# Achieving Linear CPU Scaling in WireGuard with an Efficient Multi-tunnel Architecture

**Mirco Barone, Davide Miola, Federico Parola, Fulvio Risso**
*Politecnico di Torino*
Turin, Italy
Emails: mirco.barone@studenti.polito.it, {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 routing through 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.

In this paper we investigate the multi-core scalability properties of WireGuard, identifying current limitations and proposing an improved design that aids effective scaling, reaching a near-linear throughput increase depending on the number of involved CPU cores. Furthermore, we propose a multi-tunnel approach to parallelize stages of the WireGuard pipeline limited to a single core per tunnel and propose a modified architecture tailored to multi-tunnel support. 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.

## Keywords

WireGuard, Tunneling, Multi-core scalability

## Introduction

WireGuard [2] is one of the most common tunneling technologies used in Linux, thanks to its simplicity and excellent integration in the kernel. Despite its widespread adoption, it is unable to provide high-speed connectivity between two sites when adopting a standard single-tunnel configuration; this represents a significant limitation when a secure, high-capacity interconnection is required. In fact, the capability to scale WireGuard performance with the number of available CPU cores is somehow limited, even in presence of a software architecture that is intrinsically parallel.

In this paper we investigate the multi-core scalability properties of WireGuard, identifying current limitations and proposing an improved design that aids effective scaling, reaching a near-linear throughput increase depending on the number of involved CPU cores. We first analyze the architecture of a single tunnel setup, underlining how — despite its capability to parallelize encryption and decryption stages — the presence of serial per-tunnel logic creates a bottleneck on the use of additional resources. Hence, we attempt to spread flows over multiple tunnels, in order 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 overcome this limitation 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 signal 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 18x times the throughput of a single tunnel setup on our machines. Despite limitations in handling elephant flows, the presented approach represents a starting point for further discussion and a first step towards a more scalable WireGuard architecture.

## Background

Wireguard operates by creating a virtual network device [1]. 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 sections detail the processing involved in these two directions, when the encapsulation and decapsulation are performed by two gateway nodes in a site-to-site configuration.

### Encapsulation

Traffic needing encapsulation can originate from a local application socket or a network device, in case this host behaves as a VPN gateway performing only forwarding. 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. Encapsulation is composed of the three main steps depicted in Figure 1, which operate in different execu-

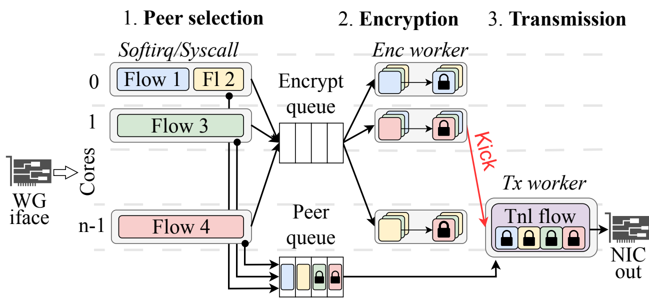


Figure 1: WireGuard processing on CPU cores on the encapsulation side in the single tunnel scenario.

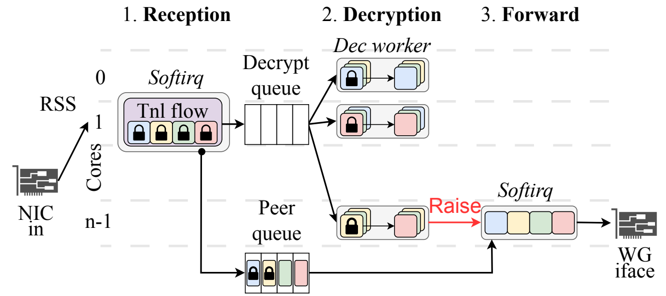


Figure 2: WireGuard processing distribution on CPU cores on the decapsulation side in the single tunnel scenario.

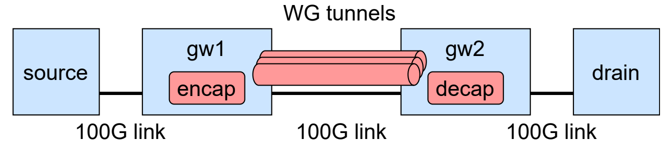


Figure 3: Testbed setup for scalability tests.

tion contexts and are subject to different parallelization approaches:

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 a nonce computed, the packet is tagged as unencrypted and 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.
*Parallelism level: 1 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.
*Parallelism level: 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 always executes on the same CPU core, selected during the handshake phase. The worker pulls packets from the serial queue until it is empty or until an unencrypted packet is encountered. These packets are encapsulated and transmitted on the physical NIC, following all necessary routing steps.
*Parallelism level: 1 CPU core per peer.*

### Decapsulation

The three asynchronous steps depicted in Figure 2 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 flow steering rules. These techniques allow for traffic steering with a maximum granularity of flow level. Given that a single tunnel between two WireGuard devices corresponds to a single UDP flow, the processing of this step (Reception) is confined to one core per tunnel, even if it encapsulates multiple flows. Traffic flows up the TCP/IP stack and is directed to a UDP socket registered by WireGuard. Upon socket reception, 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.
*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, 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 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.*

## Scaling WireGuard

### Testbed

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

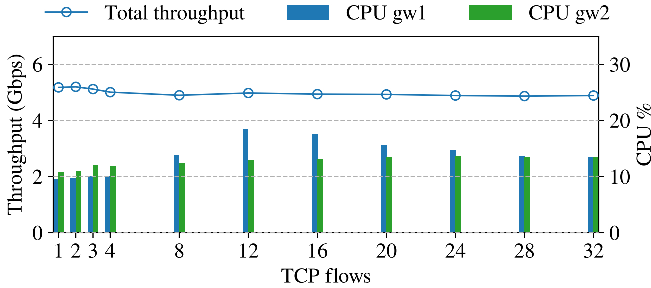


Figure 4: Aggregate throughput and CPU usage for an increasing number of TCP flows in a single tunnel setup.

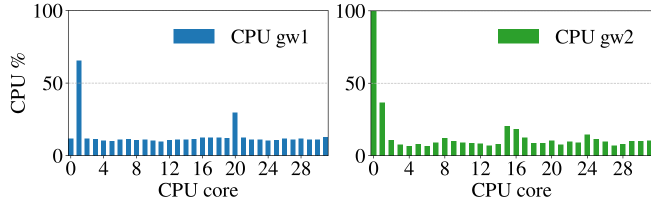


Figure 5: Per-core CPU usage on *gw1* and *gw2* in the single tunnel setup when handling 31 flows.

of the traffic/final receiver. In between them, the remaining machines operate as VPN gateways (*gw1* and *gw2*), solely involved in forwarding the traffic on a variable number of WireGuard tunnels.

We use iperf3 to exchange TCP data between *source* and *drain*. To generate multiple flows without being limited by the single-core performance of the *source* and *drain* machines we leverage multiple iperf3 client-server pairs. 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 only on the impact of the encapsulation procedure on *gw1* and decapsulation on *gw2*, being the processing of ACKs negligible. We disable hyperthreading and idle states on all nodes to avoid inconsistent measurements. All tests are repeated 10 times and the average is reported.

## Single tunnel evaluation

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 remains constant, unaffected by 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.

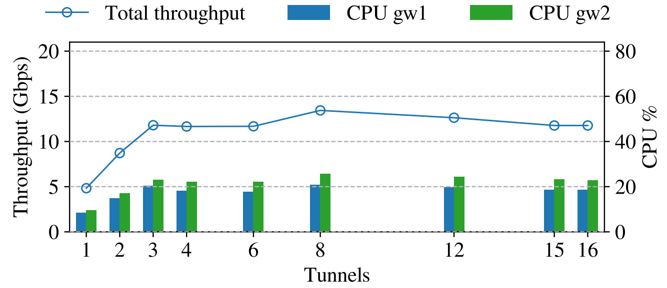


Figure 6: Aggregate throughput and CPU usage for an increasing number of TCP flows and corresponding WireGuard tunnels.

## Scaling to Multiple Tunnels and the Problem of the NAPI

One solution to enable parallelization of the per-peer stages in WireGuard is leveraging multiple tunnels and distributing traffic uniformly across them [5, 4]. To assess the maximum throughput achievable with this approach we repeat the previous test assigning each iperf3 flow to a different tunnel. As in the previous test, we configure steering rules 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 RX 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). Numbers in Figure 6 show how we achieve minimal scalability. As the number of tunnels increases up to 3, we observe a linear trend in throughput. However, beyond this point, the throughput stabilizes around 12/14 Gbps, despite increasing the number of tunnels. This value is accompanied by minor fluctuations.

A deeper analysis allowed us to identify the source of this behavior 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 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. Since `napi_poll()`s of different tunnels move to different

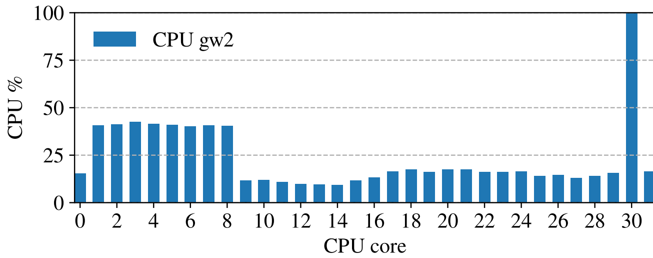


Figure 7: Per-core CPU usage on *gw2* when handling 8 flows spread over 8 tunnels.

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 can turn the core into a "black hole". Figure 7 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.

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. **Vanilla** numbers in Figure 8 show 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, the scaling is still far from being linear and we are not able to saturate our 100 Gbps links.

## Wireguard Inline

We analyzed the architecture of WireGuard to overcome the limitations identified in the previous section. To allow scaling parallelizable stages (en/decryption) while keeping other stages serial, processing is performed in a pipeline of different contexts (i.e., syscalls/softirqs and workers), potentially executed on different cores. This however entails synchronization overheads needed to exchange packets among contexts through rings. Additionally, contexts running on different cores lead to cache misses when exchanging a packet, while contexts on the same core require additional context switches. Handling core-context mapping is currently not

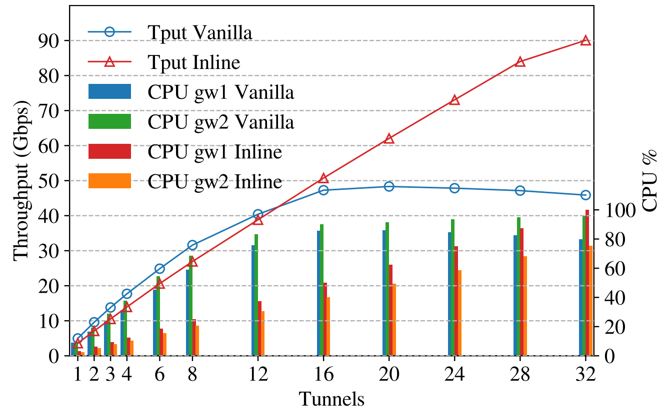


Figure 8: Aggreagate throughput and CPU usage for an increasing number of TCP flows and corresponding WireGuard tunnels, when using WireGuard Vanilla (with threaded NAPI) or Inline.

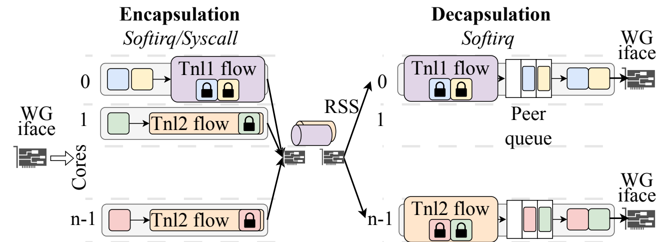


Figure 9: WireGuard Inline processing distribution on CPU cores in a 2 tunnels setup (note that, even if not represented here, a single core might encapsulate multiple tunnels).

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 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, we developed a new version of the WireGuard module, 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 scale on multiple cores. This is achieved 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 the reordering of packets within original flows, allowing the removal of the TX worker and its corresponding per-peer queue, previously needed to preserve order in transmission. 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 mechanism that requires a list of packets to poll. Still, the NAPI is sched-

uled on the same softirq that receives UDP traffic of the tunnel. Overall, applying these modifications we achieve the architecture represented in Figure 9, 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 non-tunneled traffic from *source*/*drain* and one receiving tunneled traffic. We schedule both on the same core, so each tunnel occupies a core, and spread the tunnels on the available cores. **Inline** numbers in Figure 8 show a trend in the throughput for WireGuard Inline very close to linearity. The throughput reaches approximately 90 Gbps when 32 tunnels are exploited and all available cores are fully utilized. When leveraging less than 12 tunnels the Vanilla version of Wireguard achieves a higher throughput, albeit at a much higher CPU cost (more than 2x). This underlines how our modifications decrease the throughput for a single tunnel, as parallelism in en/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 in case we have few elephant flows, since each flow is limited to a single tunnel, and hence cannot be parallelized. In this context, we envision a hybrid approach, which dynamically shifts each tunnel between Vanilla and Inline processing modes, according to the amount of traffic it is experiencing. We leave the design and evaluation of this approach as future work.

In addition to its multi-tunnel performance improvement, the Inline architecture also simplifies allocation of tunnels to CPU cores (e.g., by managing NIC's RSS and flow steering rules), avoiding interference among tunnels.

## Conclusions

In this paper we performed a thorough analysis of the multi-core scalability properties of the WireGuard kernel implementation, highlighting its limitations and the architectural choices that prevent it from scaling efficiently. We proposed an alternative approach to scaling, WireGurard Inline, based on a multi-tunnel approach and a simplified single-threaded architecture of the protocol. Our experiments highlight how our solution is much more CPU efficient than the vanilla implementation and allows to double the throughput that can be handled by our VPN gateways. Despite not being a one-size-fits-all solution, due to limitations in handling elephant flows, our approach provides an interesting starting point for further discussion and represents a first step towards a more scalable WireGuard architecture.

## Acknowledgments

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

Finanziato dall'Unione europea NextGenerationEU | Ministero dell'Università e della Ricerca | Italiadomani

has received funding from the MUR - DM 117/2023.

## References

[1] Donenfeld, J. A. 2017a. Wireguard linux kernel integration techniques. In *Proceedings of Netdev 2.2*.

[2] Donenfeld, J. A. 2017b. Wireguard: Next generation kernel network tunnel. In *NDSS*, 1–12.

[3] Duplyakin, D.; Ricci, R.; Maricq, A.; Wong, G.; Duerig, J.; Eide, E.; Stoller, L.; Hibler, M.; Johnson, D.; Webb, K.; Akella, A.; Wang, K.; Ricart, G.; Landweber, L.; Elliott, C.; Zink, M.; Cecchet, E.; Kar, S.; and Mishra, P. 2019. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 1–14.

[4] Kataria, V., and Krishnavajjala, S. 2020. Scaling vpn throughput using aws transit gateway. `https://aws.amazon.com/blogs/networking-and-content-delivery/scaling-vpn-throughput-using-aws-transit-gateway`

[5] Wei, X. S., and Vannarath, P. Systems and methods for improving packet forwarding throughput for encapsulated tunnels.