Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

MASTER'S THESIS IN INFORMATICS

Performance Analysis of VPN Gateways

Maximilian Pudelko

Technical University of Munich

Department of Informatics

Master's Thesis in Informatics

Performance Analysis of VPN Gateways

Performanceanalyse von VPN Gateways

Author:	Maximilian Pudelko
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Paul Emmerich, M. Sc.
	Sebastian Gallenmüller, M. Sc.
Date:	December 15, 2018

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching, December 15, 2018

Location, Date

Signature

Abstract

VPNs play an important role in todays Internet architecture. On the commercial side, they connect on-premises networks with cloud instances in site-to-site setups or secure the backbone traffic of mobile cellular broadband towers. But also in private usage VPNs have their place. Privacy conscious users employ them to protect themselves against abusive ISPs or to prevent eavesdropping on public WLAN access points.

Network speeds grew much faster than computing power in the last decade, which opens the question, if todays software VPN solutions are suitable for use in 10 Gbit/s or more networks. We show that state-of-the-art open-source implementations struggle to process traffic at line rate in site-to-site setups. With our evaluation we uncover the performance bottlenecks of them.

By developing our own VPN implementations based on DPDK, we show that further improvements are still possible, but require different approaches to efficiently exploit the modern multi-core CPU architecture. We present a work-sharing pipeline approach, that achieves a reliable rate of 6.3 Mpps on COTS hardware, outperforming all other implementations.

From the gathered insights we create a performance model for VPN implementations, that allows prediction of the systems performance by incorporating the individual cost factors of VPN operations.

Contents

Chapter 1

Introduction & Motivation

Todays world got increasingly more connected over the last decade. With the introduction of smart phones, nearly everyone has a Internet connected device on their hands most the time. New paradigms like Internet of Things (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 the virtual cloud networks. For a seamless integration, all major cloud provide IPsec gateways to create site-to-site VPNs.

Especially these site-to-site use-cases place huge requirements in terms of throughput on a VPN solution. Our initial research in Section 3 suggest that open-source VPN implementations are not fast enough to handle multi-gigabit links at line rate on COTS hardware. Network administrators instead rely on dedicated hardware VPN solutions, which are often expensive and not auditable from a security standpoint.

1.1 Research Question

This thesis evaluates the three open-source software VPN solutions IPsec, OpenVPN and WireGuad available on Linux. With our benchmark suite we determine, if these VPN implementations are fast enough to be deployed on 10 Gbit/s and 40 Gbit/s links with COTS hardware. Further, we measure where their performance bottlenecks lie and why they are not able to handle high loads. These measurements are completed by an analysis of the source code.

Next we develop our own VPN implementations to gain further insights on the typical operation steps of a VPN and to investigate if achieving a higher performance is possible at all. By using our own bare-bones versions, we reduce external influences and can measure the performance hazards more accurately.

From the gathered insights we create a performance model to predict the processing rate of an implementation. The model shows which areas of the VPN operation are the most resource intensive and how different factors influence the total performance.

Chapter 2

Technical Background

This chapter gives technical background information about the function and architectures of VPNs and introduces the necessary concepts from the area of cryptography.

2.1 Virtual Private Networks

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, it should make no difference if it connects to a host over a VPN or directly. What raises VPNs beyond basic tunnel protocols like L2TP, is the added privacy to the tunneled data. Through cryptographic constructs it should become impossible for a third party to read or modify the content of the transmitted messages. We expand on this aspect in Section 2.2.

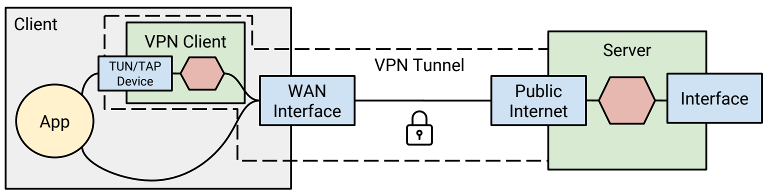
VPNs are used to connect otherwise unreachable networks, either because the host addresses are from un-routable private subnets or some parts of the network are intentionally omitted from public routing for, e.g., security reasons. To transmit these private packets over real physical links, they have to encapsulated inside new packets with different outer headers. Especially for VPNs over the public Internet conformance to the already deployed protocols is mandatory. Which parts of the packet are included depends on the layer of the OSI model the VPN operates on. Generally there are two layers where VPN operation is feasible; the Data Link Layer (L2) and the Network Layer (L3). Which to chose is an architectural decision, that depends on the requirements and existing network structure. Table 2.1 summarizes the major differences.

Layer 2	Layer 3
+ all L3 protocols are supported	- L3 protocols other than IP must be encapsulated
+ multicast frames	- multicast requires special handling
- breaks OSI model	+ integrates in OSI model
	- hosts must be in different subnets because of routing step

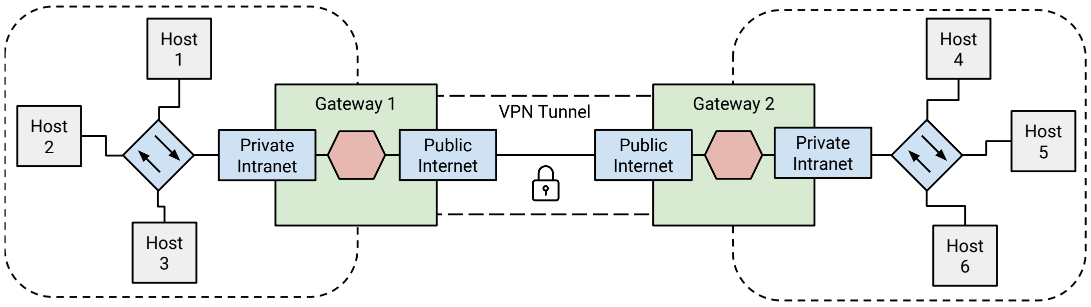
Table 2.1: Differences between Layer 2 and Layer 3 VPNs

Apart from the technological decision, there are also structural differences. VPNs can be employed at different positions in the network, achieving different results with varying requirements. Generally, there are two options where to run the VPN application, both depicted in Figure 2.1. The first position is directly on the client, where the VPN will intercept application traffic before forwarding it to a server, making this a client-server setup. This requires administrative access to each device that should be secured and is therefore often done on employee devices in corporate environments. Employees then can connect to internal databases or websites. The other approach is less intrusive and much broader in scope. Site-to-site VPNs connect entire networks instead of single devices. Gateways are placed at the network edges, so they can receive, encrypt and route traffic that is destined to the respective remote subnet. This solution is employed to interconnect geographically separate locations or to connect on-premise networks with the cloud.

Site-to-site setups are more interesting from a performance standpoint, because they represent choke-points in the network and are subject to much more traffic than a client-server setup. State-of-art laptops, the usual host of client-server VPNs, are not even equipped with physical link NICs anymore and rely on slower wireless links with throughput in 1G range at best. In contrast to server hardware, where 40G and 100G NICs are no rarity anymore. For these high-throughput requirements there also exist specialized hardware VPN appliances with dedicated FPGA or ASIC logic modules. Their high cost and in-auditability make them undesirable to some and creates a niche for open-source software VPN solutions that run on COTS hardware. Therefore, this thesis focuses on high-throughput site-to-site setups, realizable with common VPN software.



(a) Client-Server VPN setup



(b) Site-to-Site VPN setup

Figure 2.1: Different VPN network architectures

2.2 Key Concepts of Information Security & VPNs

While application of cryptography is not strictly necessary for VPN operation, most implementations provide means to secure the tunneled packets. Many use-cases involve the transmission of sensitive information over an insecure channel, thus require one or more of properties described in this section.

Over the years the cryptographic community developed a set of concepts and requirements to measure and compare ciphers, protocols or applications regarding the forms of information security they provide. In the following we list them and explain how they are relevant in the context of VPNs.

Data Integrity

Data integrity describes the property that an unauthorized third-party is not able to modify a message undetected. This includes malicious as well as accidental modifications, such as truncation, bit flips or corruption. Depending on the implementation, is may also allow detection of completely missing or duplicate packets.

In a virtual private network this grantees, that a receiver of a message can either be sure its content as not been modified in transit or can reliably detect such a modification and discard the message.

Data integrity can implemented stand-alone with message authentication codes (MAC) based on keyed hash functions (HMAC) or with asymmetric signatures (RSA-PSS, ECDSA, ed25519). The sender calculates a tag and attaches it to each messages before sending it. The receiver can verify the tag by performing the same calculations over the message's content and checks if the tags still match. A more modern approach incorporates data integrity directly with confidentiality in the form of authenticated encryption (AEAD) ciphers, like AES-GCM or ChaCha20-poly1307.

The techniques up to now covered data integrity on a per message basis, but VPNs commonly do not only transmit a single message, but a stream of information. Depending on the specific upper protocol used, duplicate injection or omission of entire packets may be detectable by the hosts. TCP keeps a window of already seen and still missing segments that triggers retransmission after a timeout. UDP on the other hand does not provide any such guarantees and allows trivial replay attacks. Since VPNs are often used over insecure channels, some implementations integrate these features on the protocol level in the form of replay windows and message counters.

Confidentiality

Confidentiality protects the plaintext of messages from being read by unauthorized third parties. Through encryption the message contents are scrambled and ideally indistinguishable from random data. For VPNs confidentiality allows communication of sensitive information over a public medium such as the Internet or an open WLAN access point.

For performance reasons confidentiality is implemented with symmetric ciphers, instead of asymmetric onces. While each technological field seemingly has its own suite of ciphers (cf. Kazumi in cellular communication), many of them proved insecure after detailed cryptographic analysis. Today only a handful of them are generally recommended [48], AES as the most prominent among them. AES comes with hardware support on many platforms making it a safe and fast choice. Newer ciphers such as ChaCha20 aim for simplicity in order to reduce the chance of implementation errors and side-channel attacks, while remaining reasonable fast, even on platforms without dedicated hardware instructions.

Availability

Availability describes the ability of a system to be resistant against service disruption, even under adverse conditions.

A holistic approach to availability requires more consideration than just the VPN application, but there are none the less some aspects where a thoughtful protocol can help. A VPN should be resistant to simple resource exhaustion attacks, where an attacker sends a stream of packets to the victim, triggering, e.g., expensive calculations or memory allocations. One vector would be the initial handshake message, which triggers the victim to do expensive asymmetric key derivation calculations, in form of curve point multiplications. This can lead to CPU exhaustion, as the server is completely occupied with answering faked initiations and is not able to handle real traffic anymore. A possible countermeasure against this are MAC cookies as part of the handshake when under load. These prevent attackers from using spoofed IP addresses successfully and then allows application of classic rate limiting schemes.

Non-Repudiation

Non-repudiation in a legal sense will not be reachable with a VPN alone. In VPNs non-reputation means, that only registered and known entities are participating in the network and messages can be attributed to a single host.

This can be implemented in various ways. In large-scale enterprise environments certificates binding public keys to identities together with a public-key infrastructure (PKI) are common. A more ad-hoc way of doing this is either keeping a list with the public key fingerprints of all allowed users on each host, known from SSH or using static passwords. Both types of manual configuration are done out-of-band.

Chapter 3

Related Work

The following section gives an overview about related work in the field of VPN performance measurements, Linux network stack analysis and flow classification.

In their extensive report from 2011 Hoekstra and Musulin [27] evaluated the performance of OpenVPN in gigabit networks, with a focus on bandwidth specific bottlenecks. In their lab setup the authors tested on two network topologies, of which the client-to-client setup nearly matches our site-to-site one. Unfortunately they did not state which version of OpenVPN they benchmarked. Also, the researchers mainly focused on throughput measurements in terms of Mbit/s instead of million packets per second (Mpps). This stems from the usage of iperf in TCP mode for packet generation, where the size of send packets can only influenced indirectly by the MTU and MSS. According to their setup scripts, they did not set a MSS, in which case iperf defaults to the MTU of the link and subtracts 40 bytes for the TCP/IP headers [33]. By configuring a link MTU of 1500 bytes, as documented by them, this results in 1460 byte packets during their tests. They identified the internal fragmentation mechanism to be a bottleneck once the MTU of the TUN device configured larger than 9000 bytes. By disabling it, they could improve throughput by around 30% to 80% for AES ciphers and 150% for the Blowfish-128-CBC cipher. For secure configurations (AES-128-CBC with HMAC-SHA1) they measured a maximum throughout of around 160 Mbit/s to 270 Mbit/s. By switching to the 256 bit blocksize version of the AES cipher, it decreased to 140 Mbit/s to 210 Mbit/s. An observation we can confirm in our benchmarks. Performance on 10G networks was not evaluated because no hardware was available to them. We build upon their research by testing the same configuration optimizations in our tests and can confirm, that these still improve performance. As a major difference to their setup we use UDP traffic, which allows exact control over the packet size. By measuring throughput in both Mpps and

Mbit/s, we show that some bottlenecks occur on a per-packet basis, while others emerge only at high bandwidths.

In another paper from 2011 Kotuliak et al. [31] compared the throughput and latency of OpenVPN to IPsec with the help of a program called IxChariot. Their setup consists of two virtual machines running on top of two Windows Vista PCs. While they acknowledge the fact that VMs introduce an additional overhead through virtualized drivers, they do not measure or otherwise quantify it. There is also no information on the type, shape or form of traffic used as a load generator, which makes comparisons with other results nearly impossible. For OpenVPN configured with an AES cipher, they report a throughput of 99 Mbit/s at 62% CPU load. IPsec, when used with AES, performs better at 142 Mbit/s, but also increases the CPU load to 90%.

A more recent comparison from Lackovic et al. [34] also 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. [47] 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). Intel [26] even measured a quadrupled rate for the AES-GCM cipher. Additionally, Lackovic evaluates the scaling opportunities of the VPNs and concludes that IPsec is more scalable than OpenVPN. They too notice the limitations of the single-threaded OpenVPN application and work around it, by setting up four instances in parallel. This increases the processing rate from 400 Mbit/s to 1050 Mbit/s for packets larger 1400 bytes. We apply the same technique in Section 5.5.4.

Sadok et al. [49] investigated the efficiency of 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 99^{th} percentile in time windows of 150 μs. With RSS this leads to a uneven resource utilization, as only a few cores are handling all traffic. To check how VPNs handle this, we also include such traffic in our benchmarks. Spayer, their software solution, relies on the ability of the NIC to deliver flows to specific queues, as implemented in the Flow Director feature of Intel NICs. Using the pseudo-random TCP checksum field, they deliver packets uniformly to all cores and only route the ones modifying the flow state to the associated core. Spayer is limited to TCP traffic due to concerns about packet reordering in UDP flows.

At the time of writing, there is very little academic research published about WireGuard. In its initial publication [8], 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, much less on 10G networks.

Chapter 4

MoonWire

This Chapter describes the design and implementation of MoonWire, our VPN sandbox. It is evaluated among the other VPNs in Chapter 5.

4.1 Design

We wanted 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.

Protocol

For the VPN protocol and data framing the WireGuard protocol is chosen due to its minimal design. Only 3 types of message frames are needed to create a new channel and exchange data.

MoonWire aims the be compatible with WireGuard, but does not reach this goal in some aspects. The cookie response mechanism for DoS protection and re-keying timers are not implemented. WireGuard's replay window of 2000 slots is sometimes to small to capture the reordering effects of having independent worker threads. On the measurements this has no impact, since we record the packet rates on the direct links from and to the DuT, i.e. only the encrypting VPN gateway, which does not include the decryption step.

Software

Previous research [23, 3] and our own measurements in Section 5.3 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 [22]. 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 [17, 38] and allows direct access to C ABI functions, i.e., the DPDK library.

For cryptographic primitives the open-source libsodium library [5] is used. It is a cross-platform collection of high-security, but none the less fast implementations, provided behind an easy-to-use, hard-to-get-wrong API and has several benefits over rolling your own implementations.

The exact versions of all used software is given in Table 5.6.

The complete source code of all variants is available in the `lua` directory of the master repository [45]. Wrappers around un-exported static inline functions of DPDK are found in the `src` directory.

4.2 Implementation

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.

4.2.1 LuaJIT Optimization Flags

While the LuaJIT compiler employs all optimizations by default, it has to choose conservative baseline values that provide good results even on resource restricted platforms. On high-end x86-64 hardware, more aggressive values can configured to yield better performance. Table 4.1 shows the parameters where we deviate from the defaults. The

Parameter	Value	Description
maxrecord	20000	Max. number of recorded IR instructions
maxirconst	20000	Max. number of IR constants of a trace
loopunroll	4000	Max. unroll factor for loop ops in side traces

Table 4.1: Used LuaJIT optimization flags

parameters are applied at runtime as part of the source code at the top of each main Lua script.

The loop-unrolling parameter proved to be the most relevant one. LuaJIT stores compiled and optimized chunks of code in form of traces. For nested loops, that generally occur in code operating on batches, the hotspot selector will only unroll and inline the inner loop if its iteration count is low. Otherwise an additional trace for the inner loop will be created. This inner trace then is entered and exited on each loop iteration, creating runtime overhead. By increasing the loopunroll factor, the heuristics about the loop iteration count are relaxed, resulting in the unrolling and inline of all inner loops, thus only a single longer trace. Here the other parameters come into play, by allowing a trace to be much longer than usual, containing more instructions and resulting in even less overhead from trace switching.

In some cases this flag combination can improve performance tenfold over the default values. All parameters have been found through previous experiments and where increased until local maximums were reached.

4.2.2 Details / Data Structures

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

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 interesting packets on the destination IP address, by checking them 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 [25, 4, 51] with many established high-performance implementations. For MoonWire we rely on the LPM library of DPDK, which uses the DIR-24-8 algorithm for its IPv4 table. For the main use-case of this thesis, site-to-site

VPNs, this table and its performance is not as important, since it will only contains a handful of long-term entries to the other gateways. This is also confirmed by our numerous benchmarks, where table lookups are virtually non-existent, cost-wise.

Peer State

The peer state encapsulates all information necessary to send and receive packets from a remote VPN peer. 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 [9]. In Listing 4.1 the minimal version of the peer state in MoonWire is shown.

```c
struct peer_no_lock {
    uint8_t rxKey[crypto_aead_chacha20poly1305_IETF_KEYBYTES];
    uint8_t txKey[crypto_aead_chacha20poly1305_IETF_KEYBYTES];
    uint8_t nonce[crypto_aead_chacha20poly1305_IETF_NPUBBYTES];
    uint32_t id;
    union ip4_address endpoint;
    uint16_t endpoint_port;
    /* rte_spinlock_t lock_; */
};
```

Listing 4.1: Definition of the MoonWire peer struct without locks

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.

A Note on Nonce Handling

Nonce counters used as IVs play an important role in protecting the confidentiality in symmetric ciphers. In the case of AES-GCM forgery attacks become possible if a nonce is reused under a given key [14]. The same applies to Bernstein's ChaCha20 cipher with Poly1305 as authenticator [37]. NIST mandates, that the probability of this occurring must be smaller than 2^{-32} in conforming implementations [14].

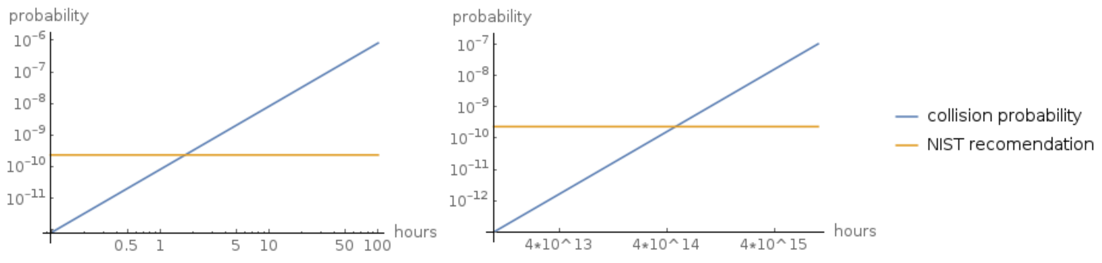
For uniformly, randomly generated nonces, this probability is a generalized form of the birthday problem and can be calculated by Equation 4.1, where the nonces are the days and the packets replace people.

$$p(\text{packets}, \text{bits}) = 1 - \frac{\text{packets}! \times \binom{2^{\text{bits}}}{\text{packets}}}{(2^{\text{bits}})^{\text{packets}}} \tag{4.1}$$

Calculating binomial coefficients of large numbers is computationally expensive. Therefore, the above formula can be approximated by the Taylor series expansion of the exponential function [50], as long as $packets \ll 2^{\text{bits}}$:

$$p(\text{packets}, \text{bits}) = 1 - e^{-\frac{\text{packets} \times (\text{packets}-1)}{2^{\text{bits}} * 2}} \tag{4.2}$$

Figures 4.1 (a) and (b) plot Equation 4.2 for a constant packet rate of 1 Mpps and nonces of 96 and 192 bit size. The horizontal line shows the NIST recommended maximum risk of 2^{-32}. On the x-axes the number of transmitted packets in hours are given. Note that both axes are logarithmic.



(a) 96 bit nonce, AES-GCM & ChaCha20 (b) 192 bit nonce, XChaCha20

Figure 4.1: Collision probabilities of randomly generated nonces at 1 Mpps

For 96 bit nonces, like used in AES-GCM (IPsec) and IETF-ChaCha20 (WireGuard), the threshold is passed in under 2 hours of traffic. This concludes why random generation of small nonces is not recommended. Instead, it is usual to keep the last number and increment it with each packet. In multi-threaded implementations this can become a bottleneck as the nonces are essentially shared state between multiple read-write threads and have to be synchronized.

A possible solution to this problem could be the usage of ciphers with larger nonces, such as XChaCha20, which uses 192 bit nonces. The enlarged value range decreases the probability of collisions over-proportionally, as seen in Figure (b). There it takes over 4×10^{14} hours to reach the same risk level as above. An equivalent of 10 times the age of the sun, for just 12 byte larger nonces. This would also scale to hypothetical future traffic flows in terabit range. Table 4.2 lists the times until the NIST recommended collision risk level is reached for various packet rates. For 1 Tbit/s traffic of 64 byte packets, XChaCha20 would still be enough for 3.19×10^{11} hours.

Packet rate	96-bit nonce	192-bit nonce
1×10^3	1687 hours	4.75×10^{17} hours
1×10^6	1.68 hours	4.75×10^{14} hours
14.88×10^6	6.6 minutes	3.19×10^{13} hours
1.488×10^9	4 seconds	3.19×10^{11} hours

TABLE 4.2: Time until NIST recommended collision risk level is reached at different packet rates

4.2.3 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.

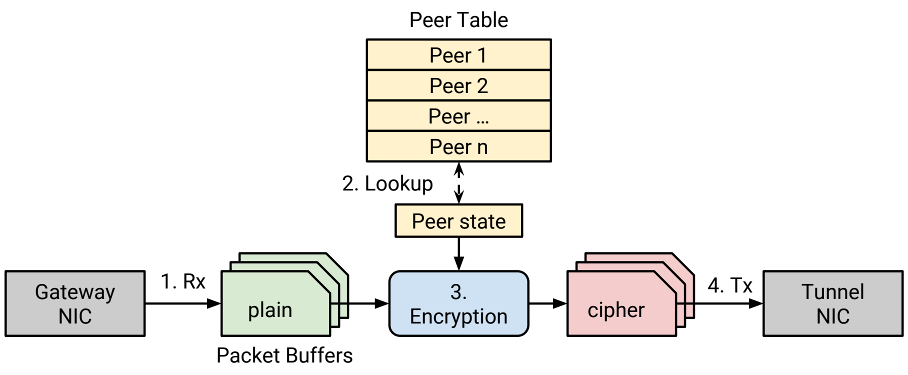


FIGURE 4.2: Flowchart of MoonWire Variant 1

In Figure 4.2 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 of `rte_pktbufs`. Each buffer is validated and the associated peer state is looked up in the peer table. If there is a running 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, thus reclaiming their descriptors and memory into the pool of the gateway.

Due to its single-threaded nature, there is no need for any kind of synchronization on the peer table or state. This simplifies the state-keeping process greatly and poses the least requirements on the used data structures. A simple LPM table such as the one provided by DPDK [42] can be used to implement the peer table. This variant does also no 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.

4.2.4 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.

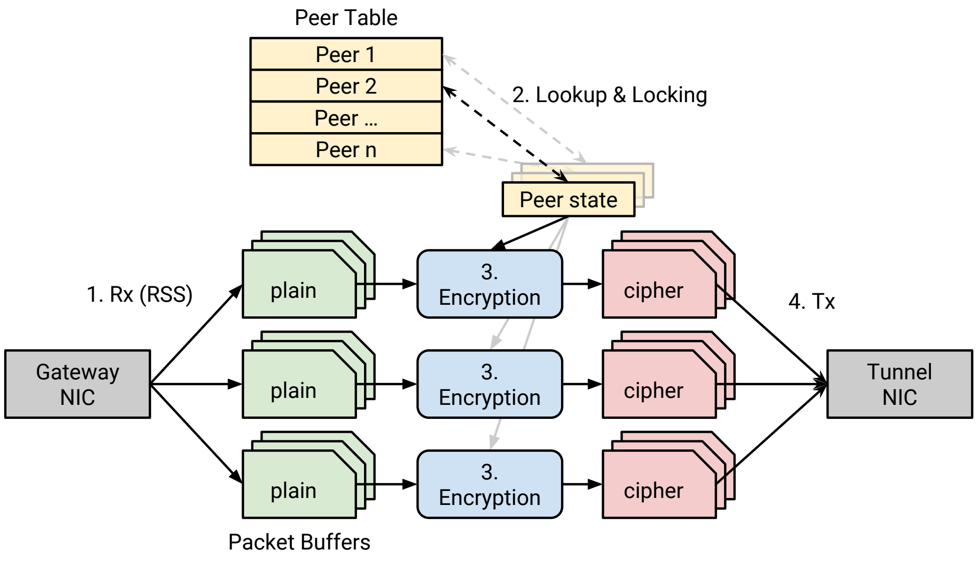


Figure 4.3: Flowchart of MoonWire Variant 2

As seen in Figure 4.3, 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. The DPDK LPM lookup table itself is not thread-safe with multiple writers for performance reasons, but read-only lookups can be performed concurrently [41]. Fortunately, changes to the table structure are rare, namely when the VPN configuration is modified, so this operation is allowed to be more expensive and slow. The peer state, on the other hand, is written to for every packet when the nonce is incremented. In our implementation it is secured by a traditional lock. Special care has to be taken when to acquire and hold it. In Listing 4.2 the correct strategy as well as a suboptimal placement, is shown. The incorrect form results from an API limitation, which automatically increments the nonce as part of the encryption call to minimize programmer errors. While this is less prone to bugs, it encourages lock function placement right before and after the encrypt call. This effectively allows only one encryption operation to take place at the same on the entire system, thus very slow performance. The correct approach recognizes that only the nonce write (and to some degree the key material read) is critical, while the actual encryption is not. It therefore acquires the lock, modifies the nonce, makes a copy of the whole state and releases the lock again. Only then the expensive cryptographic operations take place.

```
peer:lock()
sodium.sodium_increment(peer.nonce, NONCE_BYTES)
ffi.copy(tmp_peer, peer[0], ffi.sizeof(tmp_peer))
peer:unlock()
peer:lock() -- incorrect placement
msg.encrypt(buf, tmp_peer.txKey, tmp_peer.nonce, tmp_peer.id)
peer:unlock() -- incorrect placement
```

Listing 4.2: Locking the peer struct

There exist many implementations of locks. We take the classic `pthread_mutex` from the POSIX threading library and compare it to the `rte_spinlock` provided by DPDK, henceforth called Variant 2a and 2b respectively. The POSIX mutex internally relies on `futex` API of the kernel and is a general purpose lock. In particular, when waiting for a lock to be become free, the calling thread is suspended and does not consume further cycles. Spinlocks, 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 contenting for a lock, as seen by the spinlock line pattern in Figure 5.27.

4.2.5 Variant 3 - Work Pipeline

In Variant 3 the best properties of the previous versions are taken, while avoiding their drawbacks. Variant 1 showed, that not utilizing multiple cores will limit performance greatly, as operations like encryption, especially of large buffers, inherently require a lot of processing power. A shared peer state with locks, as implemented in Variant 2, introduces too much per-packet overhead through the synchronizations mechanisms. Scaling beyond a single-digit number of threads is not possible in our measurements.

Therefore, a hybrid approach in chosen for Variant 3. The peer table and state is still contained in and maintained by a single thread, therefore lock-free, while the encryption

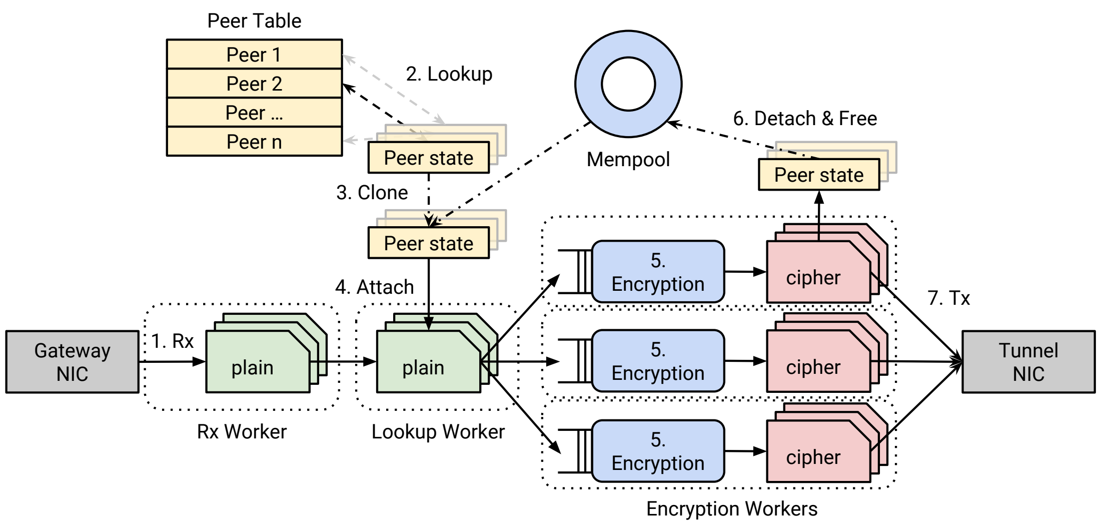


Figure 4.4: Flowchart of MoonWire Variant 3

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 [56] or the Message Passing Interface (MPI) library [44].

Figure 4.4 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 increments the nonce counter, clones the entire state with all keys and endpoint information into a separate empty buffer, and attaches this buffer to the packet buffer. The `rte_membuf` struct includes a member field called `udata64` for this purpose. The parallel workers now find all information needed for encryption contained in the packet buffer and do not have to updated any shared state. Since this crypto-args buffer is separate allocated memory, it has to be reclaimed after use. Traditional memory management with `malloc()/free()` is too slow to be used in the data path a million times per second concurrently [19, 40]. Therefore, it is implemented the same way as the packet buffers; as fixed-sized, memory-pooled data buffers. DPDK also exposes the specialized packet buffer memory pools as plain `rte_mempools` without the network packet specific customization, so we use them here. These plain memory pools are untyped and are not aware of the stored data, apart from the size of each element. Only the semantics of the user code give the returned `void*` pointers their meaning. Listing 4.3 shows the definition of the message struct, which we construct inside the buffers. Note, that the peer struct is fully embedded and not merely referenced. We also include a pointer back to the owning memory pool. This information is needed for the free operation, to put a buffer back in the correct pool. A pedantic reader could argue, that this information is redundant, as we only have one memory pool and could pass this pointer to the workers out-of-band once. We argue, that this is a worthy trade-off. Firstly, this pointer is only set once at pool initialization and then is never changed, since buffers do not migrate between pools. Secondly, this allows extension to multiple pools in the future, i.e., one per NUMA domain.

```c
struct work {
    struct peer_no_lock peer;
    struct rte_mempool* pool;
};
```

Listing 4.3: Definition of the message structure attached to packet buffers

Packet distribution between the stages happen through DPDK `rte_rings`; lock-free, FIFO, high performance queues with batched functions to reduce overhead. The step from the lookup to the encryption workers could be solved by having one MPMC queue, but this introduces more shared state than multiple SPSC queues. For one, SPSC semantics allow DPDK to use more relaxed and faster enqueue/dequeue functions without an atomic CMP-XCHG loop [43]. It also reduces false sharing cache evictions caused by dequeues from other workers. Overall this trades memory usage for performance. Distribution over the workers happens in a round-robin fashion to ensure equal load among them.

In a previous version the Rx and the Lookup worker where unified into one. But as shown in Figure 5.32, there is a significant amount of time spend for state cloning. To speed this process up, packet reception has been factored out in the final design to give this task as much processing power as possible.

Back-Pressure in Pipelines

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 what can be encrypted. Noticing and propagating this information backwards is necessary to govern the input stages, prevent wasting of resources and reduce overload [32]. In our design this is done implicitly over the queues connecting the stages. If a queue is full, no further entries can be added and the enqueue call fails. The caller then only has to notice this and clock itself down a little bit. Listing 4.4 shows the regulation mechanism in the Rx worker. If passing the packets to the next stage fails, a counter in increased and this workers suspends itself for fails^2 microseconds. This repeats itself with exponentially increasing back-off timers until the following stage accepts packets again. The 802.11 standard for wireless cards specifies a similar behavior [28] to deal with collisions.

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.

```lua
local fails = 0
while lm.running() do
    local rx = gwDevQueue:tryRecv(bufs, 1000 * 1000) -- 1 ms
    if rx > 0 then
        local suc = outputRing:sendN(bufs, rx)
        if not suc then
            bufs:free(rx)
            fails = fails + 1
            lm.sleepMicros(fails * fails) -- exponential backoff
        else
            fails = 0
        end
    end
end
```

Listing 4.4: Self-regulating Rx worker

Chapter 5

Evaluation

This chapter evaluates the presented VPN implementations. The sections about the individual implementations are structured similarly. First we give background information about the test setup and protocols . Then we list how the applications are setup for the benchmarks and what configurations we tested. If changes to the test setup were necessary, these are also described there. Lastly follows the presentations of the results and a detailed analysis of the bottlenecks we could identify.

All setup and benchmark scripts can be found in the `benchmarks` directory of the main repository [45].

5.1 Testbed Hard- and Software

Testbed

All benchmarks are performed inside a separate testbed on dedicated hardware. We use a standard Ubuntu 16.04 live image with Linux kernel version 4.4.0-109. This version does not yet include the patches to Spectre, Meltdown et al., so the measured rates might be lower compared to future versions. The Intel Turbo Boost feature of the CPUs has been disabled, so that the clock rates remain within their nominal range and the results are more consistent. In Table 5.1 the exact specifications are listed.

Figure 5.1 shows the wiring between the load generating host and the DuT.

	Load generator	DuT
CPU	Intel Xeon E5-2620 v3	Intel Xeon CPU E5-2630 v4
RAM	32 GiB	128 GiB
NICs	4× Intel XL710	4× Intel XL710
OS	Ubuntu 16.04 Linux 4.4.0-109	Ubuntu 16.04 Linux 4.4.0-109

Table 5.1: Testbed hardware

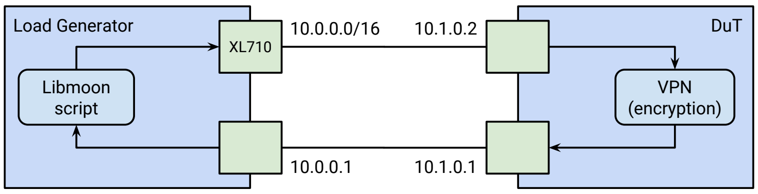


Figure 5.1: Testbed wiring

Load Generator

To generate the required traffic we wrote a Lua script based on the libmoon toolkit [15]. Compared to tools like iperf, this gives us much more control about timings, the exact data send, while also being much more performant [17]. By bypassing the kernel network stack, we can generate any kind or pattern of traffic, without being restricted by security measures or configuration limitations. E.g. it is possible to generate traffic originating from an entire subnet with a single NIC, without having to set any IP address on the interface. Also it allows the usage of precise rate limits, a topic traditional user-space tools struggle with, as show by [16].

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 load generator both the outgoing and the incoming packet rates are recorded by the packet counters in the NICs.

5.2 Traffic Characteristics

Figure 5.2 shows the architecture, which we want to emulate. It is not necessary to replicate it in detail, as the hosts can easily emulated by generating traffic from and to multiple IP addresses on the load generator. We are interested in the gateways performance, therefore gateway 1 is our DuT.

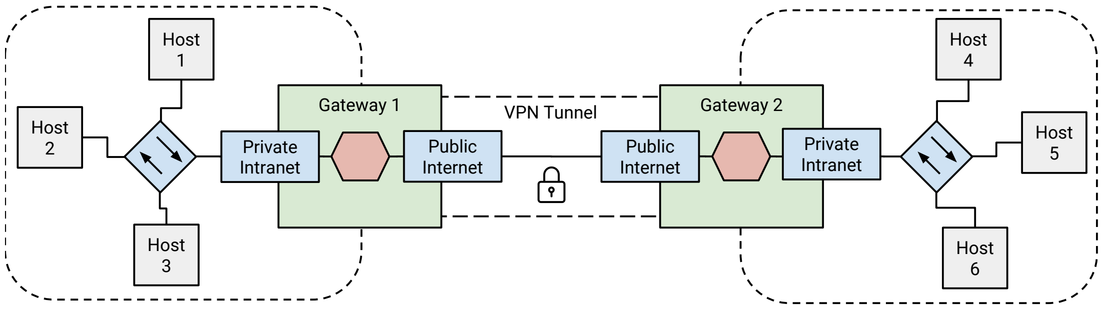


Figure 5.2: Overview of a Site-to-Site VPN Setup

Number of Flows

A flow is usually classified by a 5-Tuple of L2 source and destination address, L3 protocol and L3 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. We found that not all NICs can perform RSS on L3 field information or do so by default. Therefore we only use the L2 addresses to differentiate flows.

Protocols

UDP on top of IPv4 is the protocol of choice since it is stateless and does not depend on feedback or acknowledgments from the other side. It also does not exhibit many side effects TCP has. Correctly benchmarking TCP connections can be difficult and error prone, as TCP has a complicated internal state machine with many configurable variables. It is out of the scope of this thesis to cover the complex interplay of the link MTU, TCP window & maximum segment sizes (MSS) and scaling algorithms like reno or cubic.

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.

Programs seldom scale linearly with the offered load, but exhibit different pattern over the load range. At lower loads a device can handle the all packets, which results in the output rate being equal to the input rate and a resource utilization lower than 100%.

At a certain point this changes as the device enters the overload phase and has to drop packets. This does not mean that all resources of the device are fully utilized, but merely that a particular piece has become the bottleneck.

Rate Control Methods

Packet rate alone does not sufficiently define the traffic pattern. A given target load can be generated by different load-generator methods that, while all delivering the same amount of packets, look different. Emmerich et al. [18] showed that micro-bursts can have a measurable impact on the DuT. In their tests a Linux Open vSwitch router exhibited a lower interrupt rate for burst-y traffic than for constant-bitrate (CBR) traffic.

To verify that the type for traffic does not overly impact performance of the VPNs, we subjected the DuT to the different rate-limiting methods available in libmoon. With preliminary tests found that for our high throughput application, the method has no impact on processing rate. Unless one is doing latency measurements with nanosecond resolution, this not of concern.

Since the hardware based rate-liming mechanism are more flexible in usage regarding multiple threads, all other benchmarks are performed with them and in open-loop.

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 the payload if it. The variable costs on the other hand are heavily influenced by the size and scale accordingly. Encryption or decryption only happens to the degree of the bytes there to handle.

5.3 Baseline: Linux Network Stack & Router

With the exception of MoonWire, all three tested VPNs rely on the kernel network stack for low-level functionality. This includes the handing of drivers, operating the hardware, parsing protocols such as TCP, UDP, IP, queuing and routing of packets.

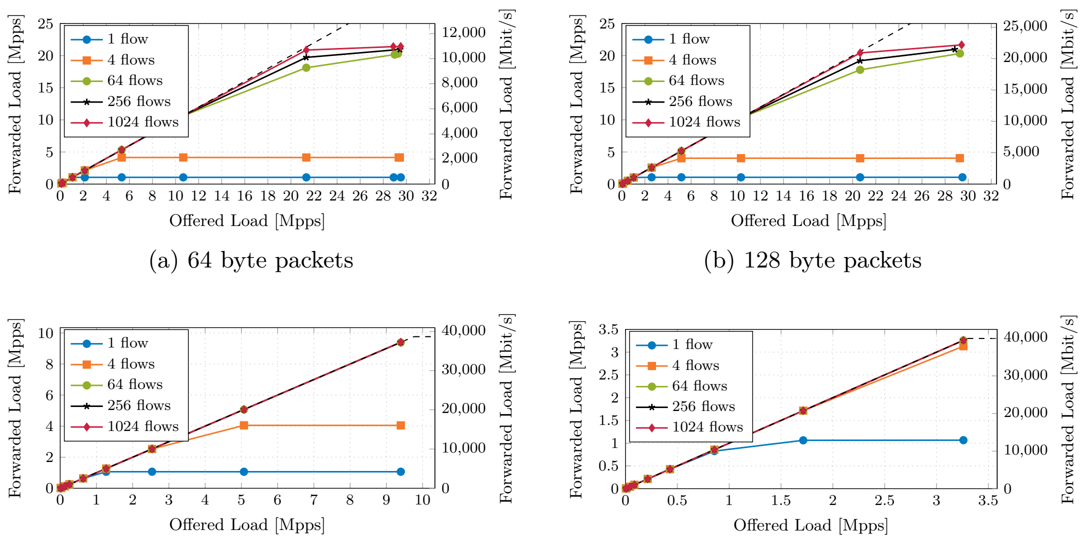
In benchmarks the costs of passing a packet through the network stack is sometimes not separable from the costs of the application. To classify the performance of the VPNs more accurately, we need some form of baseline performance to determine if a bottleneck is caused by the application or roots in the Linux network stack. Therefore we also measure the performance of it directly.

While others already published performance measurements for Linux, we want 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.

5.3.1 Forwarding Performance

Routing is a fundamental operation in a L3 VPN and often offloaded to the kernel. Before a packet is given to an application by, e.g, virtual interfaces, the kernel first validates its fields and decides if it is should be further processed. After processing by the application the encrypted packet is given pack to the Linux kernel, where its destination has to be looked up before it is sent out.

For the following benchmark we configured one manual route on the DuT, so that it forwards the packets back to the load generator. Including the default gateway to the management host and link scope routes, the main routing table contained six routes.



(a) 64 byte packets

(b) 128 byte packets

(c) 512 byte packets

(d) 1514 byte packets

Figure 5.3: Linux router forwarding rates with different packet sizes and varying number of flows

Figure 5.3 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-axes display the applied load to the DuT, while the left and right y-axes measure the traffic, that was received back on the load generator.

It can be seen that the kind of traffic has an measurable impact on the routing level already: traffic consisting of just one or a little 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 1.006 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 just sends millions of packets to lower performance for everyone else disproportionally.

The influence of the packet size is twofold. On the one hand it does not decrease the rate of which packets are processed. Traffic of four flows always peaks at 4 Mpps. On the other hand it naturally increases the throughput in terms of bytes per second. A link bandwidth of 40 Gbit/s can be reached with 512 byte packets and 64 flows.

This is somewhat expected for a router, as it only looks at the first few bytes in the Ethernet and IP header of a packet to make its routing decision. Figure 5.4 fortifies this observation.

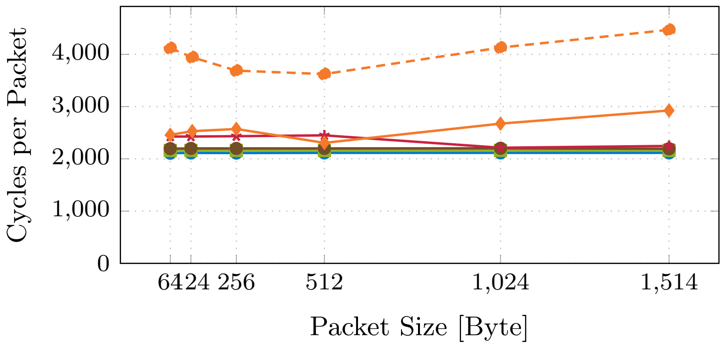
Overall it can be said that the Linux network stack is able to handle 10G links, if either the packets are not very small or a sufficient number of flows run in parallel on large CPUs.

5.3.2 Packet Distribution

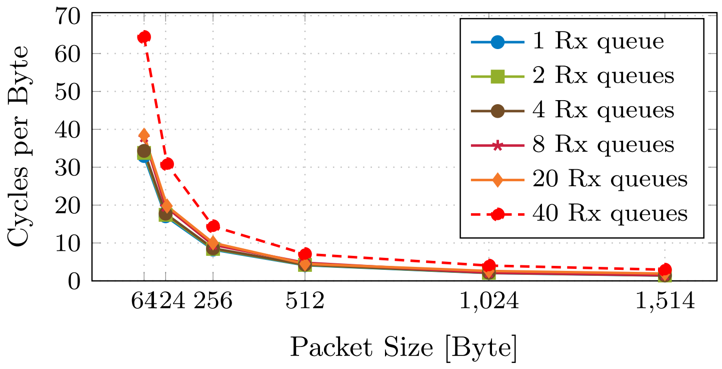
The first step to investigate the slow forwarding performance with traffic consisting of few flows is to look at the CPU utilization of the DuT. This can be done with the `perf` toolset which includes a sampling profiler. By comparing the values read from the CPU internal performance & cycle counters to the clock rate of the CPU, we can calculate the relative utilization of each core:

$$\frac{\text{recorded cycles}}{\text{runtime}} \times \frac{1}{\text{CPU clock rate}} = \text{CPU utilization} \tag{5.1}$$

Figures 5.5 (a) to (d) show heatmaps of the CPU utilization per core for the different traffic pattern. The snapshots are taken at peak forwarding rate for the respective kind of traffic.



(a) Cycles per packet

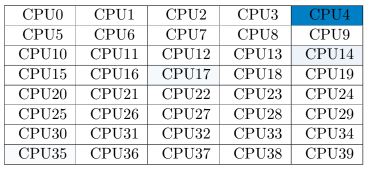


(b) Cycles per byte

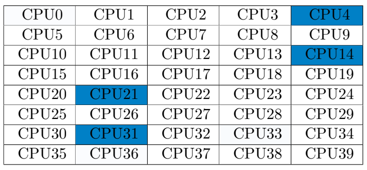
Figure 5.4: Cycle spend for Linux routing & forwarding

For the single-flow traffic it can be seen that only one core, in this case CPU 4, is actually processing incoming packets, while the others are effectively idle. This explains the much lower forwarding rate compared to the other traffic, one core can simply not keep up with the amount of packets. Compared to traffic with, e.g. 256 flows, the trend line becomes clear. More parallel flows result in a more distributed load, as seen in Figure (d), where all cores are utilized to some degree and a much higher forwarding rate can be achieved.

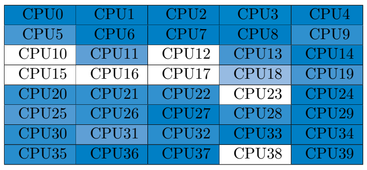
NICs uses a hash function over the packet headers to distribute incoming packets to the Rx-queues in a technique called receive-side scaling (RSS). If each header contains the same values, as it is the case for single-flow traffic, all packets end up in the same queue, negating any possible scaling effects. This is further worsened by the default configuration of how the queues are serviced by the CPUs. On symmetric multi-processor (SMP) systems the kernel configures as many queues as there are CPU cores and binds them 1:1. Each core then only receives interrupts for exactly one Rx-queue. The bind-



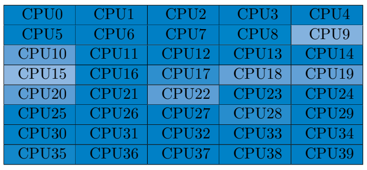
(a) 1 flow



(b) 4 flows



(c) 64 flows



(d) 256 flows

Figure 5.5: CPU utilization of the Linux router during peak load

ing, called IRQ-affinity, can be checked and configured via the `sysfs` pseudo filesystem under `/proc/irq` and `/proc/interrupts`.

Alternative IRQ/Queue Configurations

Apart from the default 1:1 mapping, there exist other possible configurations. Generally, there is advise against delivering one interrupt to multiple CPUs as it can decrease performance due to caching effects. However, in the case of single-flow traffic a round-robin distribution over multiple cores could still have a net benefit. Listing 5.1 shows how to configure a single rx-queue that delivers interrupts to the CPUs 0 to 7. On our testbed hardware this did not work, as the NIC still delivered interrupts to the first CPU of the affinity list only.

```
ethtool -L eth0 combined 1;
echo 0-7 > /proc/irq/124/smp_affinity_list
```

Listing 5.1: Commands to configure a single RXTX queue and bind it to multiple cores

Another approach is to reduce the number of queues, with the goal to improve efficiency. In Section 5.3.3 we try this approach and show, that it can reduce lock contention.

5.3.3 Source Code Analysis: Spinlocks

We also record which tasks the CPUs perform to see where the cycles are spend. Table 5.2 displays the detailed workload grouped into major categories. In cases of uneven load distribution we pick a core with a high load.

Percentage	Symbol
19.39%	[k] _raw_spin_lock
10.07%	[k] fib_table_lookup
5.73%	[k] i40e_lan_xmit_frame
5.16%	[k] i40e_napi_poll
4.02%	[k] ip_finish_output2
2.75%	[k] ip_route_input_noref
2.66%	[k] ___slab_free
2.43%	[k] ___build_skb
2.26%	[k] ___netif_receive_skb_core
2.19%	[k] kmem_cache_alloc

Table 5.2: Top 10 most cycle consuming functions in Linux router with 4 queues

Spinlock in Tx Path

We found multiple spinlocks in the send path that have to be taken. Inside the queue discipline (qdisc) code paths, there is a per queue lock, that has to be taken for every skb sent [55]. This presents an obvious bottleneck, so there also have been previous efforts to reduce lock contention [13].

We have found though profiling, that a large portion of the time is spend in synchronization functions. Synchronization often behaves differently depending on the number of participants. E.g., the performance of a mutex scales inversely with number of threads competing for it To get a better understanding of this area, we measure the cycles spend synchronizing while controlling the number of cores active. One could craft sophisticated traffic with flows that would exactly utilize the wanted cores, but this is unreliable and rather complicated. It is easier to configure the number of RX queues of the NIC directly. On Linux this can be done with ethtool:

```
ethtool -L eth0 combined 8
```

Listing 5.2: Command to configure 8 RXTX queues on a NIC

Further, traffic with many flows should be used so that all queues get equally utilized and uneven load does not occur.

In Figure 5.6 it can be seen that the synchronization share indeed increases with the number of queues, and therefore cores, active. This also explains the less-than-linear scaling effect when adding more queues. Time spend in spinlocks is not used to process packets, thus leading to a lower per-core forwarding rate with more cores.

Overall the performance of the Linux kernel is comparably good. A 10 Gbit link filled with minimally sized packets can be fully handled, as long as it does contain around

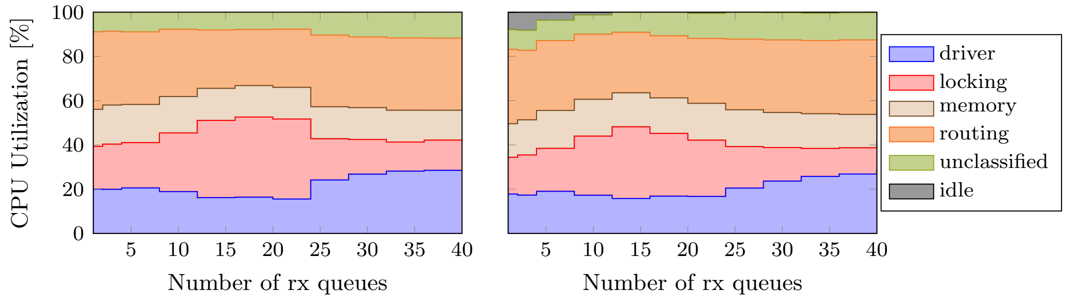


Figure 5.6: CPU cycles spend in Linux router with 64 byte and 1514 byte packets

64 concurrent flows. One should keep in mind, that these benchmarks are not fully illustrative of the capabilities of Linux as a router, but only serve as a baseline for our other measurements. A real world router has large routing tables and many NICs, while our setup only consists of two subnets and two NICs.

5.4 IPsec

IPsec is an IETF standard for network layer security in IP networks. It is engineered to academic perfection, follows a layered approach and solves key exchange by a completely separate protocol. Due to its inherent complexity, it is non-trivial to setup and run correctly.

As part of this Thesis we evaluate the IPsec implementation of the Linux kernel. Only the encryption is studied, as the key exchange mechanism is not relevant in throughput evaluations. They can have an impact on the latency of the initial packet, but do later only run as part of re-keying operations. Therefore, we refrain from using a complete IPsec software suite like strongSwan, and configured the devices directly with static keys using the iproute2 tools. We chose ESP tunnel mode with the RFC4106 AES-GCM symmetric cipher, which is an AEAD cipher. An security policy (SP) has to be configured on the DuT. Listing 5.3 shows the command.

```
ip xfrm policy add \
    dir out \
    src 10.0.0.0/16 dst 10.2.0.0/16 \
    tmpl src 10.1.0.1 dst 10.1.0.2 \
    proto esp mode tunnel
```

Listing 5.3: iproute2 command to create a SP on the gateway host

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 5.7 show the measured forwarding rates for IPsec. For single-flow traffic we see the same drawback a in the baseline measurements. All packets get delivered to a single core, overloading it. For more flows the rate increases up to a hard limit of 2 Mpps. In the Section 5.7.1 we analyze the source code for possible bottlenecks.

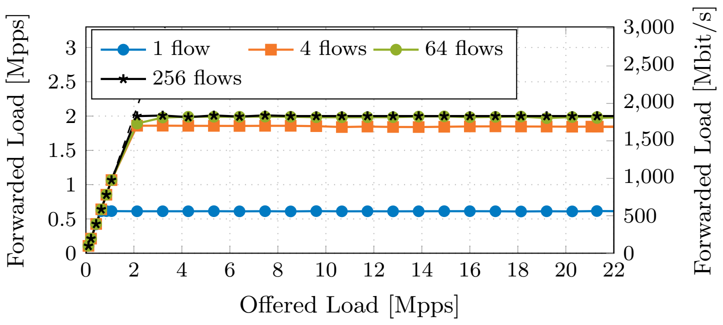


Figure 5.7: Processing rates of IPsec with 64 byte packets

5.4.1 Scaling through Multiple Security Associations

With profiling we found, that a lot of time is spend synchronizing access to data structures. One such data structure is the IPsec security association (SA) which defines how a packet should be handled. In the Linux kernel a SA is represented by the `struct xfrm_state` [52]. Among other configuration it contains the replay protection information and a spinlock to synchronize access.

To enable the detection of duplicate packets on the receiving side, each processed packet gets an incrementing sequence number, which is stored in the IPsec AH header. Since multiple CPUs can process packets governed by the same SA in parallel, each CPU first has to acquire the spinlock to the `xfrm_state` before it can modify it. This happens in the `xfrm_output_one()` function [53], which in turn calls the `xfrm_replay_overflow()` function [54] to read and increment the current output sequence number (`replay.oseq`). In summary this means, that with only a single SA a de-facto global spinlock has to be acquired and released for each packet, contributing to the slow forwarding rates compared to the baseline measurement.

To distribute this contention over multiple locks, additional SAs can be set up. Each SA then only handles a small subset of the traffic by restricting it to, e.g., a single source IP address, as show in line 6 of Listing 5.4.

```
for SUBNET in $(seq 2 1 33)
do
    ip xfrm state add \
        src 10.1.0.1 dst 10.1.0.2 \
        proto esp spi 0x$SUBNET mode tunnel aead "rfc4106(gcm(aes))" 0x77... 128 \
        sel src 10.0.0.$SUBNET/32
done
```

Listing 5.4: iproute2 command to create 32 SAs

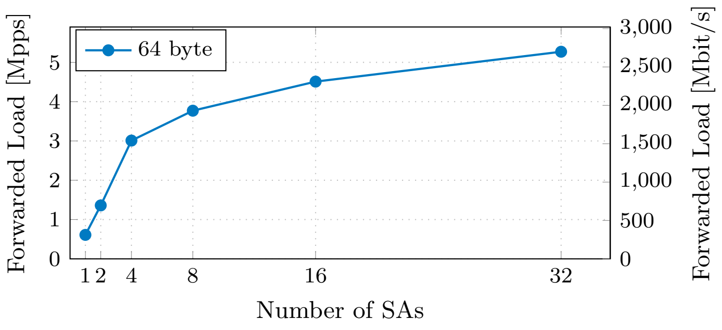


Figure 5.8: IPsec forwarding rate with configuration 1, 64 byte packets and multiple SAs

Figure 5.8 shows the results of this optimization. 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 the highest measured rate of 5.27 Mpps and 2.7 Gbit/s. This sublinear growth shows, that there are other bottlenecks in the send path, which do not get optimized by this distribution scheme.

5.5 OpenVPN

OpenVPN is a cross-platform, open-source VPN suite [29]. It defines its own protocol, wire format and includes a multiplexing layer. Its core components are the same across the supported platforms, only the networking layer is adapted to the specific operating system. On Linux the socket API and virtual TUN/TAP devices are used. This approach allows operation as a purely user-space process, without the need for kernel modules or custom drivers.

OpenVPN is very modular and adaptable to the situation at hand. Both TCP and UDP can be used as transport protocol. Through configuration files most operational aspects can be modified. Encryption, authentication or replay protection can be selectively switched off.

#	Cipher	Auth	Replay Protection
1	AES-256-CBC	HMAC-SHA256	yes
2	AES-128-CBC	HMAC-SHA256	yes
3	AES-128-CBC	none	yes
4	none	none	no

Table 5.3: Tested OpenVPN configurations

The cryptographic primitives are provided externally be the OpenSSL library, on which OpenVPN has a runtime dependency in form of a shared library. Due to the lack of a cross-platform interface for threads, OpenVPN historically is single-threaded. I/O is multiplexed with the operating system's native mechanism like poll, select or kqueues.

5.5.1 Tested Configuration

The only OpenVPN binary supplied by the distributions package manager is an optimized build without debugging symbols. To get more detailed stack traces and resolved function names in our measurements, we build our own binary with the flags shown in Line 19 of Listing 5.5. The remaining preparation steps for the DuT are also given there. In a preliminary test between the distributed and our version, we could not find a performance difference. The OpenVPN version build is 2.4.6 with OpenSSL 1.0.2g as backend.

We configured OpenVPN in L3 tunnel mode with TUN devices, since this implements our site-to-site scenario. Following common practice, packets are delivered over UDP, because it is faster than TCP and avoids problems with window sizes and packet reordering. The exact configuration can be found in Listing 5.6.

For the cryptographic parameters we choose ciphers that match the ones used in IPsec and WireGuard most closely. ChaCha20 or AES-GCM is not available, therefore AES-CBC takes their place. While OpnVPN supports a much greater number of ciphers and digest algorithms, most of them are broken or deemed insecure [21] and are not further considered. The high configurability of OpenVPN allows selective application of encryption, authentication and even replay protection. In Table 5.3 all tested combinations are listed.

5.5.2 Results and Analysis

Figures 5.9 (a) to (d) show how OpenVPN setup 1 behaves under an increasing load of packets with fixed size. Each graph contains the results for a specific packet size and

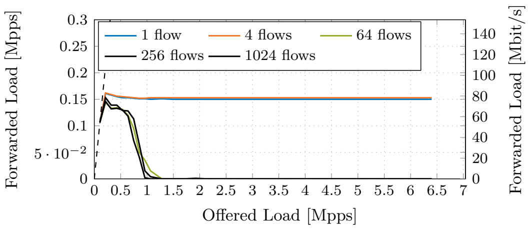
```
echo 1 > /sys/devices/system/cpu/intel_pstate/no_turbo
IF1=ens3f0
IF2=ens4f0
ip addr flush dev $IF1
ip addr flush dev $IF2
ip a add 10.0.0.1/16 dev $IF1
ip a add 10.1.0.1/24 dev $IF2
ip l set mtu 1600 dev $IF1
ip l set mtu 1600 dev $IF2
ip l set up dev $IF1
ip l set up dev $IF2
sysctl -w net.ipv4.ip_forward=1
arp -s 10.1.0.2 68:05:CA:32:44:98

# build openvpn, debug and clean version
wget https://swupdate.openvpn.org/community/releases/openvpn-2.4.6.tar.gz
tar -xf openvpn-2.4.6.tar.gz
cd openvpn-2.4.6
./configure --disable-ssl --disable-lzo --disable-plugin-auth-pam CFLAGS='-g -O2 -fno-omit-frame-point
make -j && make install

cat << EOF > ~/static.key
-----BEGIN OpenVPN Static key V1-----
69fb86b7243dc8ce3d6647d3dcbc1026
...
-----END OpenVPN Static key V1-----
EOF
```

Listing 5.5: DuT peraration steps for OpenVPN

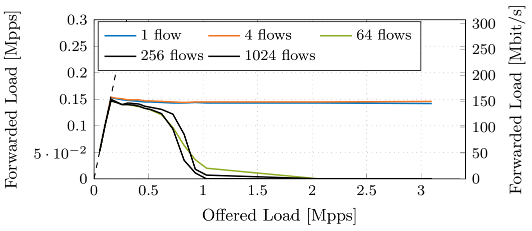
```
openvpn --verb 3 \
     --dev tun --proto udp \
     --local 10.1.0.1 --remote 10.1.0.2 \
     --route 10.2.0.0 255.255.0.0 --ifconfig 192.168.0.1 192.168.0.2 \
     --cipher AES-256-CBC --auth SHA256 --secret /etc/openvpn/static.key
```

Listing 5.6: OpenVPN command for setup 1 on the DuT

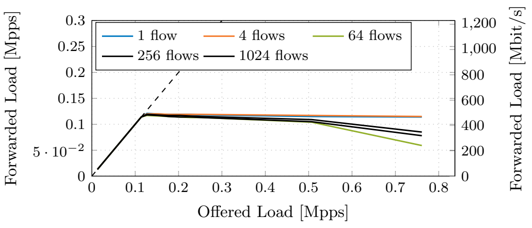
different number of flows. The bottom and top x-axes show the offered load in Mpps and Mbit/s respectively, while left and right y-axes show the processed load.



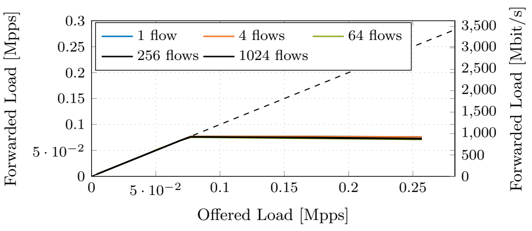
(a) 64 byte packets



(b) 128 byte packets



(c) 512 byte packets



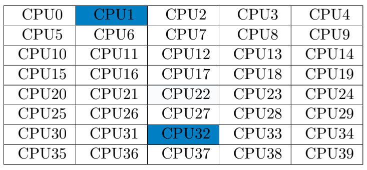
(d) 1510 byte packets

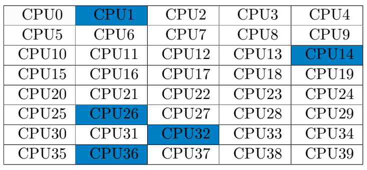
FIGURE 5.9: Forwarding performance of OpenVPN setup 1 with different packet sizes

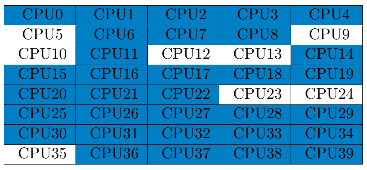
Firstly, it is evident, that the forwarded rate is not very high independent of the kind of offered traffic. It is highest for 64 byte packets at 0.XX Mpps

Another point worth noting is, that the number of flows in the traffic only has a small impact on the performance of OpenVPN. The rates vary very little at a given load and follow the same trend as the load increases. This seems different from what we observed in the baseline benchmark, but can be explained by the low amount of packets per second. The Linux router starts dropping packets at 1.06 Mpps for single-flow traffic, while OpenVPN generally processes only a fraction of this. This rules out the router as a limiting factor. If it were, the rates for 256 and 1024 flows should be higher.

Lastly, we found a repeating pattern that is independent of the packet sizes but only occurs with a high number of flows. From 0 Mpps to 0.12 Mpps the application processes each packet without dropping any. After that it can not keep up with the demand and stays nearly stable up to around 0.5 Mpps load. Then it breaks in drastically and stops forwarding at 1 Mpps load. This can be explained by looking at the DuT's CPU utilization at a per-CPU level during the runs. In the heatmap in Figure 5.10 this is visualized through the shading of each field. The higher the load on a core was, the more intense it is colored. For one flow we see two utilized cores, one doing the network stack operations as in the baseline benchmark and one core for the single-thread OpenVPN process, as expected. But as the number of flows increases, more cores start serving NIC queues and the OS is pressured to find a free CPU for the OpenVPN user-space application. The Linux scheduler by default prioritizes user-space processes lower than IRQ handlers, therefore it will not schedule the VPN process under overload, resulting in the observed 0 Mpps processing rate. In the next paragraph we expand on the drawbacks that user-space applications face, compared to native kernel code or modules.


(a) 1 flow


(b) 4 flows


(c) 64 flows

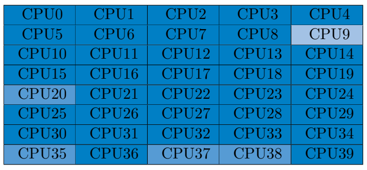

(d) 256 flows

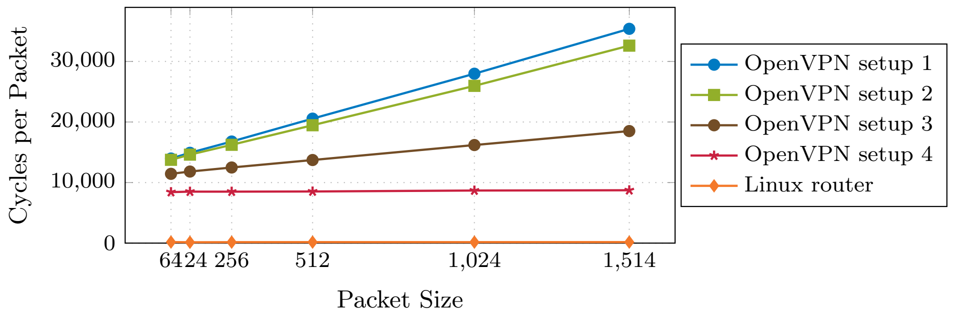
Figure 5.10: CPU utilization of the DuT with setup 1 during 1000 Mbit/s load

Cycle Analysis

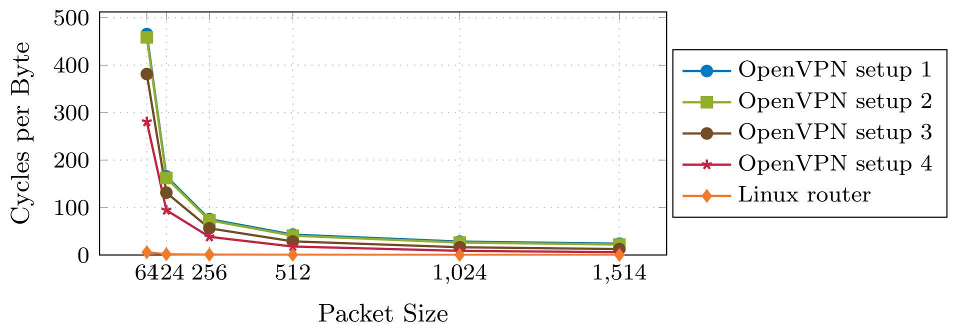
Next we analyze the efficiency of OpenVPN by calculating various relative metrics.

From our previous benchmarks we gather, that the impact of the number of flows is minimal for OpenVPN, as its processing rates are orders or magnitudes lower than what a Linux router can handle. To prevent negative interference from the IRQ handler we use single-flow traffic as the load, which is received on a single Rx queue on the DuT. To be able to make the most accurate attribution of CPU cycles, we also bind the OpenVPN process to a fixed core, so that is not accidentally scheduled on the same core the interrupts are handled. For cycle measurements it is ideal if the applied load is just right, so that the DuT achieves its maximum processing rate, but is not yet overloaded. Since this rate is dependent on the packet size, we determined it for each setting.

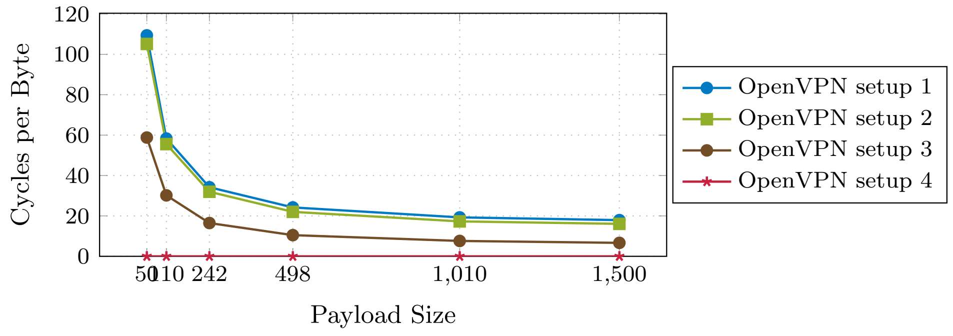
Figure 5.11 (a) shows the cycles spend in total per packet processed. Firstly, it can be seen that the costs of OpenVPN absolutely dwarfs the overhead incurred by the underlying network stack, which is nearly constant at 140 - 170 cycles per packet. This again confirms our assessment, that the bottleneck of OpenVPN does not lie within the Linux network or driver stack. Choosing the AES-256 cipher (setup 1) with a larger



(a) Cycles per packet



(b) Cycles per byte of packet



(c) Cycles per byte of payload

FIGURE 5.11: Number of CPU cycles spend by OpenVPN under peak load

block size compared to AES-128 (setup 2), incurs a measurable overhead of 1.5% to 8%. The overhead is smallest for 64 byte packets (+212 cycles) and increases with packet size (+2780 cycles for 1514 byte). A much bigger impact has the data integrity mechanism. Disabling HMAC-SHA256 signatures reduces the costs by ∼40%. This rather large value can be explained by the way OpenVPN calculates the signature: it is essentially a second, separate pass over the encrypted payload and not generated on-the-fly [30].

Setup 4 has the most distinct curve, as the costs per packet do not increase with larger packet sizes, but remain constant. This can be attributed to the configuration of the

Percentage	Symbol
11.60%	[k] entry_SYSCALL_64_fastpath
10.37%	[k] entry_SYSCALL_64
4.65%	[k] _raw_spin_lock
3.95%	[k] do_sys_poll
3.37%	[k] _raw_spin_lock_irqsave
2.90%	[k] copy_user_generic_unrolled
2.31%	[k] tun_chr_poll
2.26%	[k] fib_table_lookup
2.05%	[k] ___lock_text_start
1.89%	[k] ___fget_light

Table 5.4: Top 10 most run functions in OpenVPN setup 4

setup, which does neither encrypt data, nor does it generate signatures. With this work minimizing setting, the static overhead of handling a packet in OpenVPN has been found at 8500 cycles per packet. Unless this overhead cannot be reduced, the maximum processing rate lies at $\frac{2.2 \text{ GHz}}{8500 \text{ cycles}} = 0.26$ Mpps on the tested hardware. In Paragraph 5.5.2 we further analyze this overhead.

Figures 5.11 (b) and (c) give more insight on the dynamic processing costs. In (b) the required cycles per byte of packet data, depending on the total packet length, are displayed. The sharp decrease at 128 and 256 bytes explains, why OpenVPN is only able to achieve high throughput at large packet sizes. Figure (c) refines this metric by subtracting the static per-packet overhead and by ignoring the Ethernet header fields in the payload calculation. In a L3 VPN they do not get encrypted. The resulting values represent the costs of the cryptographic operations per by byte of data. It can be seen, that these are still comparatively huge for small data chunks at 60 to 100 cycles/byte, but amortize with larger buffers to under 20 cycles/byte. Compared to other benchmarks and implementations this is a silgthly high value [2], but confirms that the encryption is not the main bottleneck of OpenVPN, at least not a larger packet sizes.

Drawbacks of Running in User-space

OpenVPN enjoys higher portability across operation systems by completely running in user-space, as opposed to direct kernel integration like IPsec. There are however, some drawbacks that come with this architectural decision.

Switching from user-mode to kernel-mode

OpenVPN receives and sends its data via the UNIX socket API, as it has no low-level access to drivers from user-space. The API is implemented as system calls (syscalls), where the privilege level is temporarily bumped to kernel-mode and kernel functions are run. Upon entry of such a syscall, the validity of the parameters have to be checked and the CPU registers of the calling process are saved. This separation is important for security, but incurs an overhead if system functions are frequently used. In Table 5.4 we can see, that in fact 13% of the processing time is spend in the entry functions `entry_SYSCALL_64_fastpath` and `entry_SYSCALL_64`. Following the call chain, we found that these come from calls to `read()`, `sendto()` and `poll()`.

Cost of Address-Space Separation

Another overhead comes from the way sockets are implemented on Linux. Upon a `send()` syscall the data is not immediately given to a NIC and send out on the wire. To gain optimizations from batching and reduce interrupt rates, the data is first gathered in the sockets send buffer and processed by the kernel later at an opportune moment. Figure 5.12 visualizes this concept.

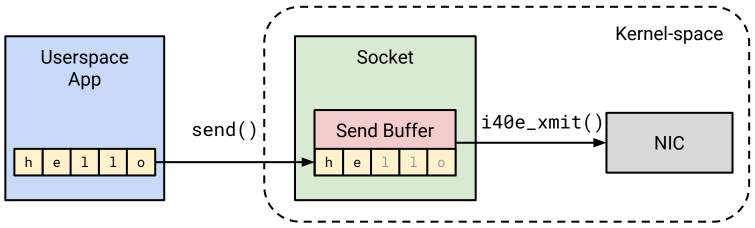


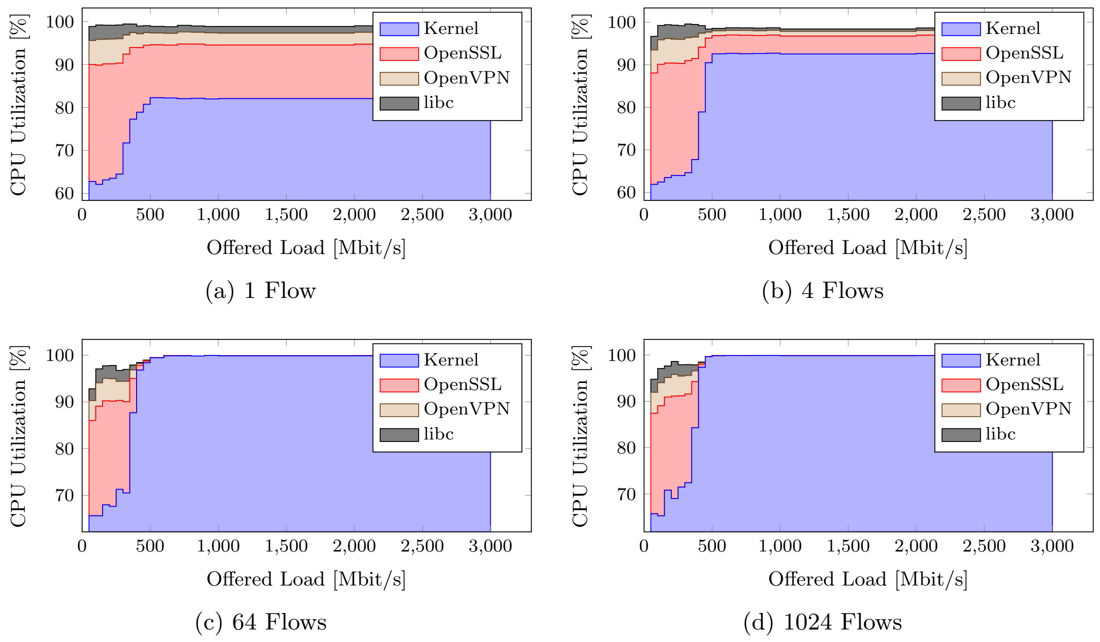
Figure 5.12: Socket send buffers on Linux

Since the API specifies that a user-supplied buffer can be reused after a successful `send()` call, the data actually has to be copied into the socket's buffer. The converse is true for received data, which is copied from driver memory into the user address space. The costs of this memory copying manifests in the entry `copy_user_generic_unrolled` of the performance measurement shown in Table 5.4, where 2.9% of the cycles are spend.

5.5.3 OpenSSL Benchmark

OpenVPN does not supply its own cryptographic primitives, but relies on the high-level OpenSSL API named EVP (EnVeloPe) for all cryptographic operations. In Section 5.5.2 we show that encryption takes a non-negligible amount of time per packet.

Nonetheless, there is some benefit in having a detailed look at the underlying primitives. Firstly, OpenVPN does not support every cipher implemented by OpenSSL. In particular the AES GCM variants are not exposed in static key mode. By benchmarking them too, we gain a broader view and can make a prediction about potential improvements.



(a) 1 Flow (b) 4 Flows (c) 64 Flows (d) 1024 Flows

Figure 5.13: CPU utilization of DuT running OpenVPN setup 1 with 60 bytes packets and varying flows

Another reason is, that neither OpenVPN, nor OpenSSL provide a switch to en- or disable the AES-NI accelerated functions, or even a simple indication if they are used. Other research shows that the availability of this instruction set extension is crucial for good performance [47][34]. In this benchmark we can explicitly control this and observe the effects.

Listing 5.7 shows how to modify the OpenSSL processor capabilities vector [20] by negating the 57th bit, so that only the non-AES-NI supporting implementations are selected.

```
OPENSSL_ia32cap="~0x200000000000000" openssl speed -evp <cipher>
```

Listing 5.7: Snippet to benchmark OpenSSL with AES-NI disabled

In Figure 5.15 the results are displayed for a selection of ciphers and how they perform on different block sizes. The left side shows the encryptions speeds with AES-NI enabled, on the right side it is disabled. Several conclusion can be drawn from this benchmark. The 128-bit block size ciphers are always equal to or faster than its 256-bit versions, but except for AES-GCM, the difference is within 10%, which matches the results in Section 5.5.2. Fastest by a wide margin are the newer GCM variants, which are not available in OpenVPN static key mode. They outperform CBC modes by a factor of

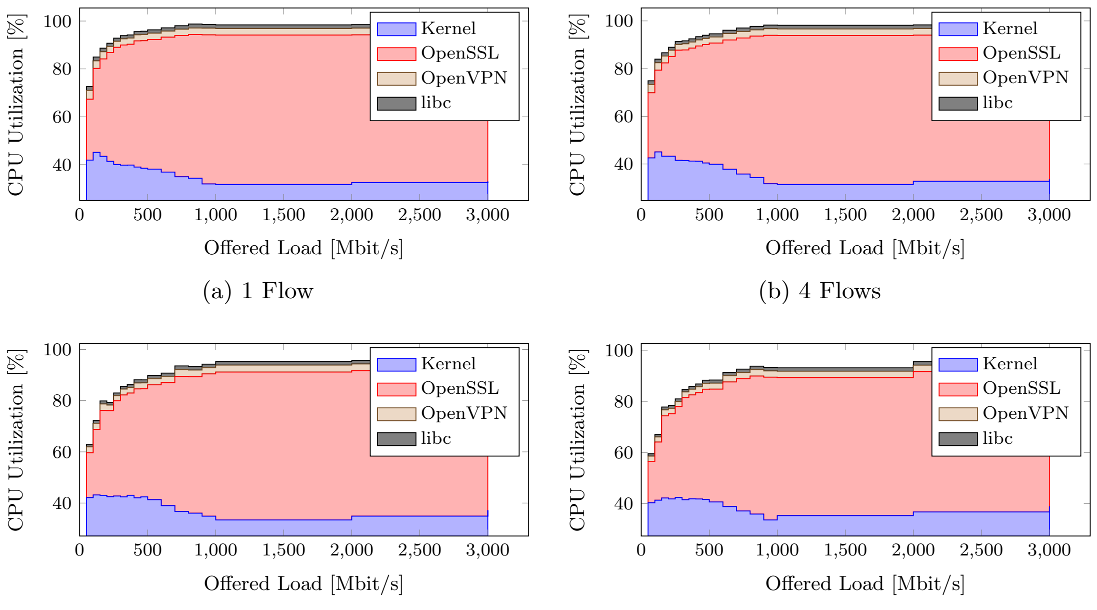


FIGURE 5.14: CPU utilization of DuT running OpenVPN setup 1 with 1514 bytes packets and varying flows

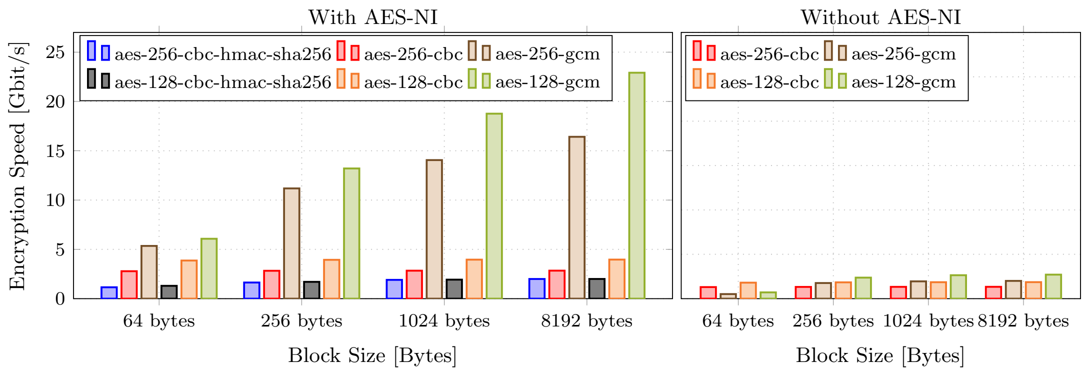


FIGURE 5.15: OpenSSL encryption speeds for AES ciphers depending on AES-NI support

1.5 to 10. We can confirm that AES-NI is indeed very important to performance. No cipher implementation even reaches 2.5 Gbit/s without this instruction set extension.

5.5.4 Multiple Instances

The single-threaded design of OpenVPN limits its performance to very low levels compared to IPsec and WireGuard, as some VPN tasks, e.g. encryption, simply need raw CPU processing power to scale.

While multi-threading has been on the development roadmap of the official OpenVPN application since 2010, very little has change since then. According to the developers, this is partly to blame on the current event-system implementation, which is only partly asynchronous and requires that, e.g. resource freeing, happens synchronously. They estimate that introducing multi-threading requires a complete rewrite of the I/O system and have postponed this to OpenVPN 3.0. For now this leaves users with multi-processing as a scaling vector. This is not a build-in feature of OpenVPN, but has to be setup manually with external tools. The approach is conceptually similar to the multi-SA technique for IPsec presented in Section 5.4.1: The incoming traffic is split up according to its destination address via multiple routes and processed by different processes. In this case, we spawn multiple OpenVPN instances, each handling a distinct /32 subnet. Contrary to IPsec, some additional steps and workarounds for limitations in OpenVPN are required, as shown in Listing 5.8.

```
 1  ethtool -L ens3f0 combined 4
 2  IP_LOW=2; CPU_LOW=4;
 3  for i in $(seq 0 1 $COUNT)
 4  do
 5      ip a add 10.1.0.$(expr $IP_LOW + $i)/24 dev ens4f0
 6      taskset -c $(expr $CPU_LOW + $i) openvpn --verb 3 \
 7          --dev tun$i --proto udp --cipher AES-256-CBC --auth SHA256 \
 8          --local 10.1.0.$(expr $IP_LOW + $i) --remote 10.1.0.254
 9          --route 10.2.0.$(expr 2 + $i) 255.255.255.255
10          --ifconfig 192.168.0.1 192.168.0.$(expr 2 + $i)
11          --secret ~/static.key &
12  done
```

Listing 5.8: Script to set up multi-process OpenVPN

Firstly, we reduce the number of NIC queues to 4 and bind them to the first four cores, so that the interrupt handlers do not interfere with the OpenVPN processes. In Line 5 additional IP addresses on the outgoing interface have to be configured for each instance. Otherwise they would fail with an `EADDRINUSE` (address already in use by other socket) error. With `taskset` we prevent unnecessary CPU migrations and contention for processing power by cleanly separating them of the kernel threads. Having each process bound to a different CPU core limits energy efficiency as these cores can not be put in low power states during low load, but this may not be a concern in high throughput setups.

Figure 5.16 (a) shows the improvements in processing rate that can be made, depending on the numbers of processes spawned. The number of flows in the offered traffic exactly matches the number of processes, to get an even work distribution. Since each worker process is essentially independent with its own TUN device, route and IP address, we see the expected near linear speedup. For traffic of 64 byte packets the rates increase from

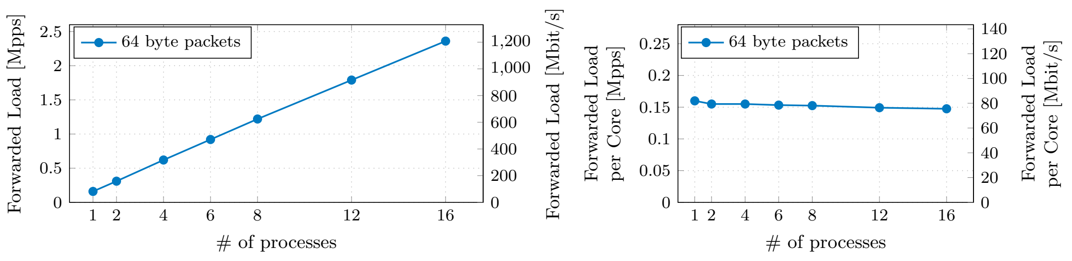


Figure 5.16: Processing rates with multiple OpenVPN instances

the already known 0.16 Mpps in single-instance mode, to 2.36 Mpps with 16 processes. The throughput consequently also grows to 1.2 Gbit/s. This means the DuT is able to handle an entire Gigabit link of incoming traffic. To score the effectiveness of additional resources and to forecast the scaling capabilities beyond 16 processes, the processing rates per core are displayed in Figure 5.16 (b). While there is a measurable reduction, especially in the step from one to two processes, the rate only decreases marginally after that and stays close to 0.148 Mpps per added process. Provided one additional free CPU cores, it would be possible to scale this up further.

Caveats

While some performance gains can be made, this approach is not without serious drawbacks. Firstly, the incoming traffic must be evenly distributed on the configured routes, since a single instance is not faster than before and can still be the bottleneck. In real-world networks this often not the case as a few "elephant" flows dominate bandwidth-wise. Since the filter mechanism is basically the Linux routing table, the granularity of a filter is quite low. /32 subnets (a single destination IP) is the finest setting. Consequently this means, that the uplink capacity to a single host is still limited to 0.16 Mpps. In an environment where many hosts push data to a single server, this approach could be much less applicable. There also is an additional configuration & maintenance burden to be taken. Each instance needs a free IP address and has to be started, monitored and stopped. Lastly, this setup is completely static and does not react sudden traffic changes in form or amount. A smarter implementations could monitor the processed traffic and continuously update the filter routes or scale the number of processes.

5.6 WireGuard

WireGuard is a new L3 VPN product developed by Jason Donenfeld [8]. It defines its own protocol and comes with a reference implementation as a Linux kernel module. At the time of writing integration into the mainline Linux kernel is underway.

The 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: ChaCha20-poly1305. Feature negotiation of, e.g., options or compression is also absent, making backwards compatibility with future impossible. The handshake and key derivation mechanism even includes a hash of the protocol version number [10], 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 [35].

WireGuard also forgoes the layered approach to key exchange found in IPsec. The identities of peers (ed25519 public keys) have to be exchanged out-of-band, e.g., over SSH or E-Mail. In the Diffie-Hellman authenticated handshake these identities are verified and temporary session keys providing perfect forward secrecy are derived. WireGuard relies on the IK handshake of the Noise protocol [39], a suite of verified key exchange and handshake protocols for various purposes. The security properties of the WireGuard protocol have been formally verified by Donenfeld and Miller in 2017 [6].

On the wire level, WireGuard defines just 4 message types. Two for the handshake initiation and response, one for a DoS mitigation cookie reply and one for the actual data exchanges. All packets are encapsulated in UDP to sidestep the problems of TCP-over-TCP tunneling and gain higher performance.

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 lesses the burden on the administrator and user significantly. 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 usable by everyone.

5.6.1 Modifications made to accommodate Test setup

In contrast to the other VPNs, WireGuard does not have a static key or stand-alone configuration, where it can be run without a corresponding instance on the other side. We do not want to introduce a second host between the load generator and the DuT, because then the traffic would have to be processed twice, without any possibility for us to determine where exactly the bottlenecks lie. To get the most accurate measurements, the in and out links should be directly connected to the load generator.

The first hurdle lies in the design of the handshake and key derivation algorithm, requiring a second peer participating to reach a state where packets get encrypted and forwarded. Even running a second peer in the beginning only to complete the handshake would not solve this problem. WireGuard uses a passive keepalive mechanism to detect broken connections or unresponsive peers. If for `Keepalive-Timeout` (10 sec) + `Rekey-Timeout` (5 sec) no kind of response is seen, this peer is assumed to be dead and after `Rekey-Attempt-Time` (90 sec) no further packets would get sent, preventing any kind of long-term measurements.

To work around these issues, we modified the source code of WireGuard. The changes we made are as small and un-intrusive as possible and do not influence the performance, as confirmed by counter-checks. We decided against removing any checks in the hot code paths that normally ensure a session validity and did also not replace peer state lookups with a fake static peers and keys. Instead, we simply emulate responses to handshake initiations as soon as they are send out, by calling the same functions that would be triggered by receiving a real response. This advances the internal state machine so a valid and ordinary session is created.

The keepalive timeout is mitigated by prolonging its underlying timer by 24 hours instead of disabling it after the handshake. In the send path WireGuard checks if this timer is already running and otherwise activates it with the aforementioned 15 seconds.

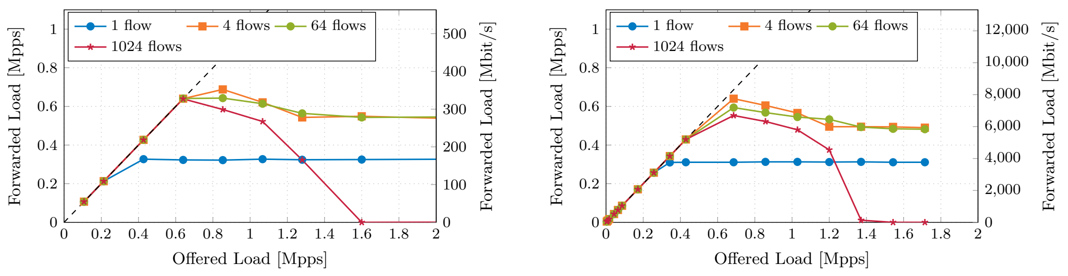
Listing 5.9 shows the complete modifications made. Overall this ensures, that we benchmark a real-world implementation with the same code paths taken and all necessary checks performed. We applied these to the WireGuard base version with commit hash cbd4e34. The patched code is available in our forked repository [46] under commit 798d234.

5.6.2 Performance Analysis

We measure the forwarding rate of WireGuard with our standard benchmark traffic. Figure 5.17 displays the processing rates of the DuT with 64 byte and 1514 byte packets.

```
diff --git a/src/send.c b/src/send.c
index d3e3d752..c963cbb4 100644
--- a/src/send.c
+++ b/src/send.c
@@ -40,6 +40,28 @@ static void packet_send_handshake_initiation
timers_any_authenticated_packet_sent(peer);
socket_send_buffer_to_peer(peer, &packet,
    sizeof(struct message_handshake_initiation), HANDSHAKE_DSCP);
timers_handshake_initiated(peer);
+
+       down_write(&peer->handshake.lock);
+       peer->handshake.state = HANDSHAKE_CONSUMED_RESPONSE;
+       index_hashtable_insert(&peer->handshake.entry.peer->device->index_hashtable,
+                               &peer->handshake.entry);
+       up_write(&peer->handshake.lock);
+       if (noise_handshake_begin_session(&peer->handshake, &peer->keypairs)) {
+           timers_session_derived(peer);
+           timers_handshake_complete(peer);
+           timers_any_authenticated_packet_received(peer);
+           timers_any_authenticated_packet_traversal(peer);
+       } else {
+           printk(KERN_INFO "Peer␣handshake␣failed\n");
+       }
+   } else {
+       pr_debug("create␣init␣failed");
    }
}

diff --git a/src/timers.c b/src/timers.c
index d3150de2..22a509f1 100644
--- a/src/timers.c
+++ b/src/timers.c
@@ -137,7 +137,7 @@ void timers_any_authenticated_packet_sent
void timers_any_authenticated_packet_received(struct wireguard_peer *peer)
{
if (likely(timers_active(peer)))
-       del_timer(&peer->timer_new_handshake);
+       mod_timer(&peer->timer_new_handshake, jiffies + 24 * 60 * 60 * HZ);
}
```

Listing 5.9: Patches to make WireGuard standalone



(a) 64 byte packets (b) 1514 byte packets

Figure 5.17: Encryption rates of WireGuard at different packet sizes and varying number of flows

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 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 5.6.2.

Another observation is, that the size of the processed packets has no impact on performance. Subfigures (a) and (b) 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, else we would see lower rates for larger packets. In the following we explore where most processing power is spend on.

The observed performance pattern is tied to the threading model of WireGuard, as shown in Figure 5.18. It consists of three stages that perform the VPN operations.

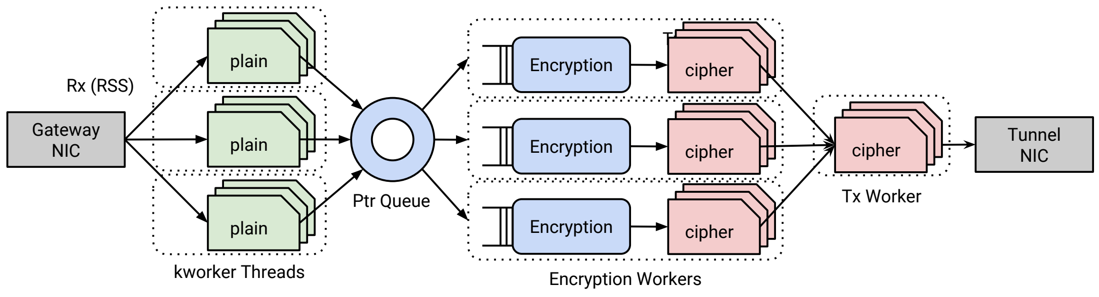


Figure 5.18: WireGuard threading model for encryption

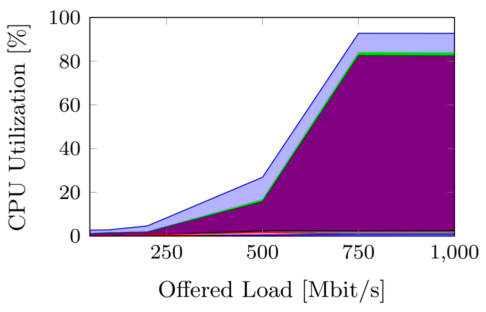
kworker threads serve the gateway NIC and receive packets from it. After determining through a routing lookup, that a packet is eligible for processing, they transmit it through a virtual WireGuard device via the generic Linux driver `xmit` function [7]. Despite the name of the function, the packet is not really send out in this step, but only enqueued 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.

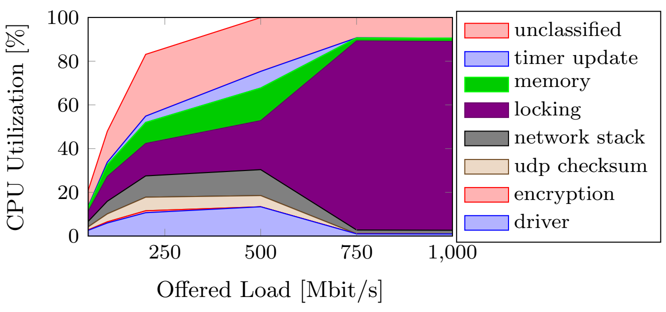
There is only a single Tx worker per peer. It collects all encrypted packets and transmits them via the normal UDP socket API.

With three pipeline stages, this design has three potential choke points, which we investigate further in the next paragraph.

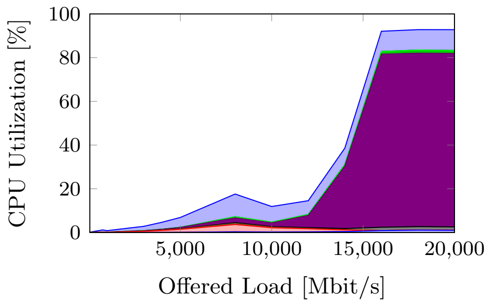
Cycle Analysis



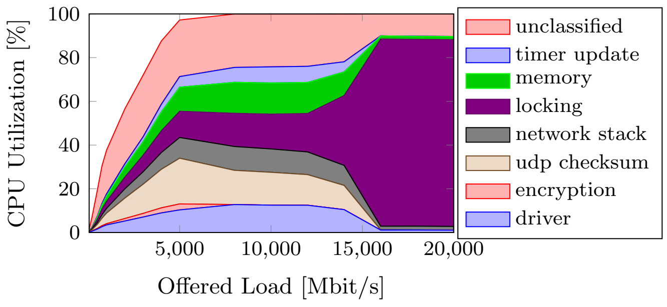
(a) 64 byte packets, kworker & encryption worker



(b) 64 byte packets, Tx worker



(c) 1514 byte packets, kworker & encryption worker



(d) 1514 byte packets, Tx worker

Figure 5.19: CPU utilization of WireGuard with 1024 flow traffic

Figure 5.19 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 solely it can cause a total overload of WireGuard. The first column show a CPU 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. In the second column the Tx worker is displayed.

One can immediately notice, that the most 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. We found, that spinlocks have to be taken in a pipeline stages. More importantly, they are also used to implement mutual exclusion in the pointer rings, that connect the pipeline stages.

In the definition of the `ptr_queue` structure we find, that the head and tail of the queue are protected by two spinlocks [11]. For all en- or dequeue operations, the respective lock has to be taken. The `ptr_queue` is used to implement the crypt and tx queues in WireGuard.

The kworker code paths contain multiple locks. In the `xmit` function an incoming packet is checked for validity and moved to the `staged_packet_queue` of the peer. This queue is guarded by a spinlock. In 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. Additionally, at the end of the `xmit` function, the kworker call the `packet_send_staged_packets` function [12], which looks up the peer state and nonce. At the beginning of this function, all currently staged are dequeue, which again requires the lock.

The encryption and Tx worker are lock-free, apart from the aforementioned pointer rings.

Together these sources explain the baseline locking load in Subfigures (a) and (c), 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 spinlocks.

5.6.3 GSoC Performance Improvements

Google Summer of Code (GSoC) is an annual program to bring students to open-source software development under the guidance of a mentor. During 2018 the three students Jonathan Neuschäfer, Thomas Gschwantner and Gauvain Roussel-Tarbouriech participated in the program and worked on WireGuard with the goal to improve performance and do measurements.

They created a number of commits, of which many smaller ones got merged. In their final reports, they name time constraints and setup difficulties as why they could not perform the planed extensive profiling and performance measurements [24, 36]. Consequently they did not include detailed documentation in their report about the performance improvements they could achieve.

Changes	Commit	Comment
`tg/mpmc_ring`	0df81af3d	Broken: compile error
`jn/mpmc-wip`	bb76804	Broken: Disables preemption indefinitely
`tg/mpmc-benchmark`	6f909b2	Includes `jn/mpmc-wip` and fixes preemption
Prefix-Trie byte order	5e3532e	Negligible time is spent on lookups
NAPI	6008eacb	Only applies to Rx path

Table 5.5: Git hashes of WireGuard GSoC commits

Table 5.5 gives an overview of the non-trivial changes made. In the prefix trie, the data structure used to implement the whitelist of allowed IPs, the byte order in which the addresses are stored, was changed to the machine native little-endian format. This is more of a code quality fix than a performance issue, as our measurements show, that only a negligible amount of time is spend on lookups. Neuschäfer adapted the packet receive functions to the New API (NAPI) of Linux on the Rx path. The main benefits of the NAPI are interrupt moderation and packet throttling, but these only apply to the receive (decryption) side of WireGuard[1].

MPMC Queue

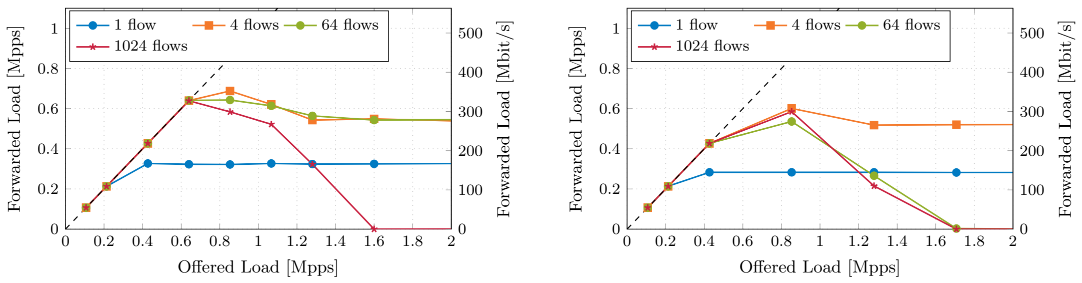
In one of the major tasks, the students focused on improving the packet distribution between the stages and replacing the spin-lock based ring buffer. They choose a lock-free multi-producer multi-consumer (MPMC) implementation based on the ConcurrencyKit [1]. From our measurements in Section 5.6.2, we know that the spinlock ring

[1] The DuT does of course receive plaintext packets, which it then encrypts, but this is handled by the i40e driver of the physical NIC

buffer poses a bottleneck. Therefore it is sensible, to evaluate commits related to this area.

As the students could not finish the implementation in time, it has not been merged into the master branch. There exist three branches which contain MPMC implementations: `tg/mpmc_ring` is the initial version of Thomas Gschwantner, `jn/mpmc-wip` contains the combined efforts of him and Jonathan Neuschäfer and `tg/mpmc-benchmark` extends the previous version by a simple benchmark script and patch to fix scheduling of producers by disabling preemption during enqueue operations. A fix that is needed because of deadlocks and interrupt masking bugs where processors remain in unresponsive state for over 20 seconds. In your tests this occurred regularly and made consistent measurements impossible. Therefore we only evaluate their final version including the preemption fix (`tg/mpmc-benchmark`).

It should be noted that both `jn/mpmc-wip` and `tg/mpmc-benchmark` do not compile cleanly due to missing or wrong header files. The first branch includes a non-existing header, that, in its correct spelling, contains processor specific symbols. After consulting the Linux kernel documentation we concluded that this statement is most likely just a minor mistake and removed it, upon which the code compiled without errors or warnings. The second branch uses the `preempt_enable_no_resched()` function, which has been deprecated since kernel version 3.14. This has been done to prevent modules from interfering with preemption and introduce bugs like the one described above [57]. Since the test systems are on kernel version 4.4.0, we unconditionally replaced the call with its replacement function. Listing 5.10 shows the exact changes made.



(a) Baseline, 64 byte packets (b) Improved, 64 byte packets

Figure 5.20: Performance improvements by the GSoC commits

Figure 5.20 show the improvements of the MPMC replacement compared to the baseline version. It can be seen the the overall performance did not improve, but even decreased. The single-flow rate is now under 0.3 Mpps, while the rate drop that previously only affected 1024 flow traffic, now applies to 64 flows as well. We did not explore if this

```
diff --git a/src/mpmc_ptr_ring.h b/src/mpmc_ptr_ring.h
index da9393e..e035b9e 100644
--- a/src/mpmc_ptr_ring.h
+++ b/src/mpmc_ptr_ring.h
@@ -44,7 +44,7 @@
#include <linux/compiler.h>
#include <linux/errno.h>
#include <linux/log2.h>
-#include <linux/processor.h>
+// #include <linux/processor.h>
#include <linux/slab.h>
#include <linux/stddef.h>

diff --git a/src/mpmc_ptr_ring.h b/src/mpmc_ptr_ring.h
index a4f8344..2cee50a 100644
--- a/src/mpmc_ptr_ring.h
+++ b/src/mpmc_ptr_ring.h
@@ -47,7 +47,9 @@
//#include <linux/processor.h>
#include <linux/slab.h>
#include <linux/stddef.h>
+#include <linux/preempt.h>

+#define preempt_enable_no_resched() preempt_enable()

struct mpmc_ptr_ring {
/* Read-mostly data */
```

Listing 5.10: Patches to compile MPMC commits

stems from an implementation error in the MPMC queue or is and just the performance characteristic of this lock-free implementation.

5.7 MoonWire

This sections presents the benchmarking results of our implemented MoonWire VPN variants. The design and implementation details of them are explained in Section 4. Table 5.6 lists the exact versions of all used software and libraries.

Name	Version
MoonWire	commit b1ba910
Libmoon	commit 57beade
Libsodium	1.0.16
DPDK	17.11
LuaJIT	2.1.0-beta3

Table 5.6: Versions of used software

5.7.1 Variant 1 - Naive Single Instance

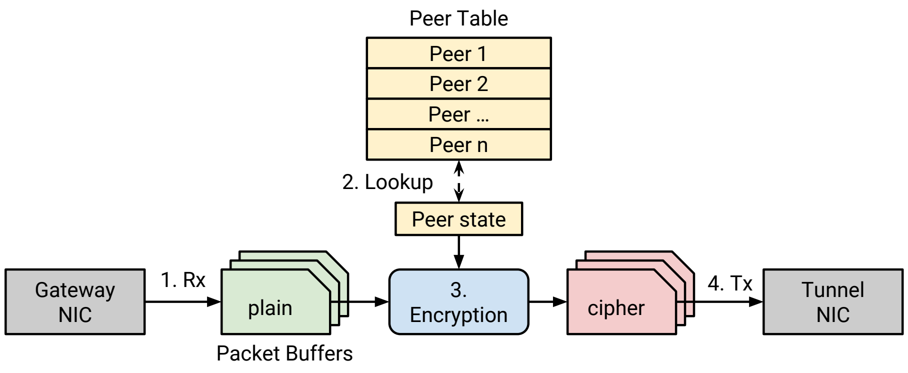
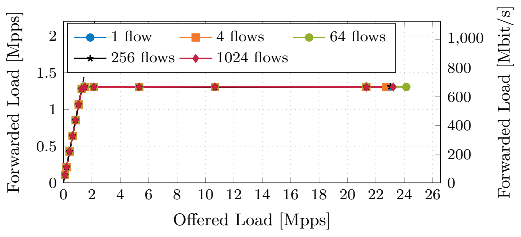
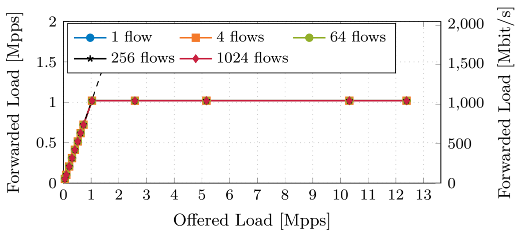


Figure 5.21: Flowchart of MoonWire Variant 1

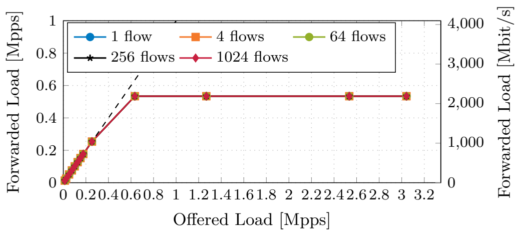
Variant 1 is the most simple and straightforward of our designs, without threads, RSS or other advanced features. A single thread performs all steps of the VPN process. The lower processing rates, shown in Figure 5.22, are therefore not unexpected. Nonetheless it is much faster than OpenVPN, which design it mimics. 64 byte packets get processed at 1.3 Mpps, a factor 10 increase over the 0.16 Mpps of OpenVPN. It also reacts well to overload. Packets that can not be handled are dropped by the DPDK drivers on the NIC hardware level and do not decrease the processing rate of the DuT, as seen by the constant rates after peak. Its absolute peak in terms of throughput is reached with the large 1514 byte packets at 4 Gbit/s. This shows, that a naive and simple implementation is not enough to serve 10G links.



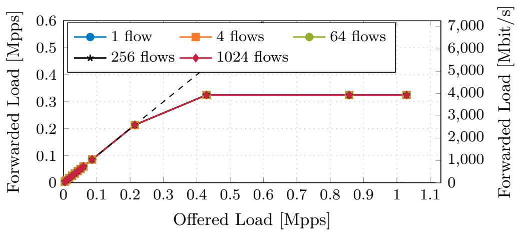
(a) 64 byte packets



(b) 128 byte packets



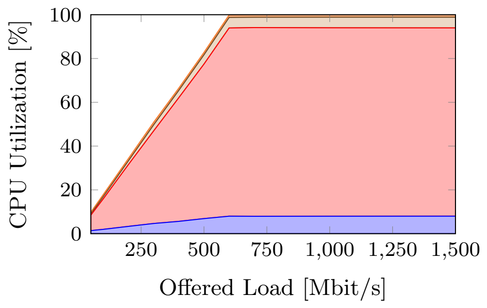
(c) 512 byte packets



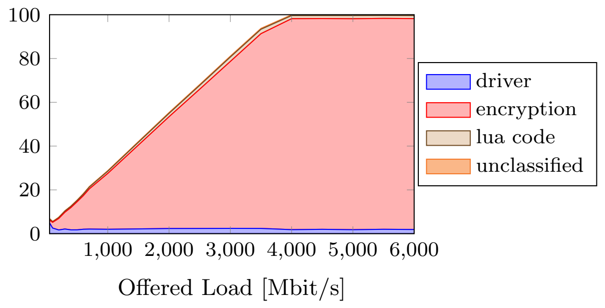
(d) 1514 byte packets

Figure 5.22: Encryption rates of MoonWire Variant 1 with different packet sizes and varying number of flows

We did not find any issues regarding different number of traffic flows. Without RSS, all are handled on one single core, consequently there are no differences in the processing rates between them and the lines in Figure 5.22 overlap perfectly. Packet reordering, neither intra- nor inter-flow, can also not occur: the application fetches a batch of packets, processes them in the received order and then sends them out in this exact order.



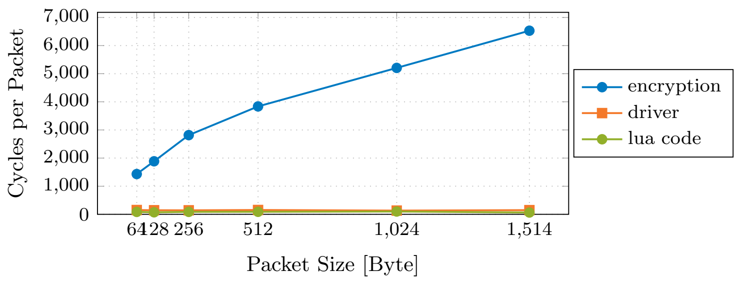
(a) 64 byte packets



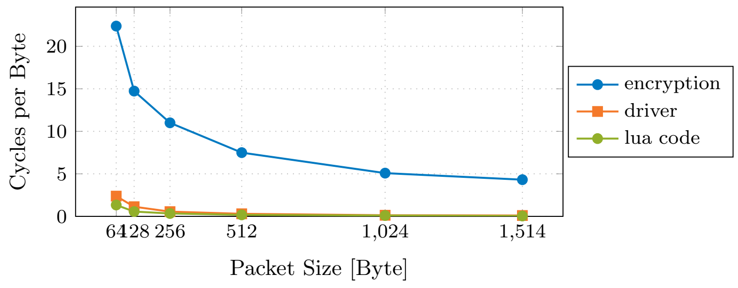
(b) 1514 byte packets

Figure 5.23: CPU utilization of Variant 1

One interesting result is, that the peak rates in terms of Mpps vary depending on the packet size. I.e., its highest for small packets and decreases the more bytes have to be processed. Compared to OpenVPN where we found a static peak at 0.13 Mpps and large fixed costs independent of packet size, that imposed this artificial limit. From this we can conclude, that in Variant 1 the per packet overhead from drivers and such is much smaller and there is simply processing power missing at some load level. With the CPU utilization plots of the worker thread in Figure 5.23 we can confirm this. Overhead from the DPDK network driver and other Lua functions makes up less than 8% and 4.8% of the cycles spent, respectively. At lower rates the CPU is mostly idle, as seen by the large white triangles in the upper left corners. Note, that this does not necessarily mean that the CPU is clocked down or could be used for other tasks. DPDK follows a busy-waiting approach for low latency reaction times and still monopolizes the core. We simply observe a large number of cycles spend in the `rte_delay_us_block` function.



(a) Cycles per packet



(b) Cycles per byte

Figure 5.24: Cycles analysis for MoonWire Variant 1

With 1432 cycles or 85% of the total, encryption consumes the most resources by a wide margin (Figure 5.24(a)). This increases further as the packets get larger. To encrypt 1514 byte packets, it takes 6500 or 96% of the total CPU cycles, completely marginalizing any other costs. Across all packet sizes, the driver code uses only 132 to 152 cycles per packet, much less than the Linux baseline measurement of 2000 cycles.

As a pure user-space application, there is no overhead from syscalls or memory copying from kernel-space. In fact, Variant 1 is truly zero-copy regarding the packet data.

5.7.2 Variant 2 - Independent Workers

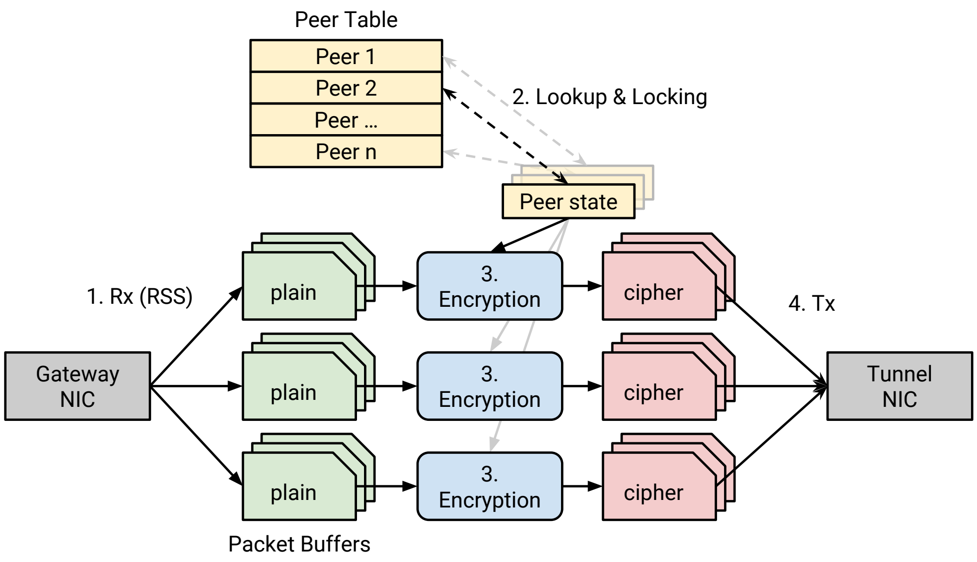


Figure 5.25: Flowchart of MoonWire Variant 2

Variant 2 solves the processing power bottleneck around encryption of Variant 1 by distributing the packets to multiple parallel workers. The distribution starts at the hardware level already by configuring multiple Rx queues combined with RSS. Consequently this variant suffers from the same problems as the Linux router and IPsec, which also rely on RSS. Figure 5.26 shows the familiar picture: Few flows in the traffic result in uneven distribution and lower processing rates.

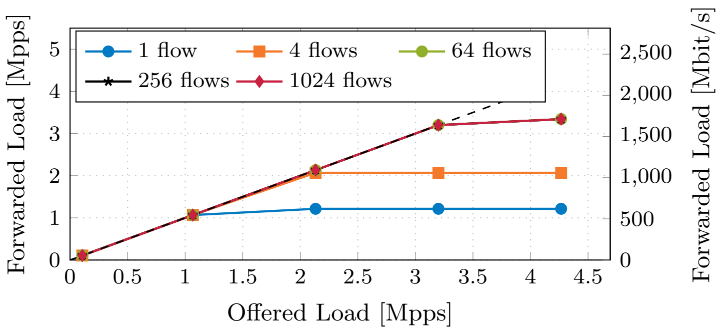
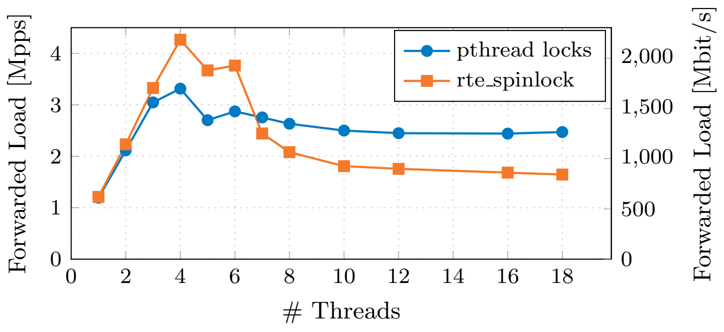
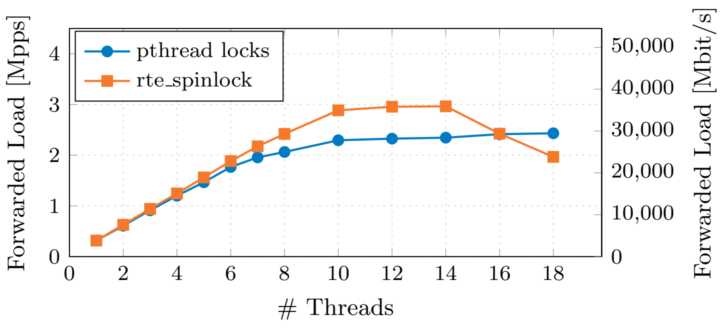


Figure 5.26: Effects of RSS with 64 byte packets on MoonWire Variant 2a with 4 workers

But for traffic with sufficient flows (≥ 64), significant speedups can be measured, as shown in Figure 5.27. In these measurements the number of threads is increased gradually and the individual peak rate is recorded.



(a) 64 byte packets

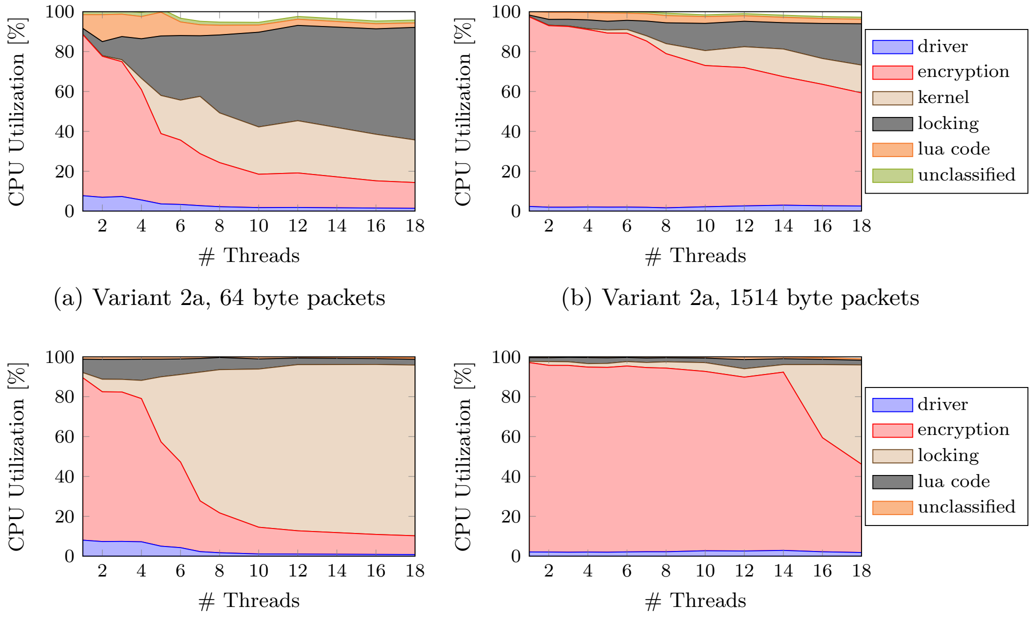


(b) 1514 byte packets

FIGURE 5.27: Scaling capabilities of MoonWire Variant 2a and 2b

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.

Spinlocks have a higher peak rate, but are also more susceptible to high contention. This is a result of their implementation. `rte_spinlocks` internally use atomic compare-exchange instructions in a loop until they succeed, thus implementing busy-waiting. For fewer threads this is faster than relying on the OS's scheduling, but with 7 or more threads, the interference imposes too much overhead and continuously decreases total throughput with more threads. pthread locks on the other hand, out-source some of the work to the OS. The scheduler knows which threads wait for a lock and can suspend them until the lock becomes free. This results in less active contention and a consistent 2.5 Mpps rate with more than 8 threads (Subfigure (a)).



(a) Variant 2a, 64 byte packets

(b) Variant 2a, 1514 byte packets

(c) Variant 2b, 64 byte packets

(d) Variant 2b, 1514 byte packets

Figure 5.28: CPU utilization of a worker thread in MoonWire Variants 2a and 2b

Looking at the CPU utilization plots in Figure 5.28, we can see why Variant 2 only performs well with larger packets and at lower Mpps rates. The plots show the cycle usage of a single thread on a single CPU core, but since this benchmark is recorded with 1024 flow traffic, all workers perform the same tasks.

We can see the general trend, that more threads lead to more time spend 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.

Variant 2b spends increasing amounts of time directly in the lock and unlock functions, while for Variant 2a 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.

Resource intensive encryption is the main struggle of Variant 1. Subfigure (b) and (d) show, that this operation is now well parallelized over the threads, consuming more the 60% of the cycles in most cases. This means, that the concept of distributing the expensive operations generally works.

Driver code again only plays a minor role overall, with 1% to 7% of the time spend, showing that DPDK is also efficient with multiple threads operating on the same NIC.

Overall this variant demonstrates, that having, seemingly local but effectively global, locks does not scale beyond a few CPU cores.

5.7.3 Variant 3 - Work Pipeline

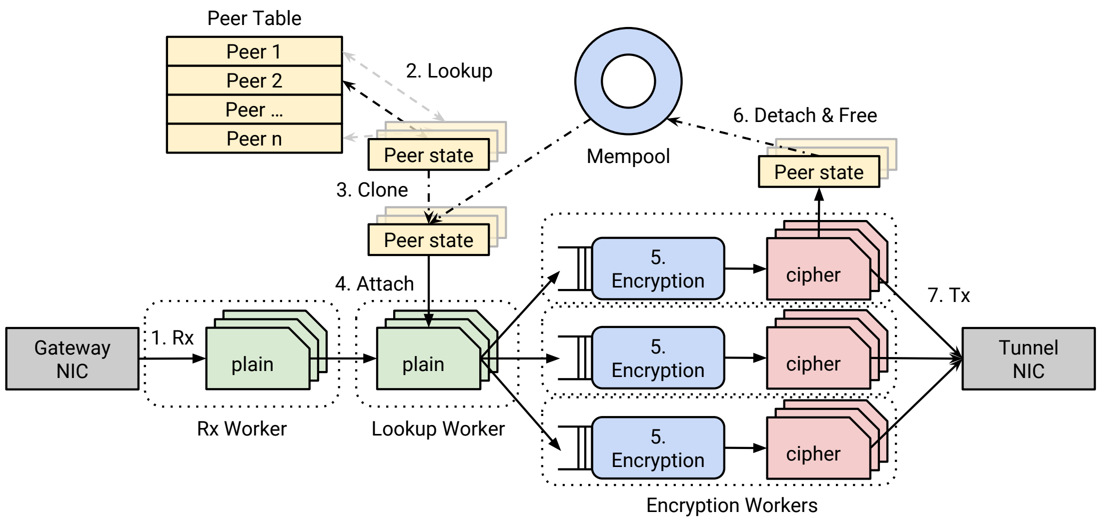


Figure 5.29: Flowchart of MoonWire Variant 3

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 5.30 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 4, 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.

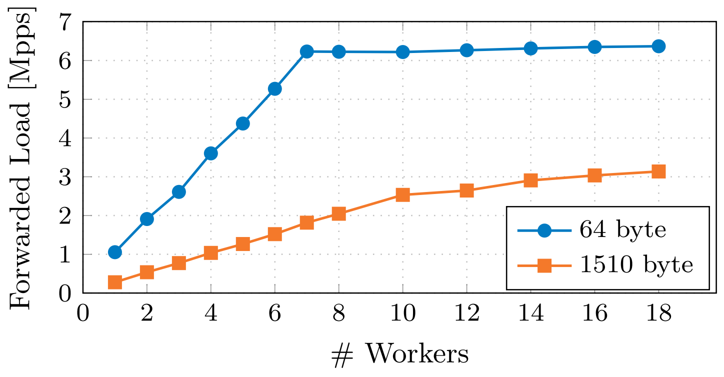


Figure 5.30: Throughput scaling of MoonWire Variant 3 with encryption workers

Figure 5.31 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 a under-utilization of some workers, here the rates are always the same.

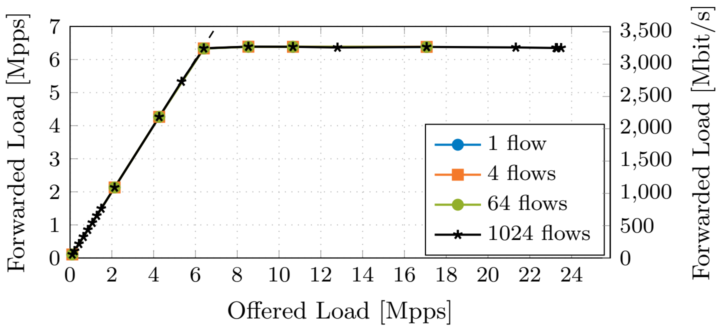
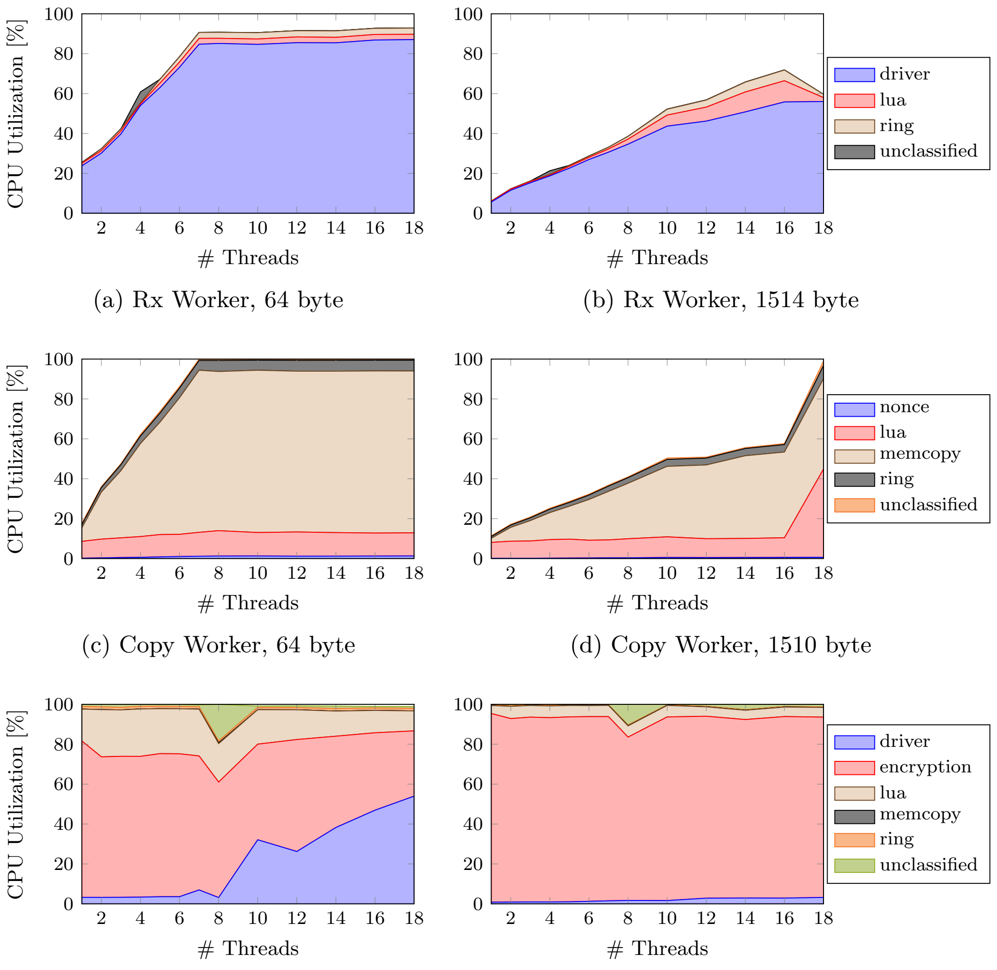


Figure 5.31: Throughput of MoonWire Variant 3 depending on the number of flows

Next we determine which stage is the limiting bottleneck. In the plots in Figure 5.32 we see the CPU utilization of each stage and for 64 and 1514 byte packets.

First, we concentrate on the results for smaller packets, which are in the first column (Subfigures (a), (c), (e)). It can be seen, 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 spend 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 with more NICs and cores, this can be scaled up further.



(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

FIGURE 5.32: CPU utilization in the pipeline stages of MoonWire Variants 3 with 64 and 1514 byte packets

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 field. In future work we could explore hybrid approach, where static field are only referenced.

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. Measuring the impact of this on real world applications and common TCP stacks remains future work.

5.7.4 Comparison

Here we show how our MoonWire variants compare to the other VPN implementations. In Figure 5.33 we present the highest achievable processing rates from our measurements across all configurations.

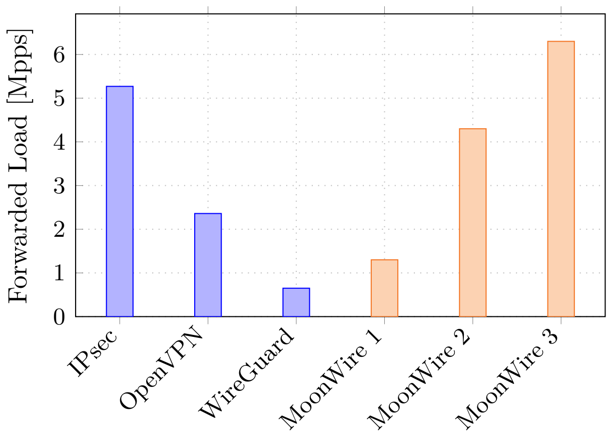


Figure 5.33: Highest measured rates over all configurations

Chapter 6

Performance Model for VPN Implementations

This chapter presents our developed performance model for site-to-site VPN implementations.

Implementations are wildly different in their details, but implement common functionality as part of their VPN operation. The following sections dissect and extract the most resource intensive core tasks of VPN processing. For these core tasks we then define equations that can be used to estimate performance numbers depending on internal and external factors. These equations are derived from experimentation, benchmark results and prior research.

$$c_{total} = c_{driver} + c_{crypto} + c_{synchonization} \tag{6.1}$$

$$forwardingrate = \frac{f_{cpu} \times \beta}{c_{total}} \tag{6.2}$$

The performance of a VPN system can be expressed as a function of the CPU cycles spent per packet processed. The total costs per packet, c_{total}, are sum of the processing costs in the three major areas driver, cryptography and synchronization, as shown in Equation 6.1. We developed these categories through our measurements and explain them in detail in the Sections 6.1 to 6.3. With the total costs, the expected processing rate of a DuT can be estimated by dividing the weighted CPU cycles though the cycle costs per packet. This yields the rate in packets per second, as seen in Equation 6.2. The CPU cycles, f_{CPU}, have to be adjusted, because of two reasons. CPU models vary in their number of cores from 1 to 128 and more. Secondly, VPN implementations do not utilize multiple cores equally well. Therefore, the weighting factor β lies in range $[1, \#CPU\_cores]$ inclusively. Where factor 1 represents the worst case with no multi-core scaling and a factor of $\#CPU\_cores$ the hypothetical best case without any overhead, where each additionally available CPU core adds equal amount of processing power. β can be determined by measuring the speedup when progressively more CPU cores are made available to the VPN application.

6.1 Network Driver & Stack

Before a packet reaches any VPN code, it has to pass up through the NIC driver and OS networking layers. There packet buffers have to be allocated, data has to be received and written to memory, and header fields have to be validated. In a site-to-site setup the packets are not destined to the gateway processing them, but the hosts behind it. Therefore the packets also have to be sent, again passing down through the layers. The costs of these operations is captured in the c_{driver} variable of our model. While a VPN implementation usually has no influence over the decision which network stack is used, a major chunk of the system's performance is decided at this low level. If the NIC driver is slow or inefficient, then even an optimized upper application will not yield faster results.

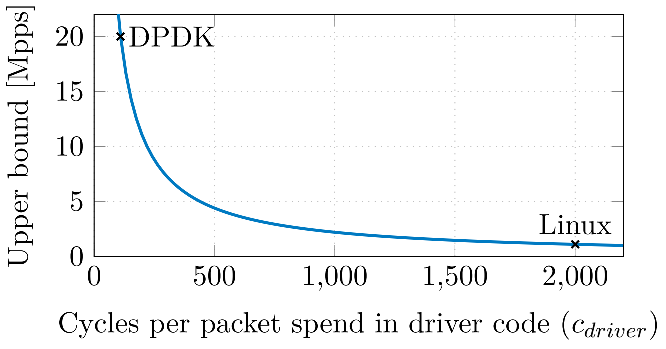


Figure 6.1: Upper bound forwarding rate limitations depending on c_{driver} costs

The severity of this can be illustrated by a simplified example calculation shown in Figure 6.1, assuming a single-core application. Given a total cycle budget of 2.2×10^9 cycles (2.2 GHz Intel E5-2630 v4 CPU), an upper bound on the forwarding rate of an application can be given, depending on the costs of handling the packet in the driver (c_{driver}). For an efficient framework like DPDK, with low driver costs of around 110 cycles per packet, the upper bound for single-core forwarding lies at 20 Mpps. The complex and more general Linux netstack on the other hand, with its i40e driver, requires around 2000 cycles, as seen in Figure 5.4 of our baseline benchmark. This sets the bound to 1.1 Mpps, which approximates the actual, measured rate of 1.006 Mpps for single-flow traffic very well.

6.2 Cryptographic Operations

Once the packets have been received and handled by the driver, they are passed to the VPN application so they can be encrypted.

To be secure against modifications, messages also have to be authenticated with HMACs or AEAD schemes. Former can lead to a second pass over the data, while the later is an integrated solution consisting of a single pass. Equation 6.3 therefore estimates c_{crypto} depending on the packet length l.

$$c_{crypto} = c_s + c_v(l) \tag{6.3}$$

Contrary to the driver costs, which accrue per packet, encryption costs depend foremost on the length of the data, and therefore on the traffic. Equation 6.3 describing c_{crypto} consists of a static portion c_s and a variable part c_v. c_s is independent from the data length and occurs constantly for every packet. It stems from overhead of extending the buffers, function calls and setting up cryptographic state. c_v captures the marginal costs of encrypting an additional byte.

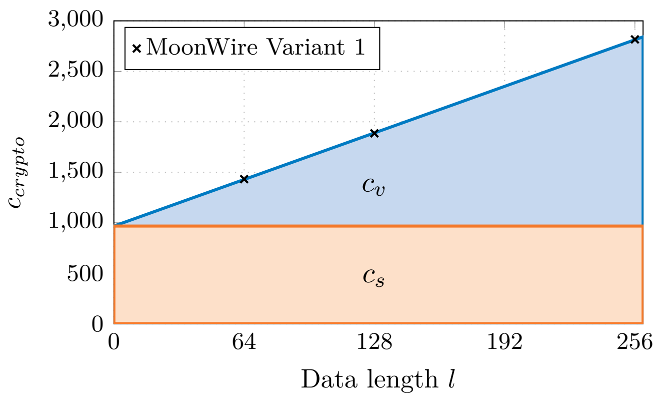


Figure 6.2: Plot of c_{crypto} depending on data length for MoonWire Variant 1

Figure 6.2 visualizes this relation on the example of MoonWire Variant 1, benchmarked in Section 5.7.1. Using the measured cycles of the three smallest packet sizes (64, 128, 256), we perform a linear least-squares fit on the data set, which yields the following linear equation: $c_{crypto}^{MoonWireV1} = 966.782 + 7.21746 \times l$, with a coefficient of determination $R^2 = 99.999$. This result tells us, that the static overhead of calling the different libsodium functions is significant with $c_s = 967$ cycles. The actual application of chacha20-poly1305 cipher then takes $c_v = 7.2$ cycles per byte of data for these small packet sizes.

6.3 Data Structures & Multicore Synchronization

From the cryptography part of our performance model in Section 6.2 we know, that encryption takes a significant amount of processing power. Some VPN implementations aim to achieve higher forwarding rates, by distributing the packets to multiple CPU cores, utilizing the modern many-core CPU architecture. This approach is not without drawbacks. The weak memory model of the x86 architecture (and its successor x86_64) requires the use of synchronization mechanisms, if data is to be shared. These introduce additional overhead not present in a serial implementation, as seen by comparing MoonWire Variant 1 (1.35 Mpps) and Variant 2 configured with a single thread (1.2 Mpps).

From our evaluation and investigation of the source code, we know that the critical data structure is the peer state and its container, the peer table. It has to be both read and written to for every handled packet of a specific peer. In the context of site-to-site setups with few or just one peer, all accesses focus on a single data item.

We have found and implemented the two common approaches to packet distribution and data synchronization.

Locking - A peer state data item is secured with a mutex that has to be taken by threads before they are allowed to read or update any values of it. While this approach it easy to implement with existing library support, it does not adequately scale with many cores and hot resources.

Exclusive access means, that only a single thread can hold the mutex at a given point time, while others must wait or request it at a later point int time. This creates an distinct performance pattern, modeled in Figure 6.3.

As long as there are still free time slots in which the lock is not held by anyone, adding more threads does improve overall performance (Area A). New threads then take these slots up and performance scales linear or nearly linear. But once a lock is saturated and held at any given time point, then no further improvement is possible (Point B). Other threads will have to wait, either actively or passively, making no progress. Further, there

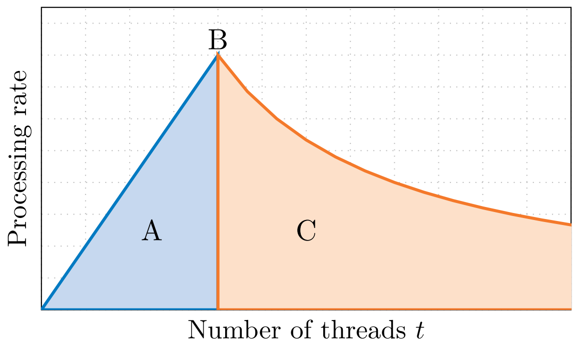


Figure 6.3: Performance of locking depending on the number of threads t

is overhead introduced by each additional thread (Area C). Failed locking attempts (in spinlocks) also consume cycles, as well as the management overhead in the scheduler. The severity of impact of this accumulated overhead depends on the lock implementation and the presence of other rate limiters. For MoonWire Variant 2a which uses spinlocks (busy waiting), the impact is measurable with a 50% processing rate decrease from 4 to 16 threads. Other implementations such as Variant 2b (pthreads mutex) or IPsec (rate limiter) exhibit a less drastic reduction and remain at a nearly constant rate.

Message Passing - As an alternative to locks, message passing can be used for communication. Then the peer state is not shared, but managed by only a single thread, thus synchronization-free. This thread creates messages for the other workers by copying the relevant parts of the peer state and attaching it to a packet, forming stand-alone work items. From the measurements of our MoonWire Variant 3 implementation we know, that the efficiency of this management task is crucial for the overall performance. While the other tasks like packet reception and encryption can be distributed to multiple threads, the management task can not. This naturally leads to a multi-stage pipeline design, where much of the work is moved to prior or later stages.

$$c^{mp}_{synchronization} = c_m + c_r \times (\#stages - 1) \tag{6.4}$$

The total synchronization costs in a message passing design is modeled in Equation 6.4. It consists of the one-time cost of creating the message c_m and the variable costs c_r of moving the packets between the different pipeline stages. We implemented this with the help of lock-free ring data structures and found, that batching improves performance greatly. E.g. the cost of enqueuing 64 packets instead of 1 is nearly the same, but the former amortizes c_r across them, leading to a much lower per packet overhead.

c_m is mainly dominated by the size of a message. At multiple million packets per second and equally many messages, the cost of simply copying bytes in memory becomes significant. Efficient usage of the CPU caches, in form of cache-line optimized data packing, is mandatory.

Chapter 7

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. We found, that a considerable amount overhead stems from the underlying Linux network stack, on which all tested versions rely on. It is too general, has to be compatible with exotic and legacy hardware and follows a packet buffer model, which maps poorly to the run-to-completion model in modern high-speed NICs.

Both IPsec and WireGuard[1] are directly integrated inside the Linux kernel. A common argument for implementing performance critical applications in kernel is superior performance due to closeness . Proponents argue, that placing such complex code inside ring 0 over-proportionally increases the attack surface and is thus undesirable. They propose a micro-kernel like approach, where tasks like VPNs are handled by confined user-space applications. Exploits there do not results in a compromise of the complete system.

The second implementation we evaluated, OpenVPN, is such a user-space application. Nonetheless we found, that the overhead of making syscalls and the single-threaded nature make OpenVPN the slowest of the valuated versions. Its peak rates are around 0.16 Mpps for 64 byte packets and 0.10 Mpps for 1514 byte, which translates to only 80 Mbit/s to 1 Gbit/s data rate. As long as the kernel does not provide a high-performance

[1] At the time of writing, the migration from module into mainline was nearing its completion

interface to the NICs, fast user-space applications on top of the existing APIs remain unlikely.

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 a order of magnitude over OpenVPN with a peak rate of 1.3 Mpps compared to the 0.16 Mpps. 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 manual encrypt-then-mac ciphers like AES-CBC-HMAC-SHA-256 by performing the encryption and authentication in a single pass over the data, instead of two. From our benchmarks this can reduce the required cycles per byte from 19 to under 5 on the same test hardware and large packets. Data synchronization proved to be the third major area where performance is decided. Multi-threaded applications require efficient data organization or else the speedup from having multiple cores will be negated by the synchronization overhead. We observed very poor scaling of lock-base-synchronization in both Linux IPsec and WireGuard, where up to 90% of the time is spend 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. Instead of synchronizing frequent accesses to hot data from all threads, only one dedicated worker in the pipeline This requires optimization of the serial code paths and strict distribution of the parallel sections, else this single worker will become a bottleneck. Batching operations over multiple packets at a time proved to be a necessity. 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.

We describe these three performance categories; driver, encryption and data synchronization, with our performance model and give equations to estimate a systems processing rate.

7.1 Future Work

Future work could include measurements of the impact of the patches mitigating the CPU bugs of Meltdown, Spectre et al. The counter measures add additional overhead to system calls, context switches and page faults, which especially concerns user-space applications triggering these often. DPDK should not be affected, as calls withing this framework do not pass kernel boundaries.

Another area for future research is the decryption side of the VPN operation. While the core components of our model also apply there, some details of packet handling are different and are worth investigating, as the complete site-to-site setup also includes this step.

List of Figures

List of Tables

List of Acronyms

IoT	VPN.
ISO	International Organization for Standardization.
MAC	Medium access control.
Mpps	Million packtes per second.
MSS	Maximum segment size.
MTU	Maximum transmission unit.
OSI	Open Systems Interconnection. Reference model for layered network architectures by the OSI.
PDU	Protocol data unit. Refers to a message at a specific layer of the OSI model including all headers and trailers of the respective layer and all layers above.
RSS	Receive-side Scaling. Technique to distribute packets to NIC queues based on a hash of the packet header fields.
SCTP	SCTP. Datagram-oriented, semi-reliable transport layer protocol.
SDU	Service data unit. Refers to the payload of a message at a specific layer of the OSI model excluding all headers and trailers of the respective layer.
TCP	Transmission control protocol. Stream-oriented, reliable, transport layer protocol.
UDP	User datagram protocol. Datagram-oriented, unreliable transport layer protocol.
VPN	Virtual Private Network. Extended, secured private network of a public network.

Bibliography

[1] Samy Al Bahra. *Concurrency primitives, safe memory reclamation mechanisms and non-blocking data structures for the research, design and implementation of high performance concurrent systems.* URL: `http://concurrencykit.org/` (visited on 11/30/2018).

[2] Daniel J. Bernstein and Tanja Lange. *eBACS: ECRYPT Benchmarking of Cryptographic Systems.* URL: `https://bench.cr.yp.to/results-stream.html` (visited on 12/12/2018).

[3] Raffaele Bolla and Roberto Bruschi. "Linux software router: Data plane optimization and performance evaluation". In: *Journal of Networks* 2.3 (2007), pp. 6–17.

[4] Tzi-cker Chiueh and Prashant Pradhan. "High-performance IP routing table lookup using CPU caching". In: *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE.* Vol. 3. IEEE. 1999, pp. 1421–1428.

[5] Frank Denis. *Libsodium: A modern, portable, easy to use crypto library.* URL: `https://libsodium.org` (visited on 10/16/2018).

[6] Jason Donenfeld and Kevin Milner. *Formal Verification of the Wireguard protocol.* 2017.

[7] Jason A Donenfeld. *WireGuard device xmit function.* URL: `https://git.zx2c4.com/WireGuard/tree/src/device.c#n125` (visited on 10/30/2018).

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

[9] Jason A Donenfeld. *WireGuard peer struct.* URL: `https://git.zx2c4.com/WireGuard/tree/src/peer.h#n37` (visited on 10/30/2018).

[10] Jason A Donenfeld. *WireGuard peer struct.* URL: `https://git.zx2c4.com/WireGuard/tree/src/noise.c#n29` (visited on 10/30/2018).

[11] Jason A Donenfeld. *WireGuard pointer queue definition.* URL: `https://git.zx2c4.com/WireGuard/tree/src/compat/ptr_ring/include/linux/ptr_ring.h?h=cbd4e34&id=cbd4e34#n34` (visited on 11/20/2018).

[12] Jason A Donenfeld. *WireGuard `send_staged_packets` function.* URL: `https://git.zx2c4.com/WireGuard/tree/src/send.c?h=cbd4e34&id=cbd4e34#n285` (visited on 11/20/2018).

[13] Eric Dumazet. *net: add additional lock to qdisc to increase throughput.* URL: `https://github.com/torvalds/linux/commit/79640a4` (visited on 12/12/2018).

[14] Morris J Dworkin. *Recommendation for block cipher modes of operation: Galois/-Counter Mode (GCM) and GMAC.* Tech. rep. 2007.

[15] Paul Emmerich. *libmoon is a library for fast and flexible packet processing with DPDK and LuaJIT.* URL: `https://github.com/libmoon/libmoon/` (visited on 12/12/2018).

[16] Paul Emmerich et al. "Mind the Gap – A Comparison of Software Packet Generators". In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2017).* Beijing, China, May 2017.

[17] Paul Emmerich et al. "MoonGen: A Scriptable High-Speed Packet Generator". In: *Internet Measurement Conference (IMC) 2015, IRTF Applied Networking Research Prize 2017.* Tokyo, Japan, Oct. 2015.

[18] Paul Emmerich et al. "Moongen: A scriptable high-speed packet generator". In: *Proceedings of the 2015 Internet Measurement Conference.* ACM. 2015, pp. 275–287.

[19] Jason Evans. "A scalable concurrent malloc (3) implementation for FreeBSD". In: *Proc. of the bsdcan conference, ottawa, canada.* 2006.

[20] OpenSSL Software Foundation. *Manpage of the OPENSSL_ia32cap capabilities vector.* URL: `https://github.com/OpenVPN/openvpn/blob/release/2.4/src/openvpn/crypto.c#L287-L294` (visited on 10/01/2018).

[21] The OWASP Foundation. *TLS Cipher String Cheat Sheet.* URL: `https://www.owasp.org/index.php/TLS_Cipher_String_Cheat_Sheet` (visited on 12/12/2018).

[22] Sebastian Gallenmüller et al. "High-Performance Packet Processing and Measurements (Invited Paper)". In: *10th International Conference on Communication Systems & Networks (COMSNETS 2018).* Bangalore, India, Jan. 2018.

[23] José Luis García-Dorado et al. "High-performance network traffic processing systems using commodity hardware". In: *Data traffic monitoring and analysis.* Springer, 2013, pp. 3–27.

[24] Thomas Gschwantner. *WireGuard GSoC 2018 Report.* URL: `https://lists.zx2c4.com/pipermail/wireguard/2018-August/003274.html` (visited on 12/12/2018).

[25] Pankaj Gupta, Steven Lin, and Nick McKeown. "Routing lookups in hardware at memory access speeds". In: *INFOCOM'98. Seventeenth Annual Joint Conference*

of the IEEE Computer and Communications Societies. Proceedings. IEEE. Vol. 3. IEEE. 1998, pp. 1240–1247.

[26] Adrian Hoban. “Using intel aes new instructions and pclmulqdq to significantly improve ipsec performance on linux”. In: *Intel Corporation* (2010), pp. 1–26.

[27] Berry Hoekstra, Damir Musulin, and Jan Just Keijser. “Comparing TCP performance of tunneled and non-tunneled traffic using OpenVPN”. In: *Universiteit Van Amsterdam, System & Network Engineering, Amsterdam* (2011), pp. 2010–2011.

[28] “IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications”. In: *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)* (2016), pp. 1–3534. DOI: `10.1109/IEEESTD.2016.7786995`.

[29] OpenVPN Inc. *OpenVPN*. URL: `https://openvpn.net/` (visited on 12/10/2018).

[30] OpenVPN Inc. *OpenVPN encrypt function*. URL: `https://github.com/OpenVPN/openvpn/blob/release/2.4/src/openvpn/crypto.c#L287-L294` (visited on 10/01/2018).

[31] I Kotuliak, P Rybár, and P Trúchly. “Performance comparison of IPsec and TLS based VPN technologies”. In: *Emerging eLearning Technologies and Applications (ICETA), 2011 9th International Conference on.* IEEE. 2011, pp. 217–221.

[32] Sameer G. Kulkarni et al. “NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication.* SIGCOMM ’17. Los Angeles, CA, USA: ACM, 2017, pp. 71–84. ISBN: 978-1-4503-4653-5. DOI: `10.1145/3098822.3098828`. URL: `http://doi.acm.org/10.1145/3098822.3098828`.

[33] Lawrence Berkeley National Laboratory. *iPerf 3 user documentation*. URL: `https://iperf.fr/iperf-doc.php#3doc` (visited on 12/12/2018).

[34] 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.* IEEE. 2017, pp. 466–471.

[35] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. “This POODLE bites: exploiting the SSL 3.0 fallback”. In: *Security Advisory* (2014).

[36] Jonathan Neuschäfer. *My GSoC contributions to WireGuard*. URL: `https://lists.zx2c4.com/pipermail/wireguard/2018-August/003240.html` (visited on 12/12/2018).

[37] Y. Nir et al. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 8439. RFC Editor, 2018. URL: `https://tools.ietf.org/html/rfc8439`.

[38] Michele Paolino et al. "SnabbSwitch user space virtual switch benchmark and performance optimization for NFV". In: *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE. 2015, pp. 86–92.

[39] Trevor Perrin. *The Noise Protocol Framework*. URL: `https://noiseprotocol.org/noise.html` (visited on 10/30/2018).

[40] DPDK Project. *DPDK Programmer's Guide: Malloc*. URL: `http://doc.dpdk.org/guides-17.11/prog_guide/env_abstraction_layer.html#malloc` (visited on 12/10/2018).

[41] DPDK Project. *DPDK Programmer's Guide: Thread Safety of DPDK Functions*. URL: `http://doc.dpdk.org/guides-17.11/prog_guide/thread_safety_dpdk_functions.html#fast-path-apis` (visited on 10/11/2018).

[42] DPDK Project. *DPDK rte_lpm API*. URL: `http://doc.dpdk.org/api-17.11/rte__lpm_8h.html`.

[43] DPDK Project. *DPDK rte_ring enqueue function*. URL: `http://git.dpdk.org/dpdk/tree/lib/librte_ring/rte_ring_generic.h#n96` (visited on 10/11/2018).

[44] The Open MPI Project. *Open MPI: Open Source High Performance Computing*. URL: `https://www.open-mpi.org/` (visited on 12/02/2018).

[45] Maximilian Pudelko. *MoonWire GitHub Repository*. URL: `https://github.com/pudelkoM/MoonWire` (visited on 10/30/2018).

[46] Maximilian Pudelko. *Patched Stand-alone WireGuard*. URL: `https://github.com/pudelkoM/WireGuard/tree/798d23` (visited on 11/20/2018).

[47] Daniel Raumer et al. "Efficient serving of VPN endpoints on COTS server hardware". In: *Cloud Networking (Cloudnet), 2016 5th IEEE International Conference on*. IEEE. 2016, pp. 164–169.

[48] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC Editor, 2018. URL: `https://tools.ietf.org/html/rfc8446`.

[49] Hugo Sadok, Miguel Elias M Campista, and Luís Henrique MK Costa. "A Case for Spraying Packets in Software Middleboxes". In: *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. ACM. 2018, pp. 127–133.

[50] Mahmoud Sayrafiezadeh. "The birthday problem revisited". In: *Mathematics Magazine* 67.3 (1994), pp. 220–223.

[51] Haoyu Song et al. "Ipv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards". In: *INFOCOM 2009, IEEE*. IEEE. 2009, pp. 2518–2526.

[52] Linus Torvalds. *Definition of xfrm struct*. URL: `https://elixir.bootlin.com/linux/v4.4.109/source/include/net/xfrm.h#L128` (visited on 10/01/2018).

[53] Linus Torvalds. *xfrm output function.* URL: `https://elixir.bootlin.com/linux/v4.4.109/source/net/xfrm/xfrm_output.c#L53` (visited on 10/01/2018).

[54] Linus Torvalds. *xfrm replay function.* URL: `https://elixir.bootlin.com/linux/v4.4.109/source/net/xfrm/xfrm_replay.c#L95` (visited on 10/01/2018).

[55] Linux Torvalds. *Spinlock in qdisc code path.* URL: `https://elixir.bootlin.com/linux/v4.4.109/source/net/core/dev.c\#L2939` (visited on 12/12/2018).

[56] Steve Vinoski. "Concurrency and message passing in erlang". In: *Computing in Science & Engineering* 14.6 (2012), pp. 24–34.

[57] Peter Zijlstra. *preempt: Take away preempt_enable_no_resched() from modules.* URL: `https://lore.kernel.org/patchwork/patch/420889/` (visited on 11/30/2018).