Performance Analysis of VPN Gateways

Maximilian Pudelko[1,2], Paul Emmerich[1], Sebastian Gallenmüller[1], Georg Carle[1]
[1]*Technical University of Munich, Department of Informatics, Chair of Network Architectures and Services*
[2]*Open Networking Foundation*

***Abstract*—VPNs play an important role in today's Internet architecture. We investigate different architectures for software implementations of VPN gateways and their effect on performance. Our case study compares OpenVPN, Linux IPsec, and WireGuard. We also implement a WireGuard-compatible VPN benchmarking example application with three different software architectures inspired by the evaluated open-source solutions. Our implementation allows benchmarking of individual effects and optimizations in isolation.**

We find that WireGuard is the most promising software VPN implementation from an architectural viewpoint. Our implementation of WireGuard's pipeline architecture on top of DPDK achieves 6.2 Mpps and 40 Gbit/s, the fastest of all evaluated VPN implementations. We find that the main bottleneck for scaling software VPNs are data structures and multi-core synchronization – a problem that can be tackled with an architecture based on pipelining and message passing.

***Index Terms*—VPN, Wireguard, OpenVPN, IPsec, Linux**

I. Introduction

Today's world became increasingly more connected over the last decade. New paradigms like IoT and technologies like the upcoming 5G standard for mobile cellular communication will only increase the number of connected devices. The introduction of cloud computing brought additional challenges. Existing on-premises networks have to be extended and connected to virtual cloud networks, e.g., via a VPN gateway. Especially these site-to-site use-cases place huge requirements in terms of throughput on VPN solutions. Open-source VPN implementations are often not fast enough to handle multi-gigabit links at line rate on commodity off-the-shelf (COTS) hardware. Network administrators instead rely on dedicated hardware VPN solutions, which are often expensive and not auditable from a security standpoint.

This paper evaluates the three most popular open-source software VPN solutions IPsec, OpenVPN, and WireGuard available on Linux. Our benchmark suite determines if these VPN implementations are fast enough to be deployed on 40 Gbit/s links with COTS hardware. Further, we identify performance bottlenecks by profiling the code.

To gain further insights into the typical operation steps of a VPN we develop our own VPN implementation that allows us to investigate if achieving a higher performance is possible. By using our own bare-bones versions, we reduce external influences and are able to measure the performance hazards more accurately. Our VPN implementation consists of three implementations that mimic the three identified popular architectures to compare them in an isolated setting.

ISBN 978-3-903176-28-7 © 2020 IFIP

Our key contributions in this paper are:

- Performance evaluation of software VPN gateways
- Analysis of the effect of software architecture
- Fast implementations of a custom VPN gateway for all popular software architectures

II. Background and Related Work

On the most basic level, VPNs virtually connect two or more hosts that reside in different logical or physical networks, as if they were on the same link or in the same subnet. To a native application, there is no difference if it connects to a host over a VPN or directly. What raises VPNs beyond basic tunnel protocols are the cryptographic guarantees for confidentiality and integrity among others.

VPNs can be deployed in two different scenarios: client-server setup with multiple clients connecting to a server or in a site-to-site setup connecting two locations with a secure tunnel that handles many independent connections. The latter is more interesting from a performance standpoint, because it represents a potential choke point in the network and is subject to more traffic than a client-server setup. Secure datacenter interconnects require a high throughput and often rely on specialized hardware VPN appliances. Their high cost and black-box nature make them undesirable to some and creates a niche for open-source software VPN solutions that run on COTS hardware. This paper focuses on high-throughput site-to-site setups, realized with common VPN software.

In their extensive report from 2011 Hoekstra et al. [8] evaluated the performance of OpenVPN in gigabit networks, with a focus on bandwidth-specific bottlenecks. They focused on throughput measurements in terms of Mbit/s instead of million packets per second (Mpps) using a default MTU of 1500 bytes. For secure configurations (AES-128-CBC with HMAC-SHA1) they measured a maximum throughput of around 270 Mbit/s.

A more recent comparison (2017) by Lackovic et al. [10] measures the impact of AES-NI support on encryption speeds. In their benchmarks they find a significant speedup of 40% and 60% for IPsec AES, and a smaller increase for OpenVPN at 10% to 16%. Their findings about AES-NI show the same trend as other results, but remain a little lower. Raumer et al. [13] report a 100% to 320% increase for IPsec on Linux, depending on packet size (0.48 Gbit/s to 0.94 Gbit/s for 64 bytes and 1.33 Gbit/s to 4.32 Gbit/s for 1462 bytes). Additionally, Lackovic evaluates the scaling opportunities of the VPNs and concludes that IPsec is more scalable than OpenVPN.

Sadok et al. [14] investigated the efficiency of receive-side scaling (RSS) when the traffic only contains a small number of concurrent flows. Their measurements on a backbone link show a median of only 4 flows and 14 inside the 99th percentile in time windows of 150 μs. With RSS this leads to an uneven resource utilization, as only a few cores are handling all traffic.

At the time of writing, there is very little academic research published about WireGuard. In its initial publication [3], Donenfeld includes a short performance comparison with IPsec and OpenVPN. In this benchmark, WireGuard outperformed both other implementations and is solely able to saturate the 1G link. Apart from this data point, there are no extensive performance evaluations published, especially not on fast networks. We use a 40 Gbit/s network for the evaluation here.

III. Evaluation of Open Source VPNs

All benchmarks were run on a dual Intel Xeon E5-2630 v4 CPU (total of 40 cores including hyper threading) with 128 GiB of memory and two Intel XL 710 40 Gbit/s NICs. Spectre and Meltdown mitigations were not present in the used kernel on Ubuntu 16.04. All test load was generated with custom MoonGen [4] scripts running on a separate server. The load generator customizes the packets IP addresses and sends them to the DuT. The DuT is configured to encrypt and forward all incoming packets back on the second port.

A. *Varied Parameters*

a) Number of Flows: A flow is usually classified by a 5-Tuple of L3 source and destination address, L4 protocol, and L4 source and destination ports. By variating the number of used IP source and destination addresses, we can control the number of flows in the traffic and therefore the number of end hosts we want to emulate.

b) Packet Rate: The packet rate describes how many packets per second are processed, often given in Mpps. In our benchmark setup we can set a given input rate to the DuT and observe the output rate of the processed packets. By increasing the input rate we get a complete picture of how the device behaves under different load conditions.

c) Packet Size: Packet size is an important factor in measurements, as clever variation of the sizes allows to gain insights about the inner working of the device under test. This is possible by splitting the costs of handling a packet into two categories.

The static costs are independent of packets sizes and always occur equally. Things like routing lookups and counters fall into this category: Resolving the destination interface for an IP packet does not depend on its payload. The variable costs on the other hand are determined by its size (e.g., encryption) and scale accordingly.

B. *Baseline: Linux Network Stack & Router*

All three tested VPNs rely on the kernel network stack for low-level functionality. In benchmarks the costs of passing a packet through the network stack is sometimes not separable from the costs of the application. To classify the performance of the VPNs more accurately, we need some form of baseline performance to determine if a bottleneck is caused by the application or roots in the Linux network stack. While others already published performance measurements for Linux, we need comparable values by testing the router and the VPNs against exactly the same traffic patterns, on the same hosts, and on the same network equipment. Routing is a fundamental operation in a L3 VPN and often offloaded to the kernel. For the following benchmark we configured one manual route on the DuT, so that it forwards the packets back to the load generator. The main routing table contained six routes.

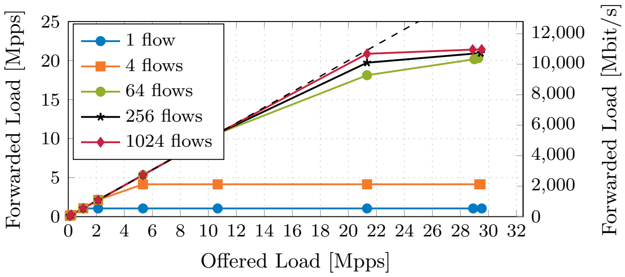


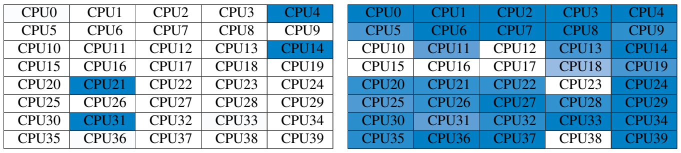
Fig. 1: Linux router forwarding rates with different number of flows and 64 byte packets

Figure 1 shows the measured rates of the Linux router with traffic of fixed sized packets and a varying number of flows. The dotted line marks the ideal case; each incoming packet is forwarded. The x-axis displays the applied load to the DuT, while the left and right y-axis measure the traffic that was received back on the load generator.

It can be seen that the kind of traffic has a measurable impact on the routing level already: traffic consisting of just one or a small number of flows is not as easily processed as traffic with many flows. For a single flow of 64 bytes the forwarding rate is at best 1 Mpps, while with 1024 flows it increases steadily to around 21.3 Mpps. Although the forwarding rates plateau at a certain level, they do not decrease after that. This means this system is not susceptible to a trivial DoS attack where an attacker sends millions of packets to lower performance for everyone else disproportionally.

The test was repeated with varying packet sizes but we found the size to have no impact on the throughput unless hitting the line rate limit of 40 Gbit/s. This is expected for a router, as it — unlike a VPN gateway — only looks at the first few bytes in the Ethernet and IP header.

1) Packet Distribution: The first step when investigating the slow forwarding performance with traffic consisting of few flows is to look at the CPU utilization of the DuT. Figure 2 shows heat maps of the CPU utilization per core for two selected traffic patterns. NICs use a hash function over the packet headers to distribute incoming packets to the Rx-queues in a technique called receive-side scaling (RSS), this effectively divides packets per flow. So we need at least 40 flows to fully use this CPU. The random aspect of hashing means that we need more flows to actually hit all cores, explaining why there is a slight performance gain when going beyond 64 flows.



(a) 4 flows

(b) 64 flows

Fig. 2: CPU utilization of the Linux router during peak load

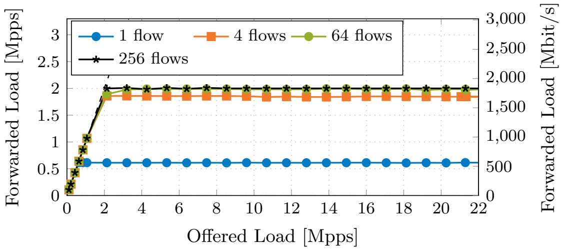


Fig. 3: Processing rates of IPsec with 64 byte packets

2) Source Code Analysis — Spin locks: We also record which tasks the CPUs perform to see where the CPU time is spent. We found multiple spin locks in the send path that have to be taken. CPU profiling shows that between 10% and 20% of the total CPU time (depending on the number of threads) is spent in spin locks. Note that these locks are uncontended in a proper configuration when only doing routing with at least as many CPU cores as queues.

These baseline measurements tell us that the absolute best case we can achieve is around 21 Mpps or 14 Gbit/s (with framing, 11 Gbit/s data rate) with minimum-sized packets and the full line rate of 40 Gbit/s with a packet size of 256 byte.

C. Linux kernel IPsec

IPsec is an IETF standard for network layer security in IP networks. It is engineered to academic perfection, follows a layered approach and performs key exchange by a completely separate protocol. We only study the encryption part skipping the key exchange by using static keys. We chose ESP tunnel mode with the RFC4106 AES-GCM symmetric cipher, which is a hardware-accelerated (AES-NI) AEAD cipher. IPsec encryption is performed in the transform (xfrm) layer of the Linux network stack. It runs as part of the software interrupt handler in ksoftirqd threads. These threads run on the same core where the original hardware interrupt arrived. This means that packet encryption happens on the same core to which the packet got delivered.

Figure 3 shows the measured forwarding rates for IPsec. For single-flow traffic we see the same drawback as in the baseline measurements. All packets are delivered to a single core, thus overloading it. For more flows (and hence cores) the rate increases up to a hard limit of 2 Mpps.

Profiling shows that the bottleneck is multi-core synchronization of the security association (SA) data structure which is protected by a spin lock. Since multiple CPUs can process packets governed by the same SA in parallel, each CPU first has to acquire a spin lock. A site-to-site VPN connecting only two subnets might have only a single SA that is a de-facto global spin lock. Figure 4 shows how performance increases as we add SAs. For the first three added SAs we see a linear increase of the forwarding rate to 3 Gbit/s. After that point, each SA adds marginally less up to the highest measured rate of 5.27 Mpps and 2.7 Gbit/s. This sub-linear growth shows, that there are other bottlenecks in the send path, which are not optimized by this distribution scheme. IPsec achieves around 23% of the baseline performance because it fails to scale to multiple cores.

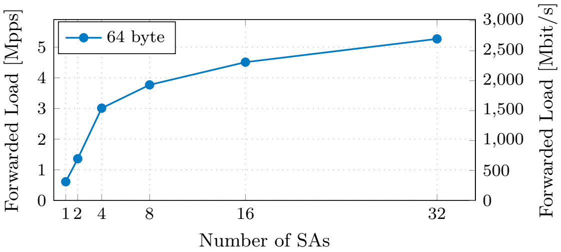


Fig. 4: IPsec forwarding rate, 256 flows, 64 byte packets and multiple SAs

D. OpenVPN

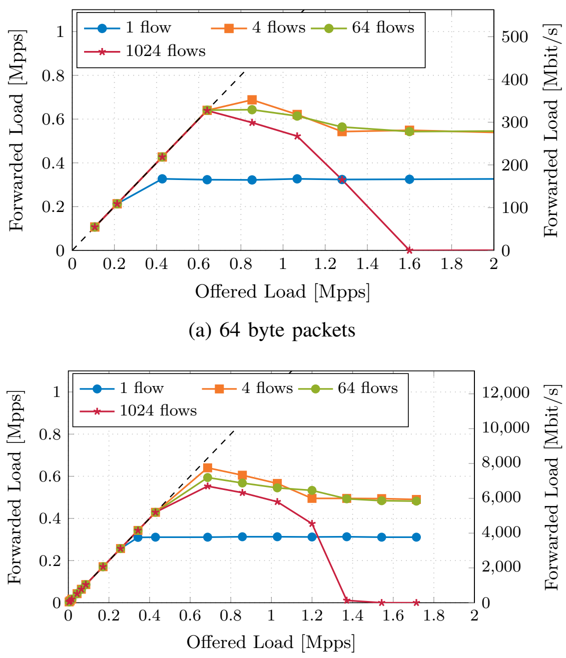
OpenVPN is a cross-platform, open-source VPN suite defining its own protocol and wire format. Its core components are the same across all platforms, the networking layer differs between platforms. On Linux, the socket API and virtual TUN/TAP devices are used. This approach allows operation as a pure user-space process without the need for kernel modules or custom drivers. Due to the lack of a cross-platform interface for threads, OpenVPN historically is single-threaded. Modern AEAD ciphers are not available, so we select AES-CBC without message authentication to achieve performance similar to a modern cipher (but without its security guarantees).

The performance is very low with only 0.12 Mpps regardless of the number of flows. (We benchmarked other configurations as well, this was the fastest result). These findings are consistent with the related work and the single-threaded architecture. Profiling the whole system shows that the time is mostly spent on context switching between user-space and kernel. Time spent on encryption is negligible with hardware acceleration (AES-NI) enabled. Disabling AES-NI significantly impacts performance further, showing that hardware acceleration is important for AES encryption.

The only way to scale OpenVPN is by running multiple instances of it. Doing so indeed scales perfectly linearly up to 16 instances. The other 4 physical CPU cores are dedicated to the kernel network stack. Further instances hit hyper-threading CPUs on our test setup. This achieves a throughput of 2.5 Mpps while fully using 20 CPU cores.

E. WireGuard

WireGuard is a new L3 VPN proposed by Jason Donenfeld in 2017 [3]. It defines its own protocol and comes with a reference implementation as a Linux kernel module. The work



(a) 64 byte packets

(b) 1514 byte packets

Fig. 5: Encryption rates of WireGuard at different packet sizes and varying number of flows

presented here was performed in 2019 with this kernel module before WireGuard was available in the mainline kernel. The integration of WireGuard into the kernel was completed while this article was under peer review for publication.

WireGuard's protocol was developed from scratch, based on best cryptographic practices and using the newest ciphers. Contrary to committee-guided protocols like IPsec, WireGuard is strongly opinionated on certain topics and includes radical ideas. It does away with cryptographic agility by locking in on a single AEAD cipher and authentication algorithm with no feature negotiation: ChaCha20-poly1305. Backwards compatibility is explicitly missing: handshake and key derivation include a hash of the protocol version number, so two different implementations will derive distinct keys, making them permanently non-compatible. These measures heavily incentivize keeping the software up-to-date and prevent degradation attacks found in SSL [11].

The kernel module implementation introduces virtual interfaces to integrate WireGuard into the network stack. This benefits adoption and usability, as all standard interface tools, like iproute2, work as expected. Compared to the xfrm layer in which IPsec operates, this design features a better user experience. Packets going out of a WireGuard interface are guaranteed to be encrypted, while received ones are guaranteed to be authentic and form known peers. Overall, WireGuard strives to be a as-simple-as-possible, very secure and hard-to-get-wrong VPN that is usable by everyone.

Figure 5 shows the basic measurement with different packet sizes. Multiple patterns can be observed. Again, we see that the number of flows in the traffic matters for performance. With just one flow the RSS configuration of underlying

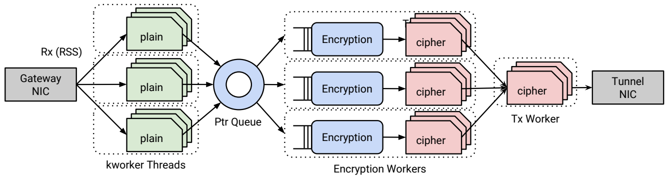


Fig. 6: WireGuard threading model for encryption

Linux network stack limits the forwarding rate to 0.32 Mpps. Adding more flows elevates this and increases the peak rate to 0.65 Mpps, until it plateaus at 0.57 Mpps for 4 and 64 flows and high load. Most interesting is the case of 1024 flows. Contrary to the other VPNs where this traffic yields the best results, WireGuard struggles with increasing load and effectively stops processing any packets at a load of 1.4 Mpps. We analyze this in Section III-E1.

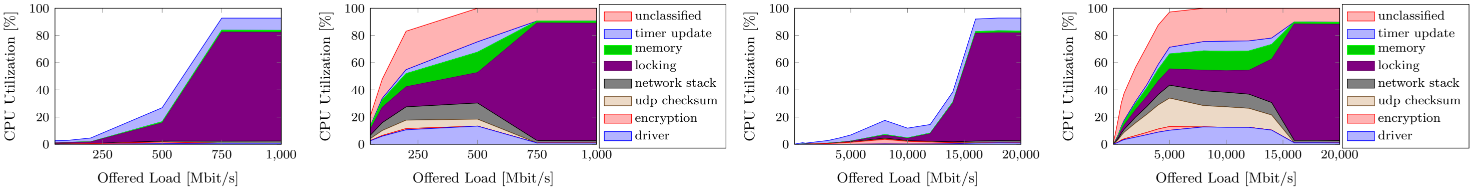
Another observation is that the size of the processed packets has no impact on performance. Subfigures 5a and 5b show nearly the same patterns with peaks and plateaus at the same points, despite comparing the smallest to the largest packets. This indicates, that encryption is not a bottleneck in WireGuard, otherwise we would see lower rates for larger packets.

The observed performance pattern is tied to the threading model of WireGuard, as shown in Figure 6. It consists of three stages that perform the VPN operations.

kworker threads serve the gateway NIC and receive packets from it. After determining through a routing lookup that a packet is eligible for processing, they transmit it through a virtual WireGuard device enqueuing them in a per-device pointer queue as a work item. The creation of this work item involves more than just memory allocation. The kworker already checks if the source and destination IPs are allowed, looks up the peer state with the encryption keys and increments the nonce per packet. These additional steps performed in this early stage explain why the performance rates of WireGuard in the single flow case are lower than the 1 Mpps of the Linux baseline benchmark. With only a single flow, these operation must be performed by one kworker and are thus effectively not parallelized.

In the default configuration of Linux there is one RSS queue and kworker per CPU core. The work items are picked up from the queue by the encryption workers, which are spawned on all online cores at module activation. As the encryption key and nonce are bundled with each packet, they just have to perform the actual encryption, without touching any peer state and thus operate lock-free. Processed packets are handed over to the Tx worker. There is only a single Tx worker per peer. It collects all encrypted packets and transmits them via the normal UDP socket API. With three pipeline stages, this design has three potential choke points, which we investigate further.

1) Locking and back-pressure: Figure 7 shows on which operations the different workers spend the most cycles on when traffic with 1024 flows is processed. This kind of traffic is chosen because it can cause a total overload of WireGuard.



(a) 64 byte packets, kworker & encryption worker

(b) 64 byte packets, Tx worker

(c) 1514 byte packets, kworker & encryption worker

(d) 1514 byte packets, Tx worker

Fig. 7: CPU utilization of WireGuard with 1024 flow traffic

Subfigures 7a and 7c show CPUs on which a kworker and encryption worker is running. They are aggregated because WireGuard spawns the encryption workers on every core, without a way to prevent this. Subfigures 7b and 7d visualize the Tx worker on separate cores.

The dominant factor is locking. On the kworker and encryption worker, driver or encryption operations are run less than 10% of the time. On the Tx worker a variety of tasks are done, while not overloaded. Memory is allocated and freed, the UDP checksums of the packets are calculated and the internal WireGuard timers get updated. The dominance of locking can be explained by analyzing the source code. Spin locks have to be taken in pipeline stages. More importantly, they are used to implement mutual exclusion in the pointer rings interconnecting the stages.

The kworker code paths contain multiple locks. In the `xmit` function an incoming packet is checked for validity and moved into a queue (guarded by a spin lock) associated with the peer. In a site-to-site setup with only one peer and thus one queue, this lock is contended by all kworkers of the system and has to be taken and released for every single packet.

The encryption and Tx worker are lock-free, apart from the aforementioned pointer rings. Together these sources explain the baseline locking load in Subfigures 7a and 7c where the offered load is below 500 Mbit/s and 12000 Mbit/s. The large increase with more load after these points comes from a design flaw. Under overload a kworker will keep inserting the new packets into the crypt and even tx queues, while dropping older ones. This lack of back-pressure increases contention on the locks of the pointer queues even more. Under extreme overload the kworker monopolize all locks, as seen in the graphs, so that over 80% of the time is spent busy-waiting for spin locks.

IV. MoonWire: Design and Implementation

We implement MoonWire, our VPN sandbox, based on the learnings from the open-source architectures. We aim for a modular, easy-to-change software that allows fast iteration and experimentation. With MoonWire it is possible to rapidly try out different approaches to specific sub-problems and to observe the performance shifts.

For the VPN protocol and data framing the WireGuard protocol is chosen due to its minimal design. We did not implement all features, we skipped the cookie response mechanism for DoS protection and re-keying timers. This has no impact on the measurements presented here but makes it unsuitable for production use (and it requires a small patch to WireGuard to make it compatible for benchmarking, this change has no effect on performance).

Previous research [1], [6] and our own measurements in Section III-B showed that the network and driver stack of the OS can be a limiting factor due to its generality and some architectural limitations. To work around this bottleneck and to gain a high configurability, MoonWire is based on the user-space packet processing framework DPDK. It comes with high-performance NIC drivers and an abundance of specialized data structures for packet processing tasks. Since DPDK applications are written in C/C++, they require a certain amount of boilerplate code which can be greatly reduced by using libmoon, a high-level Lua wrapper around DPDK [5]. The Lua code is not interpreted, but just-in-time compiled by the integrated LuaJIT compiler. It is a proven platform for packet processing related scripts [4], [12] and allows direct access to C ABI functions, i.e., the DPDK library. The libsodium library is used for all cryptographic primitives.

Since the evaluated state-of-the art VPN solutions made vastly different design choices, it makes sense to not concentrate on a single implementation. In the following variants we replicate seen designs with our MoonWire VPN kit and present a new one. Each variant is explained from the high-level design to the specific implementation details.

A. Data Structures

Since the high-level goals of a VPN is the same across the different variants, some algorithms and data structures are shared nearly verbatim between them.

1) Peer Table: To check if an incoming packet is to be processed, some form of lookup data structure is needed. Depending on the specification of the protocol, different options are possible. WireGuard identifies relevant packets by checking the destination IP address against the configured subnets of allowed IPs. IPsec is more complex in this regard and also allows filters on ports and source addresses. IP address lookup tables are an extensively studied field [2], [7], [15] with many established high-performance implementations. For MoonWire we rely on the LPM library of DPDK, which uses the DIR-24-8 algorithm.

2) Peer State: The peer state encapsulates all information necessary to send and receive packets from a remote VPN peer.

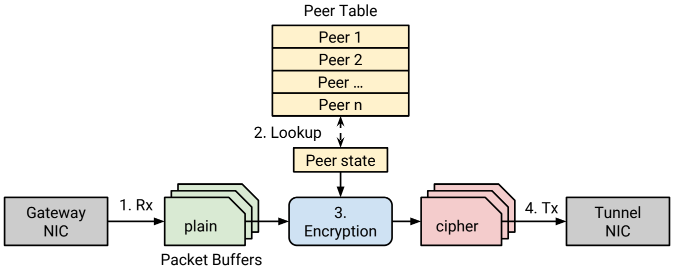


Fig. 8: Flowchart of MoonWire Variant 1

It must contain the cryptographic key material, the address of the remote endpoint, and the nonce counter. In IPsec this functionality is realized in security associations (SAs), WireGuard implements this with its `wg_peer` structure. While the values like the endpoint or keys are set once and then mostly read, the nonce counter is updated for every packet. For this to be thread-safe with multiple parallel workers, a lock can be added to the structure. Compared to a global or table-wide lock, this allows concurrent operation of workers, as long as they access different peer states.

B. Variant 1 - Naive Single Instance

The first variant is based on the design of OpenVPN and conceptually simple. The whole application consists of a single thread that completes all tasks of VPN operation.

In Figure 8 the different steps in the continuous main loop of Variant 1 are visualized. The thread starts by receiving incoming packets from the gateway interface in form of a batch. Each buffer is validated and the associated peer state is looked up in the peer table. If there is an open connection, the buffer is extended, encrypted and new IP headers are inserted. In the last step, all valid packets are sent and the remaining ones are freed.

Due to its single-threaded nature, there is no need for any kind of synchronization on the peer table or state. This variant does not suffer from averse traffic pattern like single-flow traffic. With only one thread, there are no benefits in using RSS and multiple queues. Therefore, uneven packet distribution cannot occur. On the contrary, such traffic could even be beneficial since repeated lookups of the same value utilizes the CPU caches better than different addresses which generate repeated cache misses and evictions.

On the downside, this variant shares its drawbacks with OpenVPN. Namely the total lack of horizontal scaling. Other cores can only be utilized by spawning multiple independent instances and fragmenting the routed subnets into smaller chunks.

C. Variant 2 - Independent Workers

Variant 2 takes the crude scaling attempts necessary for Variant 1 or OpenVPN and internalizes them. In summary, this version is most similar to IPsec in the Linux kernel.

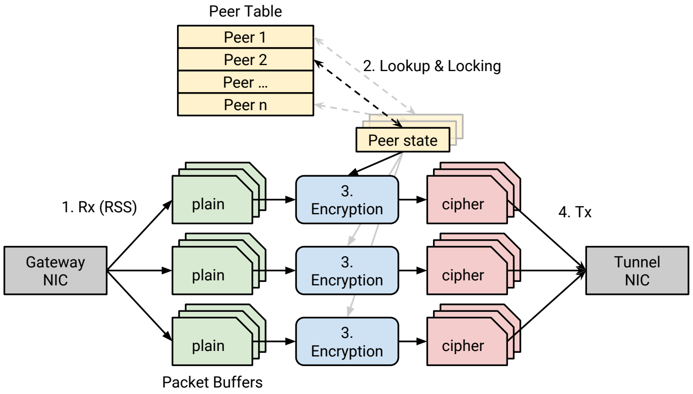


Fig. 9: Flowchart of MoonWire Variant 2

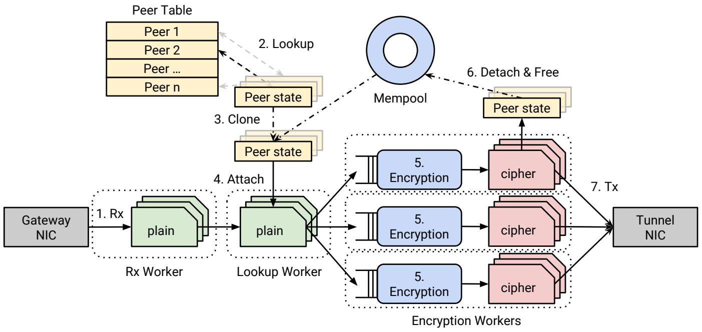


Fig. 10: Flowchart of MoonWire Variant 3

As seen in Figure 9, multiple workers are processing the packets in parallel, each with its own NIC queues. Each worker still completes the whole loop from reception, encryption to transmission. Incoming packets are distributed to multiple queues on the hardware level by the NIC through RSS. Since there are multiple threads accessing and modifying shared data, these accesses have to be synchronized to prevent corruption.

We use the classic `pthread_mutex` from the POSIX threading library and compare it to the `rte_spinlock` provided by DPDK, henceforth called mutex variant or spin lock variant respectively. The POSIX mutex internally relies on `futex` API of the kernel and is a general purpose lock. Spin locks, on the other hand, do not relinquish control to the scheduler but continuously keep trying. This trades latency for energy efficiency and leads to a slightly higher performance, when only a handful of threads are contending for a lock.

D. Variant 3 - Work Pipeline

In Variant 3 the best properties of the previous versions are taken, while avoiding their drawbacks. The peer table and state is still contained in and maintained by a single thread (therefore lock-free), while the encryption stage is parallelized across multiple workers. Instead of handing out mutable peer state references to the workers, the state is cloned and distributed in a message passing manner. This shared-nothing approach is conceptually similar to the design philosophies in Erlang [16] or the Message Passing Interface (MPI) library.

Figure 10 visualizes the complete setup. After a batch of packets has been received by the Rx worker, the lookup worker fetches the applicable state to a packet buffer as it is done in Variant 1. It then clones the entire state with all keys and endpoint information into a separate empty buffer, and attaches this buffer to the packet buffer. The parallel workers now find all information needed for encryption contained in the packet buffer and do not update any shared state. Since this crypto-args buffer is separate allocated memory, it has to be reclaimed after use. We use generic DPDK memory buffers for performance reasons.

Packet distribution between the stages happens through DPDK's `rte_rings`; lock-free, FIFO, high-performance queues with batched functions to reduce overhead. Distribution over the workers happens in a round-robin fashion to ensure equal load among them. We initially combined Rx and the Lookup worker. But profiling showed that a significant amount of time was spent on state cloning. We therefore factored packet reception out in the final design to give this task as much processing power as possible.

1) Back-pressure: In contrast to the single-stage setups in Variant 1 and 2 where each step is completed before the next is taken, Variant 3 decouples these from each other. Therefore, it is now possible to, e.g., receive more packets than can be encrypted. Noticing and propagating this information backwards is necessary to govern the input stages, prevent wasting of resources and reduce overload [9]. We implement an exponential back-off similar to ancient Ethernet. The benefits of using back-pressure are twofold. For one, it reduces energy usage by not wasting cycles on operations which results will be thrown away anyway. Secondly, it simplifies our analysis by making it easy to spot the bottlenecks in the pipeline. A stage running at 100% load, while its input stage is not fully utilized, is likely to be the constraining factor.

V. MoonWire: Evaluation

MoonWire allows us to systematically find the bottlenecks of the architecture by running the same benchmarks on different variations of the same code base.

A. *Variant 1 - Naive Single Instance*

Variant 1 is the most simple and straightforward of our designs, without threads, RSS, or other advanced features. We achieve 1.3 Mpps (10 times faster than OpenVPN). Increasing the packet size decreases throughput, i.e., encryption is the bottleneck. Profiling shows that 85% of the time is spent on encryption with small packets and 96% with 1514 byte packets, completely marginalizing any other costs.

B. *Variant 2 - Independent Workers*

Variant 2 solves the processing power bottleneck around encryption of Variant 1 by distributing the packets to multiple parallel workers in hardware (RSS). Consequently, this variant suffers from the same problems as the Linux router and IPsec, which also rely on RSS. Figure 11 shows the familiar picture: Few flows in the traffic result in uneven distribution and lower processing rates.

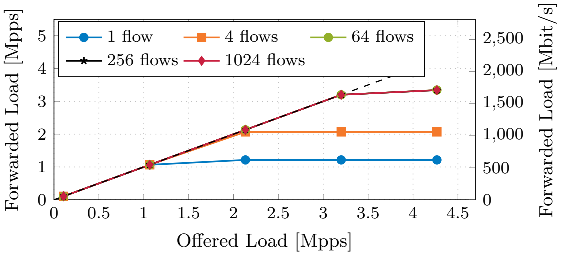


Fig. 11: Effects of RSS with 64 byte packets on MoonWire Variant 2 (mutex) with 4 workers

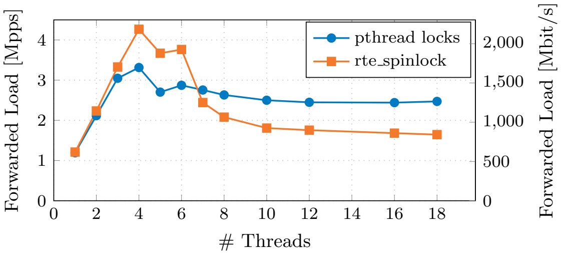
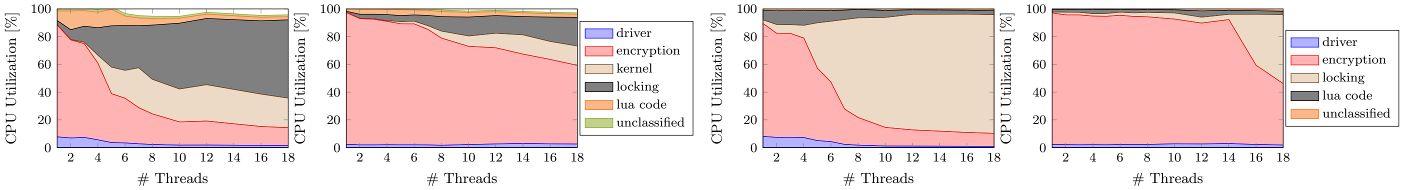


Fig. 12: Scaling capabilities of MoonWire Variant 2, comparing mutexes and spin locks

But for traffic with sufficient flows ($\geq$ 64), significant speedups can be measured, as shown in Figure 12. In these measurements, the number of threads is increased gradually and the individual peak rate is recorded.

But also the drawbacks of global locks can be observed again. Especially at smaller packet sizes and their inherent higher Mpps, the bottleneck becomes evident. For both kinds of locks, the peak rate is reached with 4 workers and decreases after this point.

Spin locks have a higher peak rate, but are also more susceptible to high contention. Looking at the CPU utilization plots in Figure 13 on the next page, we can see why Variant 2 only performs well with larger packets and at lower Mpps rates. We can see the general trend, that more threads lead to more time spent on locking and less on encryption. For 64 byte packets and high Mpps rates, a steep increase can be seen after 4 threads, mirroring our previous rate measurements. At larger packet sizes this effect occurs more delayed, but is still measurable. The spin lock implementation spends increasing amounts of time directly in the lock and unlock functions. For mutexes the locking category is split into two components: The user space side with the pthread and libc wrappers, and the kernel space part with the syscall overhead (cf. OpenVPN), the futex implementation and scheduling. Overall this variant demonstrates that this approach does not scale well beyond a few CPU cores.



(a) Mutexes, 64 byte packets (b) Mutexes, 1514 byte packets (c) Spin locks, 64 byte packets (d) Spin locks, 1514 byte packets

Fig. 13: CPU utilization of a worker thread in MoonWire Variant 2, mutexes vs. spin locks

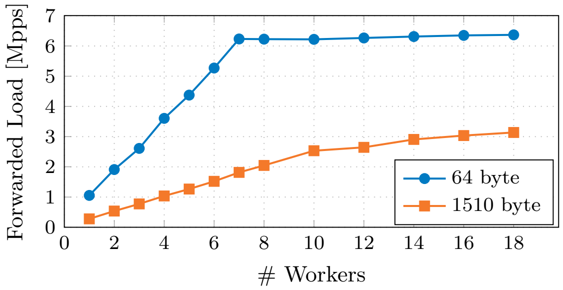


Fig. 14: Throughput scaling of MoonWire Variant 3 with encryption workers

C. Variant 3 - Work Pipeline

Variant 3 is our pipeline implementation. The incoming packets are processed in 3 stages and passed to different workers. Shared state is minimized and expensive synchronization constructs like locks are not used. Whenever function arguments are needed in later stages, they are attached to the packet buffer as messages.

In Figure 14 the scaling properties of this variant can be seen. It shows the encryption throughput for 64 and 1514 byte packets, depending on the number of encryption workers. The Rx and lookup stage is always handled by a single worker, as explained in Section IV, and not included in the worker count. For one to seven workers and small packets we measure a near linear scaling from 1.1 Mpps to 6.3 Mpps (3.2 Gbit/s). Adding more workers does not increase (but also not decrease) the rate further. This result is both our highest measured rate across all VPN implementations and also very stable. Compared to implementations with locks, Variant 3 does not crumble under overload. For 1514 byte packets, there is no plateau and the throughput increases up to 3.13 Mpps, where the 40 Gbit/s output link is fully saturated.

Figure 15 shows another benefit of not relying on RSS for packet distribution to workers: the number of flows in the traffic does not matter. Compared to Variant 2 where traffic with only a few flows lead to an under-utilization of some workers, here the rates are always the same.

Next we determine which stage is the limiting bottleneck. In the plots in Figure 16 we see the CPU utilization of each stage and for 64 and 1514 byte packets.

First, we focus on the results for smaller packets, which are in the first column (Subfigures 16a, 16c, 16e). It can be seen,

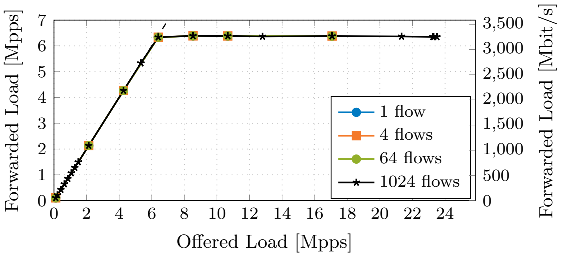
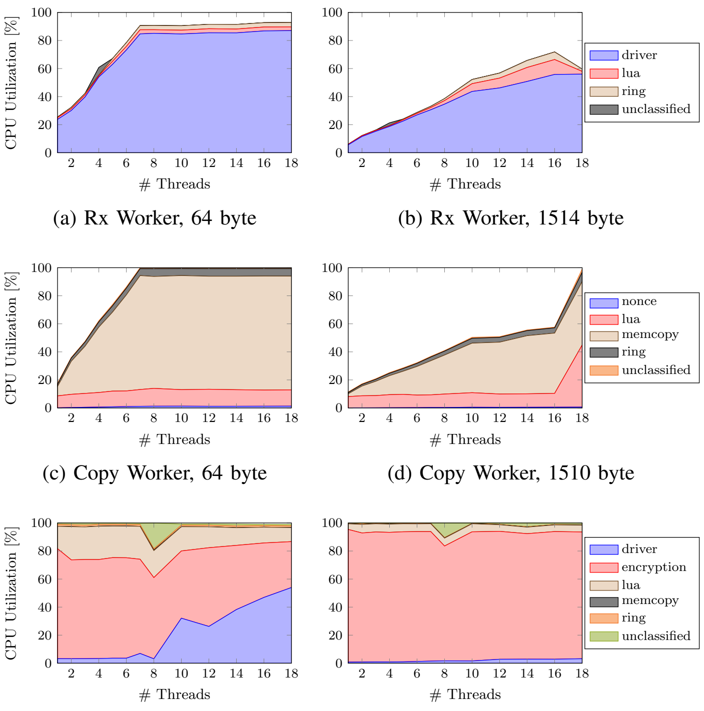


Fig. 15: Throughput of MoonWire Variant 3 depending on the number of flows



(a) Rx Worker, 64 byte (b) Rx Worker, 1514 byte

(c) Copy Worker, 64 byte (d) Copy Worker, 1510 byte

(e) Encryption Worker, 64 byte (f) Encryption Worker, 1514 byte

Fig. 16: CPU utilization in the pipeline stages of MoonWire Variants 3 with 64 and 1514 byte packets

that for smaller number of encryption workers, these are the bottleneck. The Rx worker is never running at full capacity, and the copy stage is fully utilized at 7 workers.

For large 1514 byte packets the picture is overall similar, but different in a few details. Neither the Rx worker, nor the copy stage is ever running at full capacity. Not a surprising result, since they only perform per-packet tasks and larger packets generally come at a lower rate. The encryptions workers however, are always at 100% utilization. Compared to 64 byte packets, a even larger fraction of the time (85%) is spent in encryption routines. The overhead from fetching packets from the rings and sending them is nearly negligible, making them quite efficient at their task. We expect that—given more NICs and cores—this can be scaled up further.

In conclusion, Variant 3 is our best implementation. It achieves the highest rates at all packet sizes, does not depend on RSS and still evenly distributes packets. The data structures used for, e.g., IP address lookups can be un-synchronized, allowing for simpler or faster designs than multi-threaded versions. We identified the message creation, more precisely, the memory copying, to be a limiting factor. This is the case, because our implementation clones the entire peer state, including seldom changed fields.

One drawback is packet reordering. With a round-robin distribution over workers, there are no guarantees regarding the order the packets come out of the DuT.

VI. Conclusion

We measured how the three open-source software VPN implementations IPsec, OpenVPN, and WireGuard perform in site-to-site setups on COTS hardware running Linux. With the results of our benchmarks we come to the conclusion, that none of them are currently fast enough to handle the amounts of packets in 10 or 40 Gbit/s networks. A considerable amount overhead stems from the underlying Linux network stack, on which all tested versions rely on.

With our MoonWire implementations we showed that bypassing the kernel and using the fast DPDK drivers is the first step towards higher performance. The single-threaded Variant 1 already improves performance by an order of magnitude. The next hurdle is the high processing power requirement of encrypting large amounts of data in real-time. This requires efficient exploitation of the modern many-core CPU architecture, which is a non-trivial problem. Using common strategies like mutual exclusion with locks does not scale well beyond a few cores, as seen in our benchmarks of IPsec and MoonWire Variant 2. Choosing the right cryptographic ciphers can also help. Modern AEAD ciphers like AES-GCM or ChaCha20-poly1305 improve over traditional encrypt-then-mac ciphers.

Data synchronization proved to be the third major area where performance is decided. We observed poor scaling of lock-based-synchronization in both Linux IPsec and WireGuard, where up to 90% of the time is spent in locking code. Investigations of the source code revealed code paths where one or multiple locks had to be taken for every handled packet. In MoonWire Variant 3 we did away with locking and implemented a work pipeline following a shared-nothing approach and message passing as means of inter-thread communication. Overall this approach yielded the best results with the highest measured processing rate of 6.3 Mpps (3.2 Gbit/s) for 64 byte packets. At 1514 byte packets we reached 3.13 Mpps and thus the line rate for 40 Gbit/s links.

Our implementation is a testbed for different software architectures, not a a VPN gateway meant for production use. We recommend WireGuard for production setups based on our investigation. Its architecture is similar to our proposed pipeline architecture and on the right track for a scalable modern VPN implementation and it already performs very well in many scenarios despite not being heavily optimized for performance yet.

Acknowledgments

This work was supported by the German-French Academy for the Industry of the Future and by the DFG-funded project MoDaNet (grant no. CA 595/11-1).

References

[1] Raffaele Bolla and Roberto Bruschi. Linux software router: Data plane optimization and performance evaluation. *Journal of Networks*, 2(3):6–17, 2007.

[2] Tzi-cker Chiueh and Prashant Pradhan. High-performance ip routing table lookup using cpu caching. In *INFOCOM'99*, volume 3, pages 1421–1428. IEEE, 1999.

[3] Jason A Donenfeld. Wireguard: next generation kernel network tunnel. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[4] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference (IMC)*, Tokyo, Japan, October 2015.

[5] Sebastian Gallenmüller, Dominik Scholz, Florian Wohlfart, Quirin Scheitle, Paul Emmerich, and Georg Carle. High-Performance Packet Processing and Measurements (Invited Paper). In *COMSNETS 2018*, Bangalore, India, January 2018.

[6] José Luis García-Dorado, Felipe Mata, Javier Ramos, Pedro M Santiago del Río, Victor Moreno, and Javier Aracil. High-performance network traffic processing systems using commodity hardware. In *Data traffic monitoring and analysis*, pages 3–27. Springer, 2013.

[7] Pankaj Gupta, Steven Lin, and Nick McKeown. Routing lookups in hardware at memory access speeds. In *INFOCOM'98*, volume 3, pages 1240–1247. IEEE, 1998.

[8] Berry Hoekstra, Damir Musulin, and Jan Just Keijser. Comparing tcp performance of tunneled and non-tunneled traffic using openvpn. *Universiteit Van Amsterdam, System & Network Engineering, Amsterdam*, pages 2010–2011, 2011.

[9] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. In *SIGCOMM '17*, pages 71–84, New York, NY, USA, 2017. ACM.

[10] Dario Lacković and Mladen Tomić. Performance analysis of virtualized vpn endpoints. In *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2017 40th International Convention on*, pages 466–471. IEEE, 2017.

[11] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This poodle bites: exploiting the ssl 3.0 fallback. *Security Advisory*, 2014.

[12] Michele Paolino, Nikolay Nikolaev, Jeremy Fanguede, and Daniel Raho. Snabbswitch user space virtual switch benchmark and performance optimization for nfv. In *Network Function Virtualization and Software Defined Network (NFV-SDN) 2015*, pages 86–92. IEEE, 2015.

[13] Daniel Raumer, Sebastian Gallenmüller, Paul Emmerich, Lukas Märdian, and Georg Carle. Efficient serving of vpn endpoints on cots server hardware. In *Cloud Networking (Cloudnet), 2016 5th IEEE International Conference on*, pages 164–169. IEEE, 2016.

[14] Hugo Sadok, Miguel Elias M Campista, and Luís Henrique MK Costa. A case for spraying packets in software middleboxes. In *Hotnets 2018*, pages 127–133. ACM, 2018.

[15] Haoyu Song, Fang Hao, Murali Kodialam, and TV Lakshman. Ipv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards. In *INFOCOM'09*, pages 2518–2526. IEEE, 2009.

[16] Steve Vinoski. Concurrency and message passing in erlang. *Computing in Science & Engineering*, 14(6):24–34, 2012.