if_ovpn or OpenVPN

BY KRISTOF PROVOST

Today[1], you're going to be reading[2] about OpenVPN's DCO.

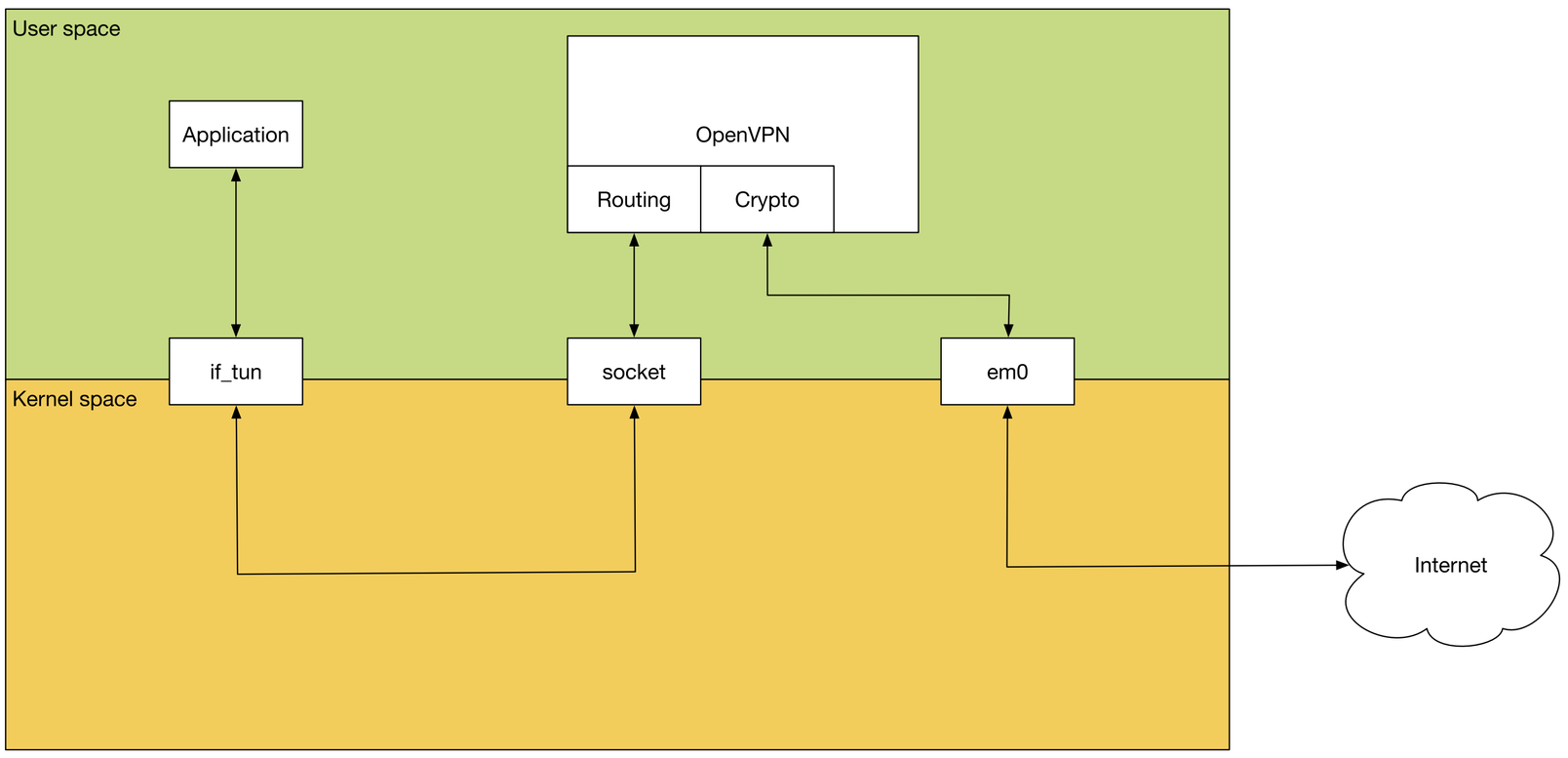
Initially developed by James Yonan, OpenVPN saw its first release on May 13, 2001. It supports many common platforms (such as FreeBSD, OpenBSD, Dragonfly, AIX, ...) and a few less common ones (macOS, Linux, Windows) as well. It supports peer-to-peer and client-server models, with pre-shared key, certificate, or username/password-based authentication.

As you'd expect with any project that's been around for more than 20 years, it grew many features for many different use cases.

The Problem

While OpenVPN is very nice, clearly there must be a problem. Without a problem, this article wouldn't be very interesting[3]. There is, indeed, an issue, and it is that OpenVPN is implemented as a single-threaded, userspace process.

It uses if_tun to inject packets into the network stack. As a result, its performance has not kept up with current connectivity rates. It also makes it difficult to take advantage of modern multi-core hardware or cryptographic offload hardware.



The main issue with OpenVPN's performance is its userspace nature. Incoming traffic is naturally received by a NIC, which would typically DMA the packet into kernel memory. It is then processed further by the network stack until that works out what socket the packet belongs to, and passes it to userspace. This socket may be UDP or TCP.

Passing the packet to userspace involves copying it, at which point the userspace OpenVPN process verifies and decrypts the packet and re-injects it into the network stack using if_tun. This means copying the plain-text packet back into the kernel for further processing.

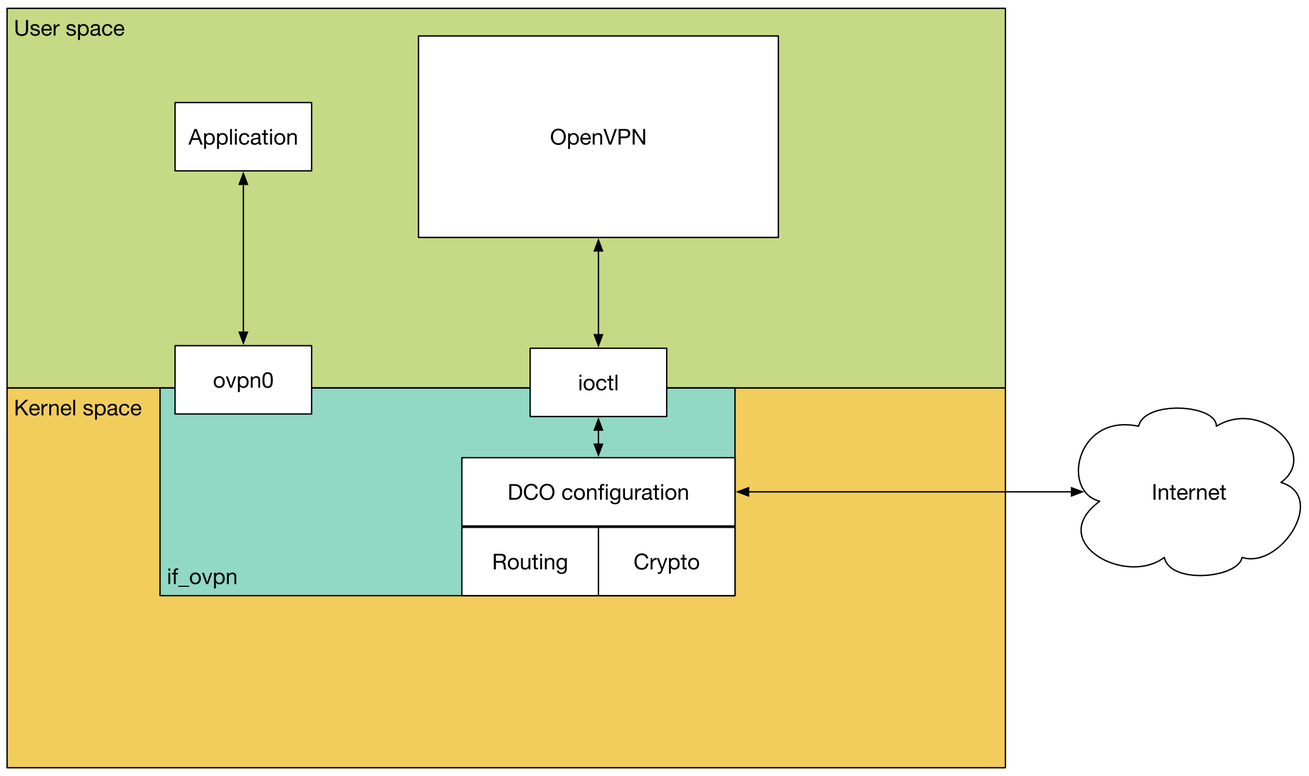
Inevitably all this context switching and copying back and forth has a significant impact on performance.

In the current architecture it's very hard to make significant performance improvements.

What is DCO

Now that we've established what our problem is, we can start thinking about solutions[4].

If our problem is context switches to userspace then one plausible solution is to keep the work inside the kernel, and that's what DCO–Data Channel Offload–does.



DCO moves the data channel, that is, the cryptographic operations and the tunneling of the traffic, into the kernel. It does this through a new virtual device driver, if_ovpn. The OpenVPN userspace process remains responsible for connection setup (including authentication and option negotiation), coordinating with the if_ovpn driver via a new ioctl interface.

The OpenVPN project decided that the introduction of DCO was a good opportunity to remove some legacy features and do some general tidying up. As part of that, they've taken the Henry Ford approach to encryption algorithm choice. You can have any algorithm you like, as long as you like AES-GCM or ChaCha20/Poly1305. In black.

DCO also does not support compression, layer 2 traffic, non-subnet topologies or traffic shaping[5].

It's important to note here that DCO does not change the OpenVPN protocol. It's possible for a client to use it with a server that does not, or vice versa. You'll get the biggest benefit when both sides use it, of course, but that's not required.

Considerations

This is the part where I talk up just how hard this all was, so you'll all be impressed that I actually got this to work. Does that still work if I tell you that's what I'm doing? Let's find out!

Anyway, there are a couple of things that needed special attention:

Multiplexing

The first issue is that OpenVPN uses a single connection to transport both the tunneled data and the control data. The tunneled data needs to be handled by the kernel, and the control data is handled by the OpenVPN userspace process.

You can see the issue. The socket is initially opened and fully owned by OpenVPN itself. It sets up the tunnel and handles the authentication. Once that's completed, it partially hands over control to the kernel side (i.e., if_ovpn).

This means informing if_ovpn of the file descriptor (which the kernel uses to look up the in-kernel struct socket), so it can hold a reference to it. That ensures the socket doesn't go away while the kernel is using it. Perhaps because the OpenVPN process was terminated, or because it was having a bad day and decided to mess with us. It's userspace, it does crazy things.

For those of you who want to follow along in the kernel code, you're looking for the `ovpn_new_peer()`[6] function.

Having looked up the socket we can now also install the filtering function via `udp_set_kernel_tunneling()`. The filter, `ovpn_udp_input()`, looks at all incoming packets for the specified socket, and decides if it's a payload packet which it should handle, or a control packet which OpenVPN in userspace should handle.

This tunneling function is also the only change I had to make to the rest of the network stack. It needed to be taught that certain packets would be handled by the kernel and others could still be passed through to userspace. That was done in https://cgit.freebsd.org/src/commit/?id=742e7210d00b359d81b9c778ab520003704e9b6c .

The `ovpn_udp_input()` function is the main entry point for the receive path. The network stack hands packets over to this function for any UDP packets arriving on the socket it's been installed on.

The function first checks if the packet can be handled by the kernel driver. That is, the packet is a data packet and it's destined for a known peer id. If that's not the case the filter function tells the UDP code to pass the packet through the normal flow as if there were no filter function. That means the packet will arrive on the socket and be processed by OpenVPN's userspace process.

> Pretty much every modern CPU has multiple cores, and it'd be kind of nice to be able to use more than just one of them.

Early versions of the DCO driver had separate ioctl commands to read and write control messages, but both the Linux and FreeBSD drivers have been adapted to use the socket instead. This simplifies handling of both control packets and new clients.

If, on the other hand, the packet is a data packet for a known peer, it is decrypted, has its signature validated, and is then passed on to the network stack for further processing.

For those of you following along, that's done here https://cgit.freebsd.org/src/tree/sys/net/if_ovpn.c?id=da69782bf06645f38852a8b23af#n1483 .

UDP

OpenVPN can be run over both UDP and TCP. While UDP is the obvious choice for a layer 3 VPN protocol, some users need to run it over TCP to transit firewalls.

The FreeBSD kernel offers a convenient filter function for UDP sockets, but has no equivalent for TCP, so FreeBSD if_ovpn currently only supports UDP and not TCP.

The Linux DCO driver developer was rather more ... courageous and has chosen to implement TCP support as well. The developer did, against the odds, in fact survive this experience, and is now significantly wiser.

Hardware Cryptography Offload

if_ovpn relies on the in-kernel OpenCrypto framework for cryptographic operations. This means it can also take advantage of any cryptographic offload hardware present in the system. This can further improve performance.

It's already been tested with Intel's QuickAssist Technology (QAT), the SafeXcel EIP-97 crypto accelerator and AES-NI.

Locking Design

Look, if you thought you were going to get a discussion of kernel code without having to talk about locking, I don't know what to tell you. That was naively optimistic of you.

Pretty much every modern CPU has multiple cores, and it'd be kind of nice to be able to use more than just one of them. That is, we can't just lock out other cores while one core is doing work. It's impolite. It also doesn't perform well.

Happily, this turned out to be reasonably easy to do. The entire approach is based on distinguishing read and write accesses to if_ovpn's internal data structures. That is, we allow many different cores to look up things at the same time but will only ever allow one to change things (and then not allow any readers while the change is being made). That turns out to work well enough because—most of the time—we don't need to change things.

Each OpenVPN DCO platform has its own unique way of communicating between userspace OpenVPN and the kernel module.

The common case, when we receive or send packets, just needs to look up keys, destination addresses and ports and other related information.

It's only when we modify things (i.e., on configuration changes or re-keying) that we need to take a write lock, and that we pause the data channel. That's brief enough that our puny human brains won't notice it, and that makes everyone happy.

There's one exception to this "we don't make changes to process data" rule, and that is packet counters. Every packet gets counted (twice even, once for the packet count, once for a byte count), and that has to be done concurrently. Here, too, we are lucky, in that the kernel's `counter(9)` framework is designed exactly for this situation. It keeps totals per CPU core so that one core will not affect or slow down another. It's only when the counters are actually read that it will ask each core for its total and will add them up.

Control Interface

Each OpenVPN DCO platform has its own unique way of communicating between userspace OpenVPN and the kernel module.

On Linux, this is done through netlink, but the if_ovpn work was completed before FreeBSD's netlink implementation was ready. As I'm still on probation for my last causality violation, I decided to use something else instead.

The if_ovpn driver is configured through the existing interface ioctl path. Specifically, the `SIOCSDRVSPEC`/`SIOCGDRVSPEC` calls.

These calls pass a **`struct ifdrv`** to the kernel. The **`ifd_cmd`** field is used to pass the command, and the **`ifd_data`** and **`ifd_len`** fields are used to pass device-specific structs between kernel and userspace.

if_ovpn deviates somewhat from the established approach, in that it transmits serialized nvlists rather than structs. This makes extending the interface easier. Or, rather, it means we can extend the interface without breaking existing userspace consumers. If a new field is added to a struct, its layout changes which either means that the existing code will refuse to accept it due to its size mismatch[7] or get very confused because fields no longer mean what they used to mean.

Serialized nvlists allow us to add fields without confusing the other side. Any unknown fields will just be ignored. This makes adding new features much easier.

Routing Lookups

You might think that if_ovpn wouldn't need to worry about routing decisions. After all, the kernel's network stack has already made the routing decision by the time the packet arrives at the network driver. You'd be wrong. I'd make fun of you for that, but it took me a while to figure it out, too.

The issue is that there are potentially multiple peers on a given if_ovpn interface (e.g., when it's acting as a server and has multiple clients). The kernel has figured out that the packet in question needs to go to one of them, but the kernel operates on the assumption that all these clients live on a single broadcast domain. That is, a packet sent on the interface would be visible to all of them. That's not the case here, so if_ovpn needs to work out which client the packet has to go to.

OpenVPN will install the new key using the OVPN_NEW_KEY command.

This is handled by **`ovpn_route_peer()`**. This function first looks through the list of peers to see if any peer's VPN address matches the destination address. (Done by **`ovpn_find_peer_by_ip()`** or **`ovpn_find_peer_by_ip6()`**, depending on address family). If a matching peer is found the packet is sent to this peer.

If not **`ovpn_route_peer()`** performs a route lookup, and repeats the peer lookup with the resulting gateway address.

Only when if_ovpn has figured out the peer to send the packet to can it be encrypted and transmitted.

Key Rotation

OpenVPN will from time to time change the key used to secure the tunnel. That's one of those hard jobs if_ovpn leaves to userspace, so some coordination between OpenVPN and if_ovpn is required.

OpenVPN will install the new key using the OVPN_NEW_KEY command. Each key has an ID, and every packet includes the key ID that was used to encrypt it. This means that during key rotation, all packets can still be decrypted, as both the old and new keys are known and kept active in the kernel.

Once the new key is installed, it can be made active using the OVPN_SWAP_KEYS command. That is, the new key will be used to encrypt outgoing packets.

Sometime later the old key can be deleted using the OVPN_DEL_KEY command.

vnet

Yes, we're going to have to talk about vnet. I'm writing this, it's inevitable.

I'm too lazy to explain it entirely, so I'll just point you at an article written by much better author Olivier Cochard-Labbé: "Jail: vnet by examples"[8].

Think of vnet as turning jails into virtual machines with their own IP stacks.

This isn't strictly required for the pfSense use case, but it makes testing much, much easier. It means we can test on a single machine, without needing any external tools (other than OpenVPN itself, for what should be pretty obvious reasons).

For those interested in how this is done there's another FreeBSD Journal article that might be useful: "The Automated Testing Framework," by ... wait, I think I know that guy, Kristof Provost.

Performance

After all of that, I bet you're asking yourself "Does this actually help though?"

Well, fortunately for me: yes, yes it does.

One of my colleagues at Netgate spent some time gently teasing a Netgate 4100[10] device with iperf3 and got these results:

`if_tun`	`207.3 Mbit/s`
`DCO Software`	`213.1 Mbit/s`
`DCO AES-NI`	`751.2 Mbit/s`
`DCO QAT`	`1,064.8 Mbit/s`

"if_tun" is the old OpenVPN approach without DCO. It's worth noting that it used AES-NI instructions in userspace, and the 'DCO software' setup did not. Despite this blatant attempt at cheating, DCO was still slightly faster. On a level playing field (i.e., where DCO does use AES-NI instructions) there's no contest. DCO is more than three times faster.

There's some good news for Intel too: their QuickAssist offload engine is even faster than AES-NI, making OpenVPN five times faster than it was previously.

Future Work

Nothing is so good that it cannot be improved, but in some ways this next enhancement is a result of the success of DCO's design.

The on-wire OpenVPN protocol uses a 32 bit initialization vector (IV), and for cryptographic reasons I won't explain here[11], it's a bad idea to re-use IVs with the same key.

That means that keys must be re-negotiated before we get to that point. OpenVPN's default renegotiation interval is 3600 seconds, and with a 30% margin for safety, that would translate to 2^32 * 0.7 / 3600, or about 835.000 packets per second. That's "only" 8 to 9 Gbit/s (assuming 1300 byte packets).

With DCO, that's already more or less within reach of contemporary hardware.

While it's a good problem to have, it's still a problem, so the OpenVPN developers are working on an updated packet format that will use 64-bit IVs.

Thanks

The if_ovpn work was sponsored by Rubicon Communications (trading as Netgate) for use with their pfSense product line. It's been in use there since the 22.05 pfSense plus release[12]. This work was upstreamed to FreeBSD and is part of the recent 14.0 release. It requires OpenVPN 2.6.0 or newer to use.

I'd also like to thank the OpenVPN developers, who were very welcoming when the initial FreeBSD patches turned up, and without whose assistance this project would not have gone anywhere near as well as it did.

Footnotes:

1. Or. whenever you read this.
2. Fine. Writing. Reading. Look, if you're going to be pedantic about this, we'll be at this all day.
3. Look, if you're not interested in DCO you can just go read the next article. I'm sure it's very nice.
4. I say "we", but as much as I'd like to take credit for the solution it was the OpenVPN developers who came up with the DCO architecture and implemented it for Windows and Linux. All I did was what they did, but for FreeBSD.
5. In OpenVPN. DCO can be combined with the OS's traffic shaping (i.e. dummynet).
6. https://cgit.freebsd.org/src/tree/sys/net/if_ovpn.c?id=da69782bf-06645f38852a8b23af#n490
7. You might also say because the struct got fat. You might, I'm too polite for that.
8. https://freebsdfoundation.org/wp-content/uploads/2020/03/Jail-vnet-by-Examples.pdf
9. https://freebsdfoundation.org/wp-content/uploads/2019/05/The-Automated-Testing-Framework.pdf
10. https://shop.netgate.com/products/4100-base-pfsense
11. Mostly because I do not understand them myself.
12. https://www.netgate.com/blog/pfsense-plus-software-version-22.05-now-available

---

KRISTOF PROVOST is a freelance, embedded software engineer specializing in network and video applications. He's a FreeBSD committer, maintainer of the pf firewall in FreeBSD. He currently spends most of his time working on pfSense for Netgate.

Kristof has an unfortunate tendency to stumble into uClibc bugs, and a burning hatred for FTP. Do not talk to him about IPv6 fragmentation.