if_ovpn, OpenVPN Data Channel Offload

Kristof Provost kp@FreeBSD.org

Abstract

if_ovpn is the FreeBSD implementation of OpenVPN's Data Channel Offload (DCO) technology. It optimises the OpenVPN data flow for increased VPN performance.

This work was paid for by Netgate, and can be found in Netgate's pfSense plus products.

In this paper we discuss the advantages and disadvantages of DCO as well as some of the if_ovpn design choices.

Background

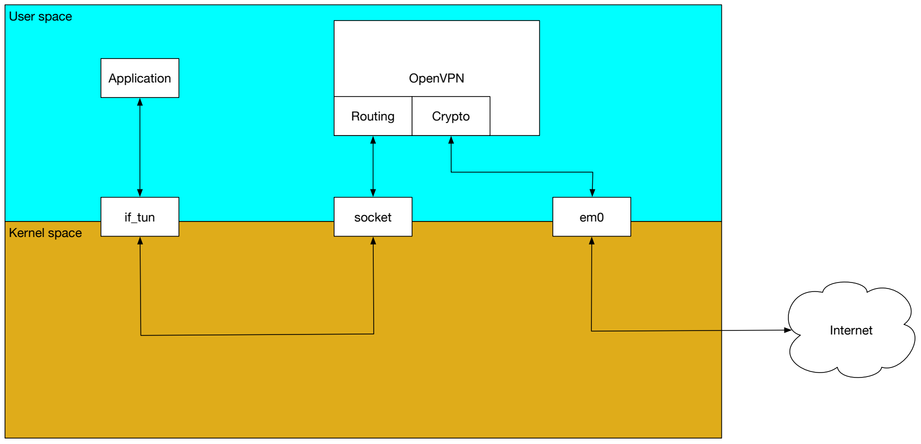
OpenVPN is a common VPN application, used for peer-to-peer or client/server mode connections. It supports pre-shared key, certificate or username/password based authentication. It supports most common platforms (Linux, Windows, macOS, Android, AIX, FreeBSD, OpenBSD, DragonflyBSD, . . .)

It was originally developed by James Yonan, about 20 years ago. Its first release was on May 13th, 2001.

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[1].

The problem

OpenVPN is fully implemented in user space as a single threaded 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 user space 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 user space. This socket may be UDP or TCP.

Passing the packet to user space 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.

Depending on the setup this may mean working out which socket will receive it and copying it to the correct user space process, or sending it onwards through a NIC.

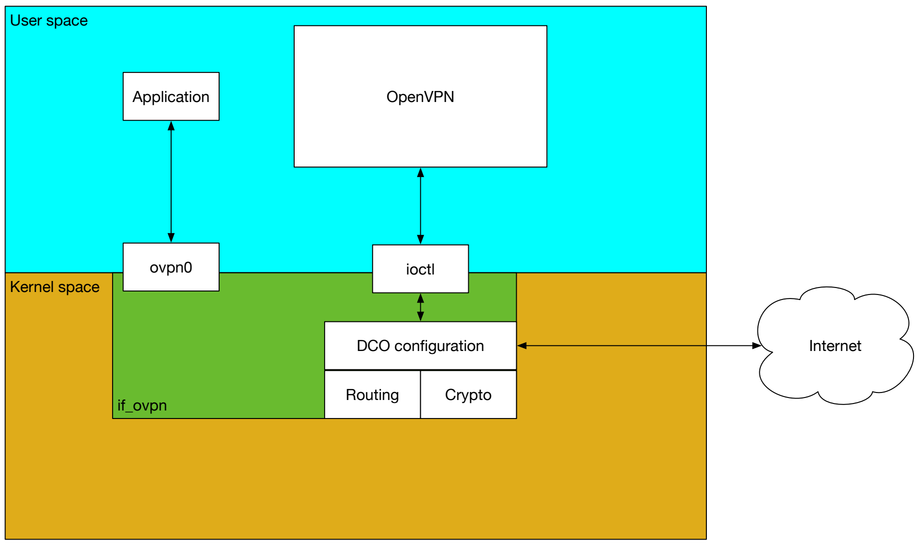
It is inevitable that both the context switching and copying involved in this have a negative effect on throughput.

Given the current architecture it's not possible to make significant improvements in performance.

What is DCO

DCO, or Data Channel Offload, is a new approach for the OpenVPN data path, where the cryptography is handled directly in the kernel, removing the need for expensive context switches and copies to and from user space.

DCO is available from OpenVPN 2.6.0's release[4] (January 26th, 2023) and is supported on Linux, FreeBSD and Windows. Linux and FreeBSD support both server and client modes, Windows supports only client mode.



This moves the data channel down into the kernel, removing the need for copies to and from user space.

It introduces a new virtual device driver, if_ovpn, to handle the cryptography and OpenVPN data path logic. The configuration of if_ovpn is handled through a new ioctl interface.

The OpenVPN project decided to use the introduction of DCO as an opportunity to remove legacy features. As part of that they've chosen to support only AES-GCM and ChaCha20/Poly1305 ciphers.

Considerations

A few historical design choices make implementing DCO challenging.

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.

Multiplexing

OpenVPN multiplexes two different streams across a connection to the remote endpoint:

- control channel Used to exchange authentication, negotiate parameters and exchange keys
- data channel User data traffic

Both channels share a single UDP connection / socket. This is initially opened by the user space OpenVPN process. Once initial authentication and handshaking is done control is - partially - handed over to the kernel driver.

The kernel will be responsible for handling the data path, but all of the complexity of authentication and the control path is deliberately left in user space.

This means that we must ensure that the socket remains open and valid as long as either the if_ovpn driver or the OpenVPN user space process continue to use it.

To allow for this OpenVPN passes the UDP socket file descriptor to the if_ovpn driver when it adds a new peer.

This is handled by the `ovpn_new_peer()`[3] function, which retrieves the `struct socket *` based on the file descriptor. It then holds a reference to it using `soref()` to ensure that the socket remains active even if the user space process goes away.

This socket is also used to install the filtering function, `ovpn_udp_input()`, using `udp_set_kernel_tunneling()`.

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 user space process.

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.

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

It’s quite common for modern systems, even relatively small and low-power embedded systems, to have multiple CPU cores.

To obtain the best possible performance from such systems it’s important to ensure that multiple cores can process OpenVPN packets at the same time. That translates into a need to consider the if_ovpn driver’s locking strategy with some care.

if_ovpn uses an rmlock to protect its internal data structures. The most important ones here are the `struct ovpn_softc`, which contains a list of `struct ovpn_kpeer`. Each `struct ovpn_kpeer` represents an active peer, and contains the per-peer information, such as the remote IP address, the tunnel address(es) and the encryption keys.

Happily the requirements of OpenVPN DCO allow for a straightforward approach, where the data path (i.e. receiving and sending packets) requires only read-only access to this data. Only configuration changes, by their nature far less frequent than data path operations, require making changes to these structures.

That means that the structures can be protected by acquiring a read lock for the data path, which allows multiple cores to perform useful work simultaneously. Taking the write lock for re-configuration does stop packet processing while the lock is held.

There’s one exception to this general rule in if_ovpn, and that is packet/byte counters. Happily the kernel’s `counter(9)` framework solves this problem for us by keeping per-CPU counters. This also avoids the caching issues caused by having multiple cores read/modify/write the same memory location, which helps performance.

ioctl interface

The control interface between the if_ovpn driver and the user space daemon uses the existing interface ioctl path. Specifically the `SIOCSDRVSPEC`/`SIOCGDRVSPEC` calls.

Usually the `struct ifdrv`'s `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 user space.

if_ovpn deviates from the traditional method though. It still uses the `ifd_cmd` field to identify the operation, but it passes a serialised nvlist in the `ifd_data` field, using `ifd_len` to indicate the serialised data length.

Nvlists make extending the interface far easier, as fields may be added (either by kernel or user space) without breaking ABI guarantees. That is, if the kernel returns an extra field user space doesn't know about it is automatically ignored. This makes adding new features much easier.

Routing lookups

The usual routing principles ensure that packets destined for an OpenVPN tunnel end up being sent out the correct interface.

For servers, with multiple clients, there is an additional routing problem however. The OpenVPN tunnel is not a single broadcast domain. The driver still needs to work out which peer to send packets to.

To deal with this the if_ovpn driver performs a second routing lookup, based on the destination address of the tunneled packet. 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.

There is one special case: if there is only one peer no lookup is performed and all traffic is sent to that peer.

Key rotation

OpenVPN will from time to time change the key used to secure the tunnel. The key negotiation for this is handled by user space, just like during initial connection setup.

Once a new key is agreed it is installed in the kernel 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.

Some time later the old key can be deleted using the OVPN_DEL_KEY command.

vnet

vnet/vimage support was added to the if_ovpn driver primarily for testing. vnets make it much more straightforward to create automated tests. There was no immediate user need for this support, as pfSense does not use vnets. Moreover, if_tun doesn't usefully support vnet either, as it needs both a network interface and a device node to be useful.

Performance

The main objective of DCO is to increase performance. The following numbers were measured with iperf3, on a Netgate 4100 device[2], using AES-256-GCM.

The 4100 is powered by an Intel® Atom® C3338R with QAT, 2-core @ 1.8 GHz.

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

It is worth noting that the if_tun case used AES-NI instructions in userspace. The DCO software case performed cryptographic operations without AES-NI instructions.

References

[1] https://www.netgate.com/blog/pfsense-plus-software-version-22.05-now-available

[2] https://shop.netgate.com/products/4100-base-pfsense

[3] https://cgit.freebsd.org/src/tree/sys/net/if_ovpn.c?id=da69782bf06645f38852a8b23afc965fc30d0e08#n452

[4] https://github.com/OpenVPN/openvpn/blob/v2.6.0/Changes.rst