Revisiting WireGuard for Line-rate, Scalable Tunneling

Revisiting WireGuard for Line-rate, Scalable Tunneling

Mirco Barone, Davide Miola, Federico Parola and Fulvio Risso
Department of Computer and Control Engineering
Politecnico di Torino
Torino, Italy
Emails: {mirco.barone, davide.miola, federico.parola, fulvio.risso}@polito.it

***Abstract*—Despite widespread adoption, the WireGuard tunneling mechanism available in the Linux kernel is unable to provide high-speed connectivity in a site-to-site setup when leveraging a standard single-tunnel configuration. In fact, its capability to scale with the number of available CPU cores is limited, even in the presence of a software architecture that is intrinsically parallel.**

This paper proposes multiple techniques to increase the throughput of the WireGuard technology. We show how greater control over the scheduling of WireGuard tasks enables performance optimizations such as NUMA awareness, in both single- and multi-tunnel setups. Finally, we further improve the scalability when leveraging multiple tunnels by proposing a custom *Inline* architecture tailored to this configuration. This architecture shows an almost 2x performance improvement over a multi-tunnel deployment of vanilla WireGuard, and supports 18x times the throughput of a single tunnel setup on our machines.

***Index Terms*—WireGuard, Tunneling, Multi-core scalability**

I. Introduction

WireGuard [1] is recognized as one of the leading tunneling technologies in Linux, thanks to its simple design and excellent kernel integration. Despite its extensive adoption, it struggles to provide high-speed connectivity between two sites when a standard single-tunnel configuration is adopted. In fact, WireGuard's performance does not scale well with the number of CPU cores, even though its software architecture is designed to be parallel.

In this paper we investigate the main causes of performance degradation of the Linux kernel implementation of the WireGuard protocol by testing custom patches to the upstream implementation of its kernel module. Our findings highlight that careful selection of the CPU cores allocated to the various stages of the WireGuard packet processing pipeline can help alleviate strain on the system and increase throughput, especially on Non-Uniform Memory Access (NUMA) systems where the default choice of scaling packet en/decryption across all available cores is shown to be detrimental to performance. Additionally, we examine strategies for effectively scaling WireGuard in multi-core architectures; we first note that — despite the capability to parallelize encryption and decryption stages — single tunnel throughput remains constrained by the serial per-tunnel stages in the processing pipeline. Hence, we attempt to spread flows over multiple tunnels to overcome this limitation. Our analysis reveals how simply leveraging multiple tunnels can end up not scaling at all, due to a subtle "black hole" condition related to the use of the standard softirq-based NAPI. We address this constraint by enabling the threaded NAPI on WireGuard interfaces, however, despite being able to leverage all the resources of our nodes, the approach still shows far from ideal scaling characteristics. To push things further, we propose a modified architecture which — for each flow — handles all WireGuard stages inline, in a single-threaded processing context, thus eliminating the costs of task and cache synchronization. This improved architecture, tailored for multi-tunnel support, shows an almost 2x performance improvement over a multi-tunnel deployment based on the vanilla WireGuard implementation, as well as being able to support 18 times the throughput of a single tunnel setup on our machines.

The remainder of this paper is organized as follows. Section II provides a background of the WireGuard architecture. Section III analyzes performance and possible improvements when leveraging a single tunnel to interconnect two sites, while Section IV explores the possibility of improving multi-core scalability leveraging multiple tunnels. We present related work in Section V and conclude the paper in Section VI.

This paper builds on our previous work [2], incorporating new improvements and experimental results regarding NUMA awareness and the management of encryption cores in WireGuard.

II. Background

WireGuard sets up a virtual network device [3] that can manage multiple tunnels, each connecting to a different peer. The host's routing table directs packets needing encapsulation to the WireGuard interface, which then encrypts them, determines the correct peer, and pushes the necessary tunnel headers. When a WireGuard-encapsulated packet is received, it is stripped of its outer headers, decrypted, and reinserted into the network stack by the WireGuard interface.

The upcoming sections explain the processes involved in both directions, focusing on how encapsulation and decapsulation are handled by two gateway nodes in a site-to-site setup.

A. Encapsulation

Traffic requiring encapsulation may come from a local application socket or a network device, particularly when the

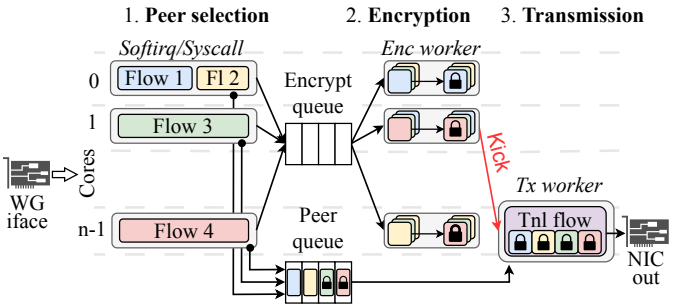

Fig. 1: WireGuard processing on CPU cores on the encapsulation side in the single tunnel scenario.

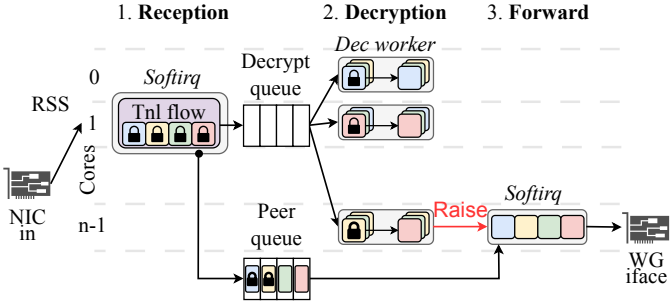

Fig. 2: WireGuard processing distribution on CPU cores on the decapsulation side in the single tunnel scenario.

host functions as a Virtual Private Network (VPN) gateway dedicated to forwarding traffic. In both cases, the Linux routing layer determines that the packet should be routed through the WireGuard interface by the destination address or other criteria. The encapsulation process is broken down into three main steps, as depicted in Figure 1. These are executed in different processing contexts and are parallelized in different ways.

1) **Peer selection.** This step occurs in the same context as the initial packet reception: *softirq* for packets coming from an interface, and *system calls* for packets generated by local processes. After selecting the peer and generating a nonce, a pointer to the unencrypted packet is placed into two separate queues. One is a per-device multi-producer multi-consumer (MPMC) ring buffer, designed to temporarily store all packets awaiting encryption associated with the same virtual WG device; this queue is drained by crypto workers running on all CPU cores (mapping of packets to cores in the pool is performed in round-robin). The other is a per-peer TX queue whose purpose is to retain the order of incoming packets: a dedicated kernel thread is spawned at peer creation time to drain this queue and perform transmission.
   Parallelism level: 1 CPU core per original flow.
2) **Encryption.** This step takes place within the core of the encryption pool selected in round-robin in the previous step. The worker retrieves packets from its input ring buffer (until empty), encrypts them, and marks them ready for transmission. Then, the TX worker is activated if not already running.
   Parallelism level: all the CPU cores of the node.
3) **Transmission.** Transmission involves extracting packets from the TX queue until encountering an unencrypted packet, or empty. Packets are encapsulated and sent out via a dedicated UDP socket, hence going through routing and the output stages of the kernel's network stack. WireGuard uses a per-peer TX thread to host this function.
   Parallelism level: 1 CPU core per peer.

B. Decapsulation

The three asynchronous steps depicted in Figure 2 take part in the decapsulation process:

1) **Reception.** Each WireGuard tunnel operates as a single UDP flow. Network Interface Cards (NICs) rely on mechanisms such as RSS (Receive Side Scaling) or flow steering rules to distribute received traffic to CPU cores at flow level; this restricts the reception stage's parallelism to a single core per tunnel. Data moves up the software stack in *softirq* context and eventually reaches the tunnel's UDP socket, where it is finally delivered to WireGuard. Similarly to the **Peer Selection** stage of the encapsulation pipeline, the packet is then enqueued to both a MPMC decryption ring and a RX buffer. One of the cores of the decryption pool is awoken in round-robin to handle the newly received packet.
   Parallelism level: 1 CPU core per tunnel.
2) **Decryption.** This step is symmetric to the encapsulation phase, with the selected worker draining the queue, decrypting each packet and updating its status accordingly.
   Parallelism level: all the CPU cores of the node.
3) **Forwarding (to application or on another interface).** Each peer is associated to a NAPI structure which establishes the appropriate callback function executed for packets received on the WireGuard interface. If the NAPI `poll()` callback of a peer is not already running on another core, the worker schedules it after packet decryption by raising a *softirq* on the current CPU core. The function drains all decrypted packets from the per-peer RX queue, handing them over to the upper layers of the network stack where they can be forwarded to an application or an output interface.
   Parallelism level: 1 CPU core per peer.

III. Single-tunnel deployments

A. Testbed

Our testbed (Figure 3) is composed of four machines running on CloudLab [4], equipped with dual Intel Xeon Silver 4314 16-core processors and interconnected through 100 Gbps links via Mellanox ConnectX-6 DX NICs. One machine acts as the *source* of the traffic and another as the *drain* of the

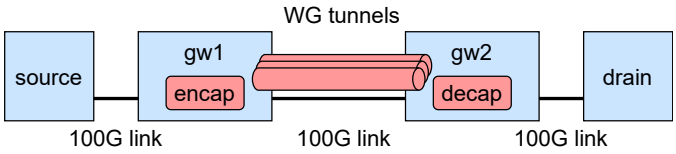

Fig. 3: Testbed setup for scalability tests.

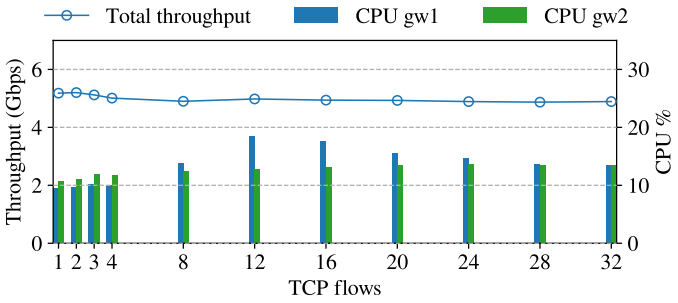

Fig. 4: Aggregate throughput and CPU usage for an increasing number of TCP flows in a single tunnel setup.

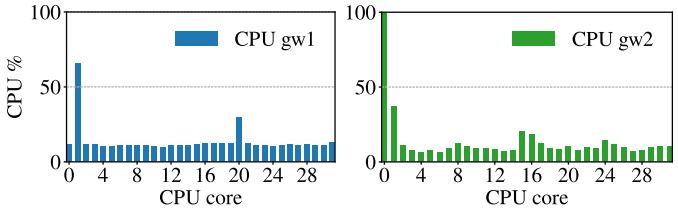

Fig. 5: Per-core CPU usage on *gw1* and *gw2* in the single tunnel setup when handling 31 flows.

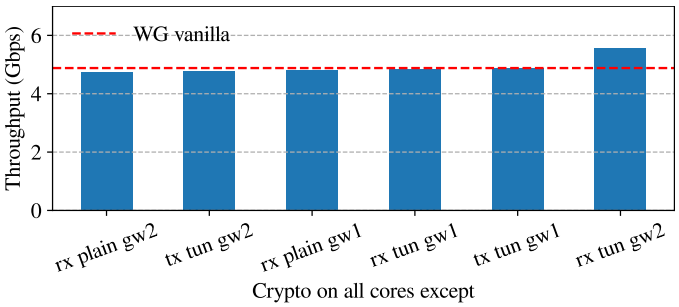

Fig. 6: WireGuard throughput when avoiding overlapping of en/decryption workers with other processing stages.

traffic/final receiver. In between them, *gw1* and *gw2* operate as VPN gateways by forwarding traffic through a number of established WireGuard tunnels.

We use *iperf3* to exchange TCP data between *source* and *drain*, and leverage multiple iperf3 client-server pairs to generate flows with different IP src/destination addresses. 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, we are able to focus on the impact of encapsulation on *gw1* and decapsulation on *gw2*, being the processing of ACKs mostly negligible. We disable hyperthreading to avoid sharing sibling hardware threads between critical components of the pipeline and ensure consistent measurements; we also disable dynamic frequency scaling and idle states on the CPU cores of all nodes for the same reason. All tests are repeated 10 times and the average is reported.

B. Single-tunnel performance

We start by analyzing the maximum performance achievable in the single tunnel scenario, typically used to interconnect two data centers. To this end, we generate an increasing number of TCP flows between *source* and *drain*, all encapsulated in a single tunnel established between *gw1* and *gw2*. To evaluate the maximum throughput in a best-case scenario, we configure flow steering rules on the *source*-facing NIC of *gw1* so that the reception of different flows happens on distinct cores, and doesn't overlap with the TX worker of the tunnel when possible (i.e., when `n_flows <= n_cores - 1`, on our setup, up to 31 flows). As depicted in Figure 4, the global throughput has virtually no dependency with the number of sessions encapsulated within the same tunnel. This is due to the fact that while multiple flows allow to leverage 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. Figure 5 shows the per-core CPU usage on the two gateways in the 31-flows test case (though a similar behavior applies to other flow counts). CPU core 0 of *gw2*, where the traffic of the tunnel is received (**Reception** stage), is the bottleneck, limiting all other stages.

C. Overlaps and NUMA Awareness

The vanilla version of WireGuard is not optimized to prevent overlap between the cores that handle the transmission and reception of tunneled traffic, and those responsible for encryption and decryption. In fact, encryption and decryption tasks are distributed across all cores in the system. Furthermore, this broad distribution can lead to performance degradation in NUMA architectures, where multiple CPUs reside within a single server and the cost of core-to-core and core-to-memory communication is non-uniform. The first way to improve single-tunnel throughput is to avoid this kind of overlap; the second is to enforce NUMA affinity for the various tasks.

1) Preventing overlaps: On each gateway, we allocate three stages of WireGuard processing to separate CPU cores: (*i*) reception of plain-text traffic (*rx plain*), (*ii*) transmission of tunneled traffic (*tx tun*), and (*iii*) reception of tunneled traffic (*rx tun*). In contrast, the transmission of plain-text traffic cannot be freely assigned, as it is scheduled by the decryption workers on the same CPU core(Section II-B).

In Figure 6, we exclude one of these cores at a time from the encryption and decryption worker pools, and record the TCP throughput. When lifting decryption duties from the *gw2* core responsible for the reception of encapsulated traffic (*rx tun gw2*), we observe a speedup of about 14%. Given that the

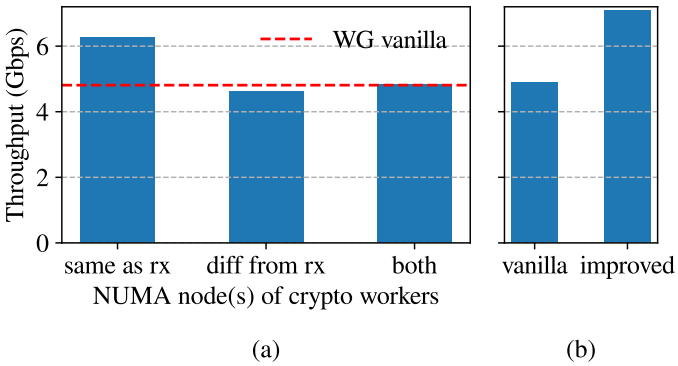

Fig. 7: (a) Effect of respecting NUMA affinity for the different pipeline stages on the two NUMA nodes of *gw2*. NUMA affinity is implicitly respected in *gw1*, too. (b) Wireguard Vanilla vs Improved Wireguard (NUMA aware + avoiding overlap) single tunnel comparison.

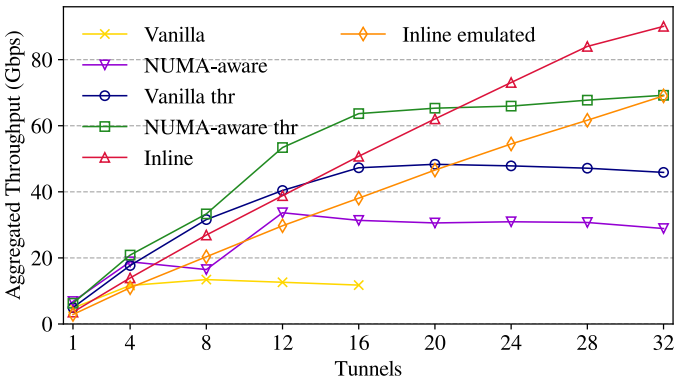

Fig. 8: Aggregate throughput of different WG configurations as the number of TCP flows and tunnels increases.

other configurations do not affect throughput, this confirms that such function is the bottleneck of the pipeline; conversely, avoiding overlapping in the other cores has almost no impact on performance.

2) NUMA affinity: WireGuard has no NUMA awareness, and all the tasks described above can freely span multiple NUMA nodes. We explored the effects of relocating various WireGuard stages—reception of plain-text traffic, encryption, and transmission on *gw1*; reception of encrypted traffic, decryption, and forwarding on *gw2*—across CPU cores to respect NUMA affinity. We concluded that acting on *gw1* yields only minor effects. This aligns with our earlier observation that, on our systems, the pipeline is bottlenecked by the packet reception stage of *gw2*. Conversely, applying NUMA-aware scheduling on *gw2* results in a 31% throughput increase compared to WireGuard Vanilla, as shown in Figure 7a. Notably, when RX and decryption are intentionally confined to opposite NUMA nodes, performance drops back to Vanilla levels.

It is possible to combine overlap prevention and NUMA affinity. In Figure 7b, the *improved* bar represents the throughput achieved by avoiding overlap on the core dedicated to receiving the encrypted traffic in *gw2*, combined with NUMA awareness. Overall, we achieve over 7 Gbps on a single tunnel, which is 45% higher than the baseline Vanilla configuration.

IV. Multi-tunnel deployments

A. Scaling to multiple tunnels

One way to enable parallelization of the per-peer stages in WireGuard is to leverage multiple tunnels and distribute traffic uniformly across them [5], [6]. To assess the maximum achievable throughput with this approach we repeat our experiment binding each iperf3 flow to a dedicated tunnel. Like previously, we configure steering rules on the inbound NICs of *gw1* and *gw2* so that each distinct flow is processed on a different core, and there is no overlap between RX flows processing and WireGuard TX workers. This allows us to scale up to 16 tunnels, after which our 32-cores servers require overlapping (with 16 tunnels, on *gw1* 16 cores perform flow reception and 16 cores tunnel transmission). The *Vanilla* line in Figure 8 shows that performance scalability is minimal. Throughput increases almost linearly up to 4 tunnels, after which it stabilizes around 13 Gbps with minor fluctuations.

B. The "Black Hole" condition

A deeper analysis of the multi-tunnel scenario allowed us to identify the source of its minimal scalability in how the NAPI processing of packets received by the WireGuard interface is handled. Each peer (tunnel) is associated with a single NAPI context and a matching NAPI `poll()` callback in charge of draining its queue. As per NAPI design, each NAPI context can only be scheduled on a single core at a time.

By default, the NAPI poll is run in *softirq* context, thus bound on a specific core until no more work is available (i.e., all packets have been extracted from the RX queue). Additionally, decryption workers will only raise a new *softirq* in case one was not currently in progress, otherwise preferring to rely on the existing instance. When traffic surges, this leads to a condition we refer to as "*Black Hole*", where multiple tunnels' NAPI poll functions end up being scheduled on the same core by chance, causing a collective slow down of the forwarding stage of all tunnels. In turn, this makes it impossible for the *softirq* to drain all the RX queues while traffic volume stays high, resulting in all WireGuard NAPI polls eventually converging onto a single core. Figure 9 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 also involved in the reception of traffic of the 8 tunnels pushing usage to around 40%. However, the bottleneck is represented by core 30 where all NAPI polls are gathered, saturating it. It is important to note that the bottleneck core can move across different tests, however the "*Black Hole*" condition consistently manifests in seconds after a traffic surge.

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

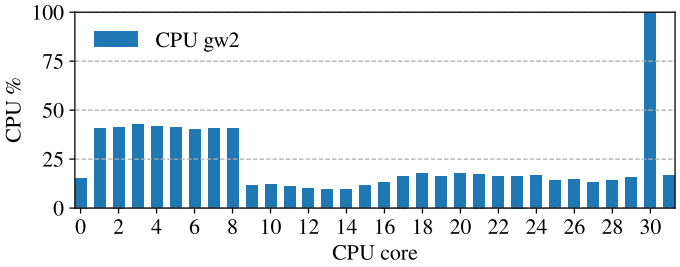

Fig. 9: Per-core CPU usage on *gw2* when handling 8 flows spread over 8 tunnels.

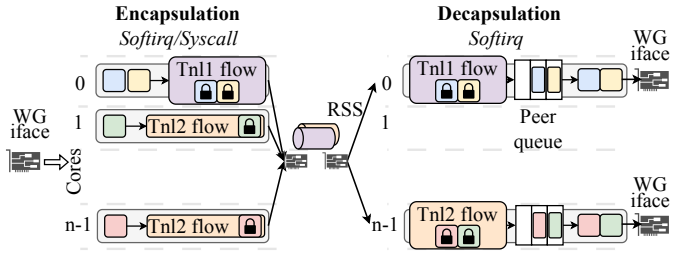

Fig. 10: WireGuard Inline processing distribution on CPU cores in a 2 tunnels setup (note that, even if not represented here, a single core might encapsulate multiple tunnels).

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 decryption occurs, but in our experiments this did not have a noticeable impact on performance. The *Vanilla thr* line in Figure 8 shows the results when enabling the threaded NAPI. This time performance scales with the number of tunnels and Wireguard can exploit almost all resources available on the two gateways, however, scaling is still far from linear and we are not able to saturate our 100 Gbps links.

C. WireGuard Inline

As noted, WireGuard employs a variety of different processing contexts (i.e., softirqs, syscalls and worker threads) and cores to run its pipeline. This however entails both synchronization overheads due to moving packets between them, as well as cache misses and context switches. This motivated the development of a new architecture for the kernel module tailored to multi-tunnel setups, "*WireGuard Inline*".

The main idea behind WireGuard Inline is to handle the whole lifecycle of a packet, including encryption and decryption, in a single context on a single core, relying solely on multiple tunnels to achieve multi-core scalability. This is accomplished by removing the en/decryption workers as well as the per-device queue used to distribute packets across them. In the encapsulation path, removing encryption workers prevents packet reordering, thus lifting the need for the TX worker and corresponding TX queue. As a result, in encapsulation, WireGuard Inline encrypts and transmits packets 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 contract that requires a list of packets to poll. Still, the NAPI is scheduled on the same *softirq* that receives UDP traffic of the tunnel. Overall, these modifications result in the architecture shown in Figure 10, which follows the same threading model as the IPsec kernel implementation.

We repeat our test with one TCP flow per tunnel for WireGuard Inline. On each gateway, we have two softirqs for each tunnel, one receiving clear-text traffic from *source* (TCP payload)/*drain* (ACKs) and one to handle tunneled packets. We schedule both on the same core, so each tunnel occupies only a single core, and pin the tunnels to the available CPUs. *Inline* numbers in Figure 8 show a remarkably linear trend for throughput, which exceeds 90 Gbps when spreading traffic over 32 tunnels and using all available CPU cores. This result is 2× the maximum throughput achieved with WireGuard Vanilla using threaded NAPI, and 18× that obtained with a single tunnel (Figure 4). In contrast, when few tunnels are used (up to 12 in our systems), our modifications lead to lower overall speeds compared to WireGuard Vanilla with threaded NAPI, as parallelism in en/decryption is no longer exploited. The inline WireGuard code is available at [7]. Additionally, we introduced a flag in the interface to toggle between the inline and vanilla versions, facilitating potential adoption.

While leveraging WireGuard Inline yields great results, the approach is somewhat disruptive. It requires deep changes to the internal architecture of the protocol, which makes its adoption in the mainline kernel module less likely. We assess whether it is possible to achieve similar benefits relying solely on the CPU core locality of tunnels (i.e., scheduling all the stages of a tunnel on a single core), without modifying the software architecture (i.e., removing the use of workers and rings to exchange packets). To this end, we start from vanilla WireGuard, but, for each tunnel, force all of its stages (rx, en/decryption, and tx) to run on the same core on both machines. The throughput we obtain with this approach (*Emulated inline* in Figure 8) follows a similar trend to the inline version but is shifted downwards. This underlines how the original architecture is not suitable for inline execution, and architectural changes are needed to leverage its full potential.

D. Scaling WireGuard NUMA-aware

On the line of exploring less disruptive solutions, we also investigate the scalability properties of NUMA-aware WireGuard, which showed promising results for single-tunnel deployments in Figure 7a. This means that for each involved tunnel, all operations are performed within the same NUMA node, both in *gw1* and *gw2*. Our methodology involves spreading the tunnels among the two NUMA nodes of *gw1* and *gw2* together with their en/decryption pools and transmission threads. Each tunnel retains full NUMA affinity, but we are able to saturate both CPUs by simply scaling up the number of connections. Past 16 tunnels, overlaps are necessary to support more tunnels.

In Figure 8, the *NUMA-aware* series represents the throughput of this system with the traditional softirq-based NAPI. The staggered trend highlights how the "*Black Hole*" condition

is affecting the tunnels in NUMA node 0 first (around 4 tunnels), and then those in NUMA node 1 (after 12 tunnels). When switching to threaded NAPI, however, the *NUMA-aware thr* line boasts the best performance up to about 20 tunnels, after which it loses to WireGuard Inline. Notably, it always outperforms WireGuard Vanilla with threaded NAPI (*Vanilla thr*). Overall, while not as CPU efficient as our Inline architecture (NUMA-aware saturates all CPU cores at around 65 Gbps, while, at the same throughput, Inline still has available CPUs to scale), NUMA-aware represents a low-disruption alternative to boost WireGuard performance in a multi-tunnel setup.

V. Related work

Different works analyze encrypted tunneling protocols and attempt to improve their performance.

In [8], authors focus on the IPsec protocol and suggest different functional and performance improvements. For multi-core scaling, the paper discards the possibility of allocating multiple tunnels to reduce the management overhead and misconfiguration risks. Instead, it suggests adding information about internal flows to the encapsulation header so NICs can steer traffic from the same tunnel on different cores (a similar approach is proposed for WireGuard in [9]). We didn't experience excessive operational complexity in our multi-tunnel experiments, and this technique doesn't modify WireGuard encapsulation, achieving better compatibility with existing implementations.

[10] compares the WireGuard, IPsec, and OpenVPN tunneling protocols, and identifies WireGuard as the most promising architecture performance-wise. Authors also propose architectural changes to improve performance, however, their analysis only focuses on the encapsulation side of tunnels, while we identified decapsulation as the source of most limitations.

Finally, UDP GSO and GRO [11], which allow handling larger UDP datagrams, are actively been investigated as a way to save CPU cycles spent in per-packet processing and improve WireGuard performance [9], [12]. These techniques are orthogonal to the ones presented in this paper and can be integrated to further increase throughput.

VI. Conclusions

In this paper we performed a thorough analysis of the performance of the WireGuard kernel implementation, highlighting its limitations and the underlying architectural details that determine them. In the common case of single tunnel deployments, our results show that control over the distribution of WireGuard tasks onto physical CPU cores can be invaluable to avoid undesirable overlaps. Furthermore, in NUMA systems, the cost of distributing crypto stages over multiple sockets proved to be overshadowing any benefit from using a wider worker pool. Overall, we were able to achieve a substantial 45% uplift in effective throughput with a customized thread allocation scheme.

We show that these considerations also apply to multi-tunnel configurations, where NUMA awareness more than doubles overall performance compared to a vanilla deployment on our systems. Moreover, we propose an alternative architecture, *WireGuard Inline*, that unlocks fully linear scaling of performance with the number of available CPU cores over multiple tunnels.

Acknowledgment

This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART"), and the European Union's Horizon Europe research and innovation programme under grant agreement No 101070473, project FLUIDOS (Flexible, scaLable, secUre, and decentralIseD Operating System). Finally, Davide Miola's scholarship is part of the project PNRR-NGEU which has received funding from the MUR - DM 117/2023.

References

[1] J. A. Donenfeld, "Wireguard: Next generation kernel network tunnel," in *NDSS*, 2017, pp. 1–12.

[2] M. Barone, D. Miola, F. Parola, and F. Risso, "Achieving linear cpu scaling in wireguard with an efficient multi-tunnel architecture," in *Proceedings of Netdev 0x18*, Santa Clara, California, July 2024. [Online]. Available: https://netdevconf.info/0x18/docs/netdev-0x18-paper23-talk-paper.pdf

[3] J. A. Donenfeld, "Wireguard linux kernel integration techniques," in *Proceedings of Netdev 2.2*, Seoul, Korea, November 2017. [Online]. Available: https://www.netdevconf.org/2.2/papers/donenfeld-wireguard-talk.pdf

[4] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14. [Online]. Available: https://www.flux.utah.edu/paper/duplyakin-atc19

[5] X. S. Wei and P. Vannarath, "Systems and methods for improving packet forwarding throughput for encapsulated tunnels," U.S. Patent US11 716 306B1, 2021.

[6] V. Kataria and S. 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.

[7] M. Barone, "Wireguard inline code," https://github.com/MircoBarone/wireguard-linux, 2024.

[8] M. Pfeiffer, F. Girlich, M. Rossberg, and G. Schaefer, "Vector packet encapsulation: the case for a scalable ipsec encryption protocol," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ser. ARES '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3407023.3407060

[9] D. Borkmann, A. Protopopov, and M. Pumputis, "Wireguard and gro? improving wireguard performance," in *Linux Plumbers Conference*, 2024. [Online]. Available: https://lpc.events/event/18/contributions/1968/

[10] M. Pudelko, P. Emmerich, S. Gallenmüller, and G. Carle, "Performance analysis of vpn gateways," in *2020 IFIP Networking Conference (Networking)*, 2020, pp. 325–333.

[11] W. De Bruijn and E. Dumazet, "Optimizing udp for content delivery: Gso, pacing and zerocopy," in *Linux Plumbers Conference*, 2018. [Online]. Available: http://oldvger.kernel.org/lpc_net2018_talks/willemdebruijn-lpc2018-udpgso-paper-DRAFT-1.pdf

[12] Tailscale, "Surpassing 10gb/s over tailscale," April 2023. [Online]. Available: https://tailscale.com/blog/more-throughput