernel should be adopted by the main Linux distributions. Both of these strategies are inefficient. An alternative could be to develop a userspace implementation, as will be explained in the following section.

- The current modifications decrease the throughput for a single tunnel, as parallelism in encryption/decryption is no longer exploited. While this does not represent a problem when a high number of flows can be spread across multiple tunnels, it could be a limitation if we have few large flows since each flow is limited to a single tunnel and hence cannot be parallelized. This problem can be solved with a hybrid approach.

8.2 Future Work

- **Splitting Sessions Across Various Tunnels:**
  In all the tests conducted in Chapter 7, sessions were manually limited to specific tunnels. To better leverage WireGuard Inline, a fair distribution mechanism among

the various tunnels is essential. There are already several potential starting points integrated into the Linux Kernel.

One such starting point is ECMP (Equal-Cost Multi-Path routing), a standard method for network load balancing. Essentially, a router connected to multiple gateways (for instance, multiple ISPs) can balance outgoing connections to enhance the total available bandwidth. This solution is applicable when multiple WireGuard interfaces to the same destination exist, and all the routes towards these interfaces have the same cost. Consequently, traffic is distributed among the various interfaces according to configurable rules, the simplest of which is to use the hashing of the 5-tuple (hashing can be configured at Level 3 or Level 2). To prevent out-of-order packets, ECMP hashing is performed on a per-flow basis; all packets with identical header fields always hash to the same next hop. If applied to WireGuard, this solution is simple (a single command is sufficient to enable it) and resilient (if one tunnel fails, the route remains valid as long as at least one tunnel is operational).

Another solution is BONDING[27]. This solution achieves the same result as ECMP but is more complex. The Linux bonding driver provides a method for aggregating multiple network interfaces into a single logical "bonded" interface. This method is more complex than ECMP because bonding allows for aggregating interfaces at the L2 level, but the WireGuard interface is an L3 interface. In other words, WireGuard interfaces are associated only with an IP address but do not have a MAC address, preventing them from being aggregated at the L2 level. To solve this issue, it is necessary to build a GRETAP interface (or an equivalent variant) on top of the WireGuard one. GRETAP operates at OSI Layer 2, encapsulating an L2 packet (for example, Ethernet) inside an L3 packet (for example, IP).

The use of a GRETAP interface introduces additional challenges. For instance, there is a reduction in the MTU and an associated decrease in performance for two reasons: the overhead is increased, and it is necessary to pass through two levels of encapsulation. Specifically, the resulting MTU for a GRETAP interface built over a WireGuard interface using IPv4 is:

$$
\begin{aligned}
&1500\,\mathrm{B} - 20\,\mathrm{B(IPv4)} - 8\,\mathrm{B(UDP)} - 32\,\mathrm{B(WireGuard)}\\
&- 20\,\mathrm{B(IP\ Gretap)} - 4\,\mathrm{B(GRE)} - 18\,\mathrm{B(Ethernet)} = 1398\,\mathrm{Byte}
\end{aligned}
$$

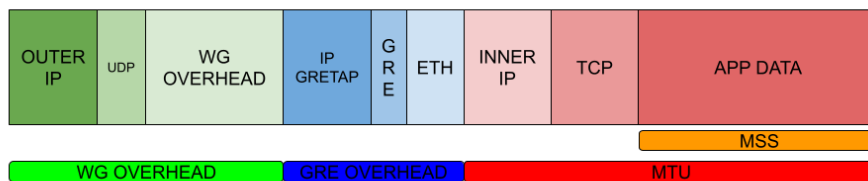

Figure 8.1: Overhead and MTU size when using GRETAP interface over Wireguard interface

Another challenge is that in this scenario, it is more difficult to perform GSO in the WireGuard interface, leading to a further decrease in performance. This is because the GRETAP interface cannot send a large packet over the WireGuard interface but is limited to sending only one packet at a time.

The overarching idea is to develop a method for splitting sessions in the various tunnels based on one of these two techniques.

- **Dynamic Tunnel Management:**
  A mechanism to manage the creation of tunnels and their distribution across different cores could boost efficiency. This mechanism should have the ability to relocate a tunnel if its assigned core is engaged by another resource-intensive process, prevent tunnel overlap (i.e., multiple tunnels on the same core), and allocate the ideal number of tunnels to match the current number of sessions.

- **Hybrid Mechanism that Integrates Wireguard Vanilla and Wireguard Inline:**
  WireGuard Vanilla continues to be the ideal solution for managing elephant flows. To boost performance and adaptability, a mechanism capable of determining the optimal moment to switch from Vanilla to Inline could ensure peak performance across all traffic patterns

- **Userspace Implementation:**
  Given the time needed for the kernel version to become available and distributed, one approach could be to incorporate the modifications introduced in WireGuard Inline into one of the userspace versions discussed in Section 4.1.3. The feasibility of applying the Inline version to the userspace variant of WireGuard should be evaluated. If viable, due to the complexity of the task and the architectural differences between userspace and kernel space, it's estimated that a minimum of six months would be required to implement these changes in one of these versions.

Bibliography

[1] *Kubernetes Documentation*. Accessed: July 15, 2024. 2024. URL: `https://kubernetes.io/docs/home/` (cit. on p. 3).

[2] Marco Iorio, Fulvio Risso, Alex Palesandro, Leonardo Camiciotti, and Antonio Manzalini. «Computing Without Borders: The Way Towards Liquid Computing». In: *IEEE Trans. Cloud Comput.* 11.3 (July 2023), pp. 2820–2838. URL: `https://ieeexplore.ieee.org/abstract/document/9984946` (cit. on pp. 6, 47).

[3] *Liqo*. Accessed: July 15, 2024. 2024. URL: `https://docs.liqo.io/en/latest/index.html` (cit. on p. 6).

[4] *Submariner documentation website*. Accessed: 2024-07-15. URL: `https://submariner.io/getting-started/architecture/` (cit. on p. 7).

[5] Vinod Kataria and Sreekanth Krishnavajjala. *Scaling VPN throughput using AWS Transit Gateway*. `https://aws.amazon.com/blogs/networking-and-content-delivery/scaling-vpn-throughput-using-aws-transit-gateway/`. 2020 (cit. on p. 9).

[6] Xiaobo Sherry Wei and Praveen Vannarath. «Systems and methods for improving packet forwarding throughput for encapsulated tunnels». US11716306B1 (cit. on p. 10).

[7] *Aviatrix Documentation*. Accessed: 2024-07-15. 2024. URL: `https://docs.aviatrix.com/previous/documentation/latest/planning-secure-networks/insane-mode-about.html` (cit. on p. 10).

[8] *Scaling in the Linux Networking Stack*. Accessed: 2024-07-11. URL: `https://www.kernel.org/doc/Documentation/networking/scaling.txt` (cit. on pp. 15, 18, 19).

[9] *NAPI*. Accessed: 2024-07-11. URL: `https://docs.kernel.org/networking/napi.html` (cit. on p. 16).

[10] Jason A Donenfeld. «WireGuard: Next Generation Kernel Network Tunnel.» In: *NDSS*. 2017, pp. 1–12 (cit. on pp. 22, 33, 36).

[11] *WireGuard: Fast, Modern, Secure VPN Tunnel*. Accessed: 2024-07-11. URL: `https://www.wireguard.com/` (cit. on p. 22).

[12] Jason A. Donenfeld. «WireGuard Linux Kernel Integration Techniques». In: *Proceedings of Netdev 2.2*. Seoul, Korea, Nov. 2017. URL: https://www.netdevconf.org/2.2/papers/donenfeld-wireguard-talk.pdf (cit. on pp. 26, 43).

[13] Lukas Osswald, Marco Haeberle, and Michael Menth. «Performance Comparison of VPN Solutions». In: *ITG Workshop on IT Security (ITSec) - (2.-3.4.2020)*. May 2020. URL: https://tobias-lib.ub.uni-tuebingen.de/xmlui/bitstream/handle/10900/100430/performance_comparison_of_vpn_solutions.pdf?sequence=1&isAllowed=y (cit. on p. 34).

[14] *Internet Protocol*. RFC 791. Sept. 1981. DOI: 10.17487/RFC0791. URL: https://www.rfc-editor.org/info/rfc791 (cit. on p. 36).

[15] Christopher A. Kent and Jeffrey C. Mogul. «Fragmentation Considered Harmful». In: *Proc. SIGCOMM '87*. Digital Equipment Corporation Western Research Lab. Oct. 1987, pp. 390–401. URL: https://dl.acm.org/doi/pdf/10.1145/205447.205456 (cit. on p. 36).

[16] J. Mogul and S. Deering. *Path MTU Discovery*. Internet Requests for Comments. RFC. Nov. 1990. URL: https://datatracker.ietf.org/doc/html/rfc1191 (cit. on p. 36).

[17] Herbert Xu. *GSO:Generic Segmentation Offload*. LWN.net. 2006. URL: https://lwn.net/Articles/188489 (cit. on p. 40).

[18] Jonathan Corbet. *JLS2009:Generic Receive Offload*. LWN.net. 2009. URL: https://lwn.net/Articles/358910/ (cit. on p. 41).

[19] Tom Herbert. «UDP Encapsulation in Linux». In: *Proceedings of netdev 0.1*. Ottawa, On, Canada, Feb. 2015. URL: https://people.netfilter.org/pablo/netdev0.1/papers/UDP-Encapsulation-in-Linux.pdf (cit. on p. 41).

[20] Willem de Bruijn and Eric Dumazet. «Optimizing UDP for content delivery: GSO, pacing and zerocopy». In: *Linux Plumbers Conference*. 2018. URL: http://vger.kernel.org/lpc_net2018_talks/willemdebruijn-lpc2018-udpgso-paper-DRAFT-1.pdf (cit. on p. 42).

[21] *Tailscale website*. Online. URL: https://tailscale.com (cit. on p. 43).

[22] Maxim Krasnyansky, Maksim Yevmenkin, and Florian Thiel. *Universal TUN/TAP device driver*. Kernel.org. 2002. URL: https://docs.kernel.org/networking/tuntap.html (cit. on p. 43).

[23] Jordan Whited and James Tucker. *Userspace isn't slow, some kernel interfaces are!* Accessed: July 6, 2024. 2022. URL: https://tailscale.com/blog/throughput-improvements (cit. on p. 44).

[24] Jordan Whited. *Surpassing 10Gb/s over Tailscale*. Accessed: July 6, 2024. 2023. URL: https://tailscale.com/blog/more-throughput (cit. on p. 45).

[25] GovanifY. *WireGuard and the Linux Networking Subsystem*. GovanifY. Aug. 2018. URL: https://govanify.com/post/gsoc-wireguard (cit. on p. 45).

[26] Dmitry Duplyakin et al. «The Design and Operation of CloudLab». In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. July 2019, pp. 1–14. URL: https://www.flux.utah.edu/paper/duplyakin-atc19 (cit. on p. 46).

[27] *Linux Ethernet Bonding Driver HOWTO*. Accessed: July 11, 2024. 2011. URL: https://www.kernel.org/doc/Documentation/networking/bonding.txt (cit. on p. 63).