Thermalcircle.de

climbing the thermals

linux, kernel, routing, netfilter, nftables, flowtables

Flowtables - Part 1: A Netfilter/Nftables Fastpath

In this article series I like to take a look at *flowtables*, which is a network fastpath mechanism in the Linux kernel, based on Netfilter/Nftables, that allows accelerated handling of forwarded TCP and UDP connections. When using an acceleration feature like this, it is important to understand how it works. If you don't, then you'll have a hard time once you are going beyond just plain forwarding and start combining that acceleration with other networking features like e.g. Firewalling, NAT, advanced routing, QoS or IPsec. In this first article I'll take a deep look at the packet flow. I'll show you how you can setup and use a *flowtable* and explain how that mechanism works internally.

Articles of the series

- Flowtables - Part 1: A Netfilter/Nftables fastpath
- Flowtables - Part 2: IPsec gateway in tunnel-mode

Overview

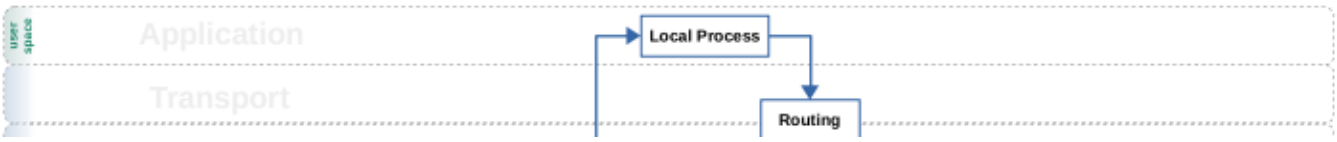
Flowtables is a network acceleration/fastpath feature within the Linux kernel, which has been created by the *Netfilter* developers. If packets belonging to a TCP or UDP connection are being forwarded (router) and are being tracked by the *Netfilter connection tracking* subsystem, then this tracked connection (also referred to as *flow*) can be *offloaded* to the *flowtables* fastpath. This makes packets of that *flow* bypass most of the steps of the classic forwarding path, which thereby accelerates packet handling. The default *flowtables* fastpath is implemented in software. However, in combination with specific hardware, it can additionally provide a hardware-based fastpath. The main focus of this article, however, is on the software fastpath. It represents the basics and principles how the *flowtable* mechanism works and should be understood first, before going deeper into the topic. Here some quick facts about *flowtables*:

- supports OSI layer 3 protocols IPv4 and IPv6 and OSI layer 4 protocols TCP and UDP
- is designed and able to to accelerate forwarded traffic (only!)
- works in use cases which involve NAT, Policy-based routing, virtual routing or QoS
- works in IPsec tunnel-mode use cases and also in GRE + IPsec transport-mode use cases
- since kernel 5.13, works in use cases which involve bridging, VLAN or PPPoE

I couldn't find documentation on most of those mentioned use cases which go beyond just plain forwarding. So I confirmed that those do work with *flowtables* by practical experimenting and reading source code. I'll describe more details in the sections below and the following article(s).

The Slowpath

Let's first take a look at packet flow on the classic forward path; thus, the path an IPv4[1)] packet takes through the kernel network stack when being forwarded – without using *flowtables*. I'll also refer to it as the *slowpath*. Figure 1 illustrates the packet flow. Of course, this Figure and my description below intentionally leave out a lot of details and focus merely on things which are relevant for the topic at hand.



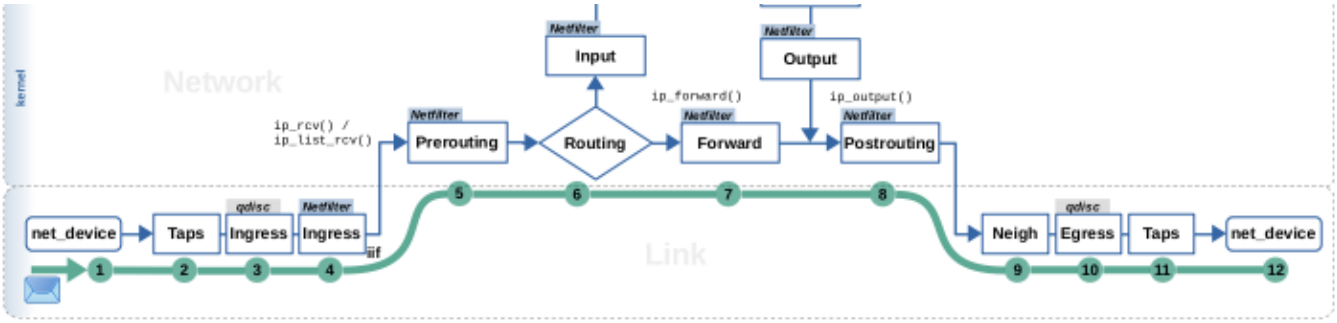


Figure 1: Classic forwarding path for IPv4 packets (click to enlarge).

Step (1): The packet is received on a network device.

Step (2): The packet traverses *taps* (where e.g. Wireshark/tcpdump could listen).

Step (3): The packet traverses the *ingress* queueing discipline of the network packet scheduler (tc).

Step (4): The packet traverses the Netfilter *Ingress* hook of the network device where the packet has been received (where e.g. a *flowtable* could be placed… more on that later).

Step (5): The packet traverses the Netfilter IPv4 *Prerouting* hook.

Step (6): The routing lookup is performed and determines that this packet shall be forwarded. The resulting *routing decision* is being attached to the packet as meta data; see Figure 2. It contains information like the outgoing network interface, whether the *nexthop* is a gateway or not, the *gateway* IP address (if existing), and function pointers, which determine the path the packet takes from now on through the network stack. My article series Routing Decisions in the Linux Kernel describes this in detail.

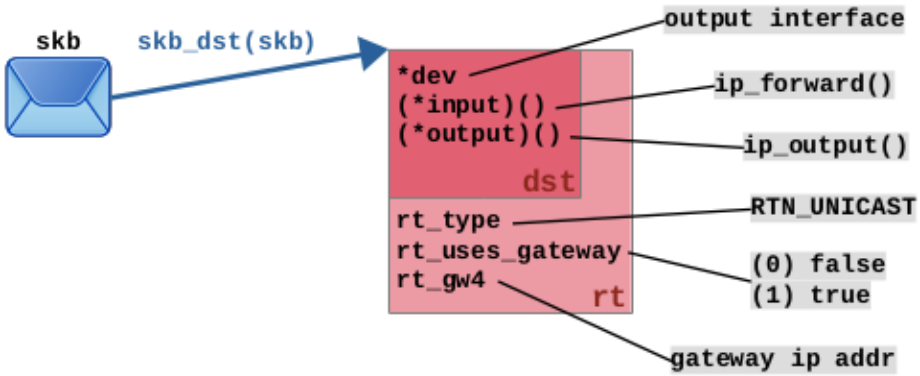


Figure 2: *Routing decision* attached to packet in step (6) (simplified); combination of `struct rtable [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/route.h#L49]` and `struct dst_entry [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L25]`. Path of network packet through remaining part of network stack is determined by function pointers `(*input)() [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L35]` and `(*output)() [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L36]`, which in this case put packet on forward path and then output path. See also The routing decision object.

Step (7): The *routing decision* puts the packet on the *forward* path. More precise, function pointer `(*input)() [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L35]` of the attached *routing decision* is called, which results in function `ip_forward() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_forward.c#L86]` being called. The packet now traverses the Netfilter IPv4 *Forward* hook.

Step (8): The *routing decision* makes sure the packet now continues its way on the *output* path. More precise, function pointer `(*output)() [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L36]` of the attached *routing decision* is called, which results in function `ip_output() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L423]` being called. The packet now traverses the Netfilter IPv4 *Postrouting* hook.

Step (9): The packet traverses the *neighboring subsystem* which does resolve the *nexthop* IP address into a MAC address, by doing

an ARP lookup, in case the address is not yet in the *neighbor cache*. The *nexthop* IP address is either the destination IP address of the packet or in case the *nexthop* is a gateway, it is this gateway's IP address. These details are carried by the attached *routing decision*.

Step (10): The packet traverses the *egress* queueing discipline of the network packet scheduler (tc).

Step (11): The packet traverses *taps* (where e.g. Wireshark/tcpdump could listen).

Step (12): The packet is sent out on a network device.

The Fastpath

Now let's see how the packet flow deviates for packets which travel on the *flowtables* fastpath. Figure 3 illustrates this. As you can see, the fastpath bypasses steps 5 till 8, which are the Netfilter *Prerouting* hook, the *routing lookup* and the Netfilter *Forward* and *Postrouting* hooks. This means, most of the normal slowpath forward packet handling is being bypassed.

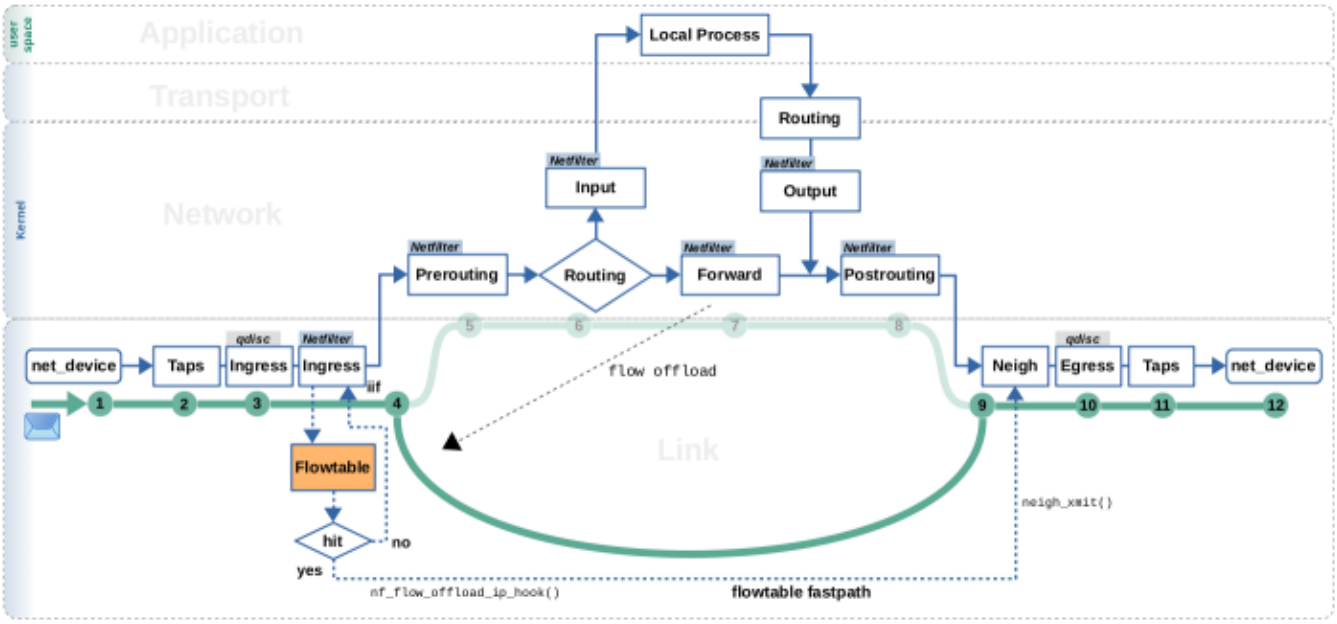


Figure 3: *Flowtable* fastpath for IPv4 packets (click to enlarge).

A *flowtable*, which is actually a hash table, is placed in the Netfilter *ingress* hook[2]. A lookup into this hash table is then performed for each IPv4/IPv6 packet traversing this hook (step 4), to decide which packets go on the fastpath and which stay on the slowpath. This lookup is based on packet characteristics like source and destination IP addresses, source and destination TCP/UDP ports, l3proto, l4proto and input interface, similar to how the connection tracking system operates. If a packet creates a hit/match, it travels on the fastpath, if not, it simply continues its way on the slowpath. In case of IPv4 packets, this hash table lookup and the fastpath packet handling are done by function `nf_flow_offload_ip_hook() [https://elixir.bootlin.com/linux/v5.14.7/source/net/netfilter/nf_flow_table_ip.c#L328]`. A *flowtable* in general works kind-of like a routing cache. It remembers the *routing decisions* (see Figure 2) for packets that belong to offloaded flows. These remembered *routing decisions* are then being attached to the packets which are travelling on the fastpath. Further, some more checks and minor things, like decrementing TTL,are done to these packets and then they are being re-inserted into the slowpath at the *neighboring subsystem* step (9). This is done by function `neigh_xmit() [https://elixir.bootlin.com/linux/v5.14.7/source/net/core/neighbour.c#L2991]`.

You can draw several important conclusions from that: You can clearly see, that the *ingress* and *egress* queueing disciplines[3] of the *network packet scheduler* (tc) are traversed by the packet, no matter if *flowtables* is used or not. Thereby, *flowtables* works well in combination with QoS. Further, you see that the *taps* are also traversed in all cases. Thus, *flowtables* does not hide your packets from tools like *Wireshark*. You also see, that packets travelling on the *flowtables* fastpath bypass the Netfilter *Prerouting*, *Forward* and *Postrouting* hooks. So, you must be aware that the *Nftables* chains and rules you potentially put there are not traversed by packets travelling on the fastpath.

How it works

The *flowtables* feature is built upon the *Netfilter* hooks, *Nftables*, and the *Connection Tracking* system. A thorough understanding of these building blocks is required to fully understand the following sections. If you feel unsure with these, please check out my article Nftables - Packet flow and Netfilter hooks in detail and my article series on Connection Tracking[4]. Figure 4 shows a simple example topology which shall serve as basis for the explanations below. It shows traffic between a client and a server, forwarded on a Linux router. Within that router, it shows the Netfilter hooks which are traversed in this case and the registered callbacks within these hooks, which are relevant for *flowtables*.

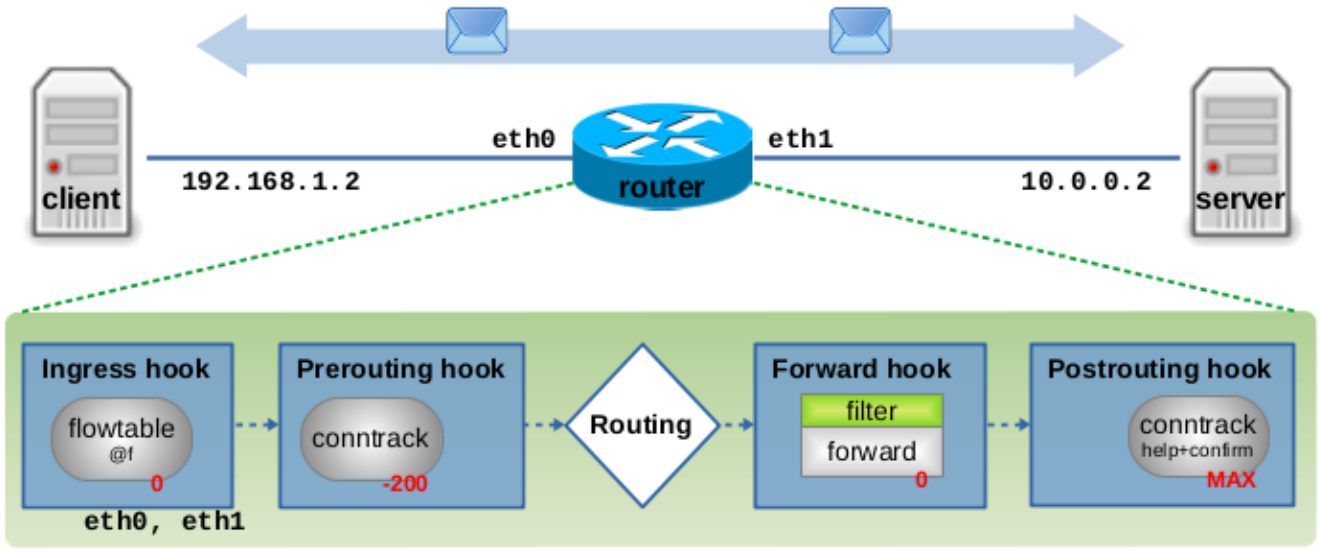


Figure 4: Netfilter hooks and callbacks traversed on a Linux router, relevant for *flowtables*.

Setting up a flowtable

To make use of the *flowtables* fastpath, you first need to create an instance of a flowtable. This must be done within an Nftables table of one of the address families `ip`, `ip6` or `inet`. Thereby you already specify, whether this flowtable instance shall handle IPv4 packets, IPv6 packets, or both. Creating a flowtable instance is very similar to creating a base chain. When creating it, Nftables registers a callback function within the specified Netfilter hook. You need to specify `hook`, `priority` and the network devices which are involved.

```
# Example, creating flowtable instance "f" in "ingress" hook of IPv4
# with "priority 0" for devices "eth0" and "eth1":
nft add table ip filter
nft add flowtable ip filter f \
   { hook ingress priority 0\; devices = { eth0, eth1 }\; }
```

The specified Netfilter hook must be `ingress`. This hook is special in a way that it does not exist globally for IPv4 or IPv6 like the other Netfilter hooks (`Prerouting`, `Forward`, …). Instead, this hook is located at a very early point in the receive path on OSI layer 2 in function `__netif_receive_skb_core() [https://elixir.bootlin.com/linux/v5.14.7/source/net/core/dev.c#L5366]`. An individual `ingress` hook exists for each network device (NIC). This is why you need to specify the network devices which are involved in your forwarded traffic when creating the flowtable instance. Your flowtable callback is thereby registered only in the `ingress` hooks of those devices, in this example in the `ingress` hook of `eth0` and in the `ingress` hook of `eth1`. In theory you can create more than one flowtable instance within the same `ingress` hook(s). In that case `priority` specifies the sequence in which those instances are traversed by network packets in the same way it does for Nftables base chains. Depending on the *address family* of your Nftables table (in this example it is `ip`), Nftables registers one of the following callback functions within the `ingress` hooks when you create the flowtable instance:

- address family `ip`: `nf_flow_offload_ip_hook() [https://elixir.bootlin.com/linux/v5.14.7/source/net/netfilter/nf_flow_table_ip.c#L328]`
- address family `ip6`: `nf_flow_offload_ipv6_hook() [https://elixir.bootlin.com/linux/v5.14.7/source/net/netfilter/`

nf_flow_table_ip.c#L564]

- address family inet: nf_flow_offload_inet_hook() [https://elixir.bootlin.com/linux/v5.14.7/source/net/netfilter/nf_flow_table_inet.c#L11]

You further need to create an Nftables base chain in the Netfilter forward hook and place a rule inside which filters for the traffic which you actually want to offload to your flowtable:

```
# Example, creating forward chain and rule
# to offload TCP traffic to flowtable "f":
nft add chain ip filter forward { type filter hook forward priority 0\; }
nft add rule ip filter forward ip protocol tcp flow add @f
```

That is all you need. Flowtables is now activated and forwarded TCP connections (*flows* which first need to be recognized by the connection tracking system) will get offloaded to the flowtables fastpath.

A few more words about the Netfilter *ingress* hook: While the other Netfilter hooks exist globally[5] within the packet handling code of IPv4 and yet another set of those same hooks exists within the packet handling code of IPv6, the *ingress* hook exists on OSI layer 2 and one instance of this hook exists for each network device. My article Nftables - Packet flow and Netfilter hooks in detail explains that in detail. This means, if you place an Nftables base chain within e.g. the *forward* hook and choose address family *ip*, then the callback function of your base chain will get registered within the *forward* hook of IPv4 and thereby your base chain will only be traversed by IPv4 packets. If you do the same with address family *inet*, then the callback of your base chain will get registered in both the *forward* hooks of IPv4 and IPv6, and thereby your base chain will be traversed by both IPv4 and IPv6 packets. However, in the example above, you create a *flowtable* within an *Nftables* table of address family *ip* and place it in the *ingress* hooks of the selected network devices on OSI layer 2. So, despite having selected address family *ip*, won't the *flowtable* callback function be traversed by all kinds of layer 3 packets which can be carried by layer 2 (Ethernet) frames, like IPv4, IPv6, ARP, and so on? Yes, it will! However, the first thing the *flowtable* callback function does, is to actively sort out all packets which are not corresponding to your selected address family and send those back to the slowpath. It will only perform the flowtable lookup for packets of the selected address family. BTW: Nftables has yet another address family called *netdev*, which makes it possible to place base chains in the *ingress* hook, which then are traversed by all kinds of packets on OSI layer 2. However, that address family cannot cannot be used for a flowtable.

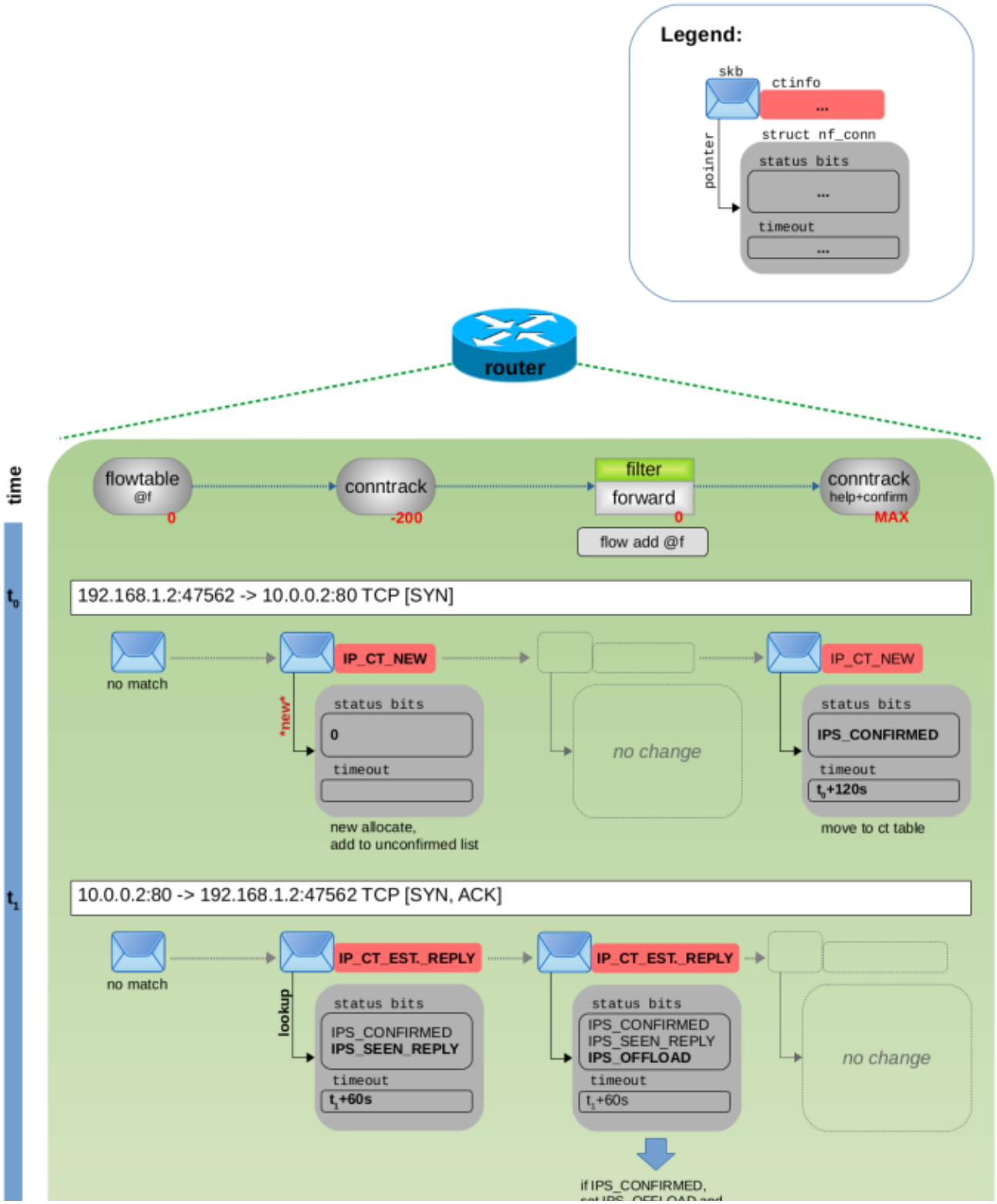
Example: TCP connection

Let's say now this is our Nftables ruleset on the router in Figure 4:

```
table ip filter {
        flowtable f { # (1)
                hook ingress priority 0; devices = { eth0, eth1 };
        }
        chain y {
                type filter hook forward priority 0; policy accept;
                ip protocol tcp flow add @f # (2)
                counter # (3)
        }
}
```

When a new TCP connection is being forwarded by this system, its first two packets (usually SYN, SYN ACK) follow the classic forwarding path (thus, they traverse the Netfilter *Prerouting*, *Forward* and *Postrouting* hooks) and are tracked by connection tracking. Originally the *flowtables* approach was using the 1st packet in *reply* direction to trigger the offload to the fastpath (conntrack state: *established*). However that was changed with this commit [https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable-rc.git/commit/?id=270a8a297f42ecff82060aaa53118361f09c1f7d] (kernel v5.2). Now the trigger is the 1st packet which arrives after conntrack considers this connection as “confirmed”. This could be the 1st packet in reply direction (here SYN ACK), but also a 2nd packet in original direction (here a 2nd SYN), in case there have been no packets in reply direction yet. Nevertheless, this packet then matches rule (2) in the *forward* chain. This rule then creates an entry within the flowtable f for that TCP connection.

The conntrack status bit `IPS_OFFLOAD_BIT [https://elixir.bootlin.com/linux/v5.14.7/source/include/uapi/linux/netfilter/nf_conntrack_common.h#L114]` is set within this tracked connection. This tracked connection instance `struct nf_conn [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_conntrack.h#L72]` is from now on owned (refcount) by the flowtable entry and `nf_conn->timeout [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_conntrack.h#L85]` is set [https://elixir.bootlin.com/linux/v5.14.7/source/net/netfilter/nf_flow_table_core.c#L325] to a very high value, so that it never expires (at least not sooner than flowtables own timeout) (= effectively the conntrack timeout mechanism is switched off). Figure 5 visualizes this sequence of steps for the 3-way TCP handshake. Please compare it to the images and explanations in my article Connection tracking (conntrack) - Part 3: State and Examples to gain full understanding on what is different here compared to how things work without flowtables.



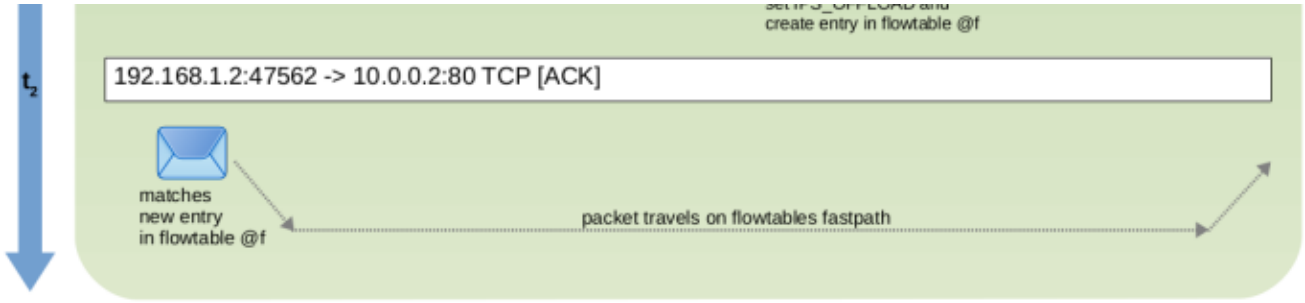


Figure 5: 3-way handshake of a new forwarded TCP connection, being tracked by *connection tracking* and being offloaded to a *flowtable* (click to enlarge).

All following packets of that TCP connection (3rd, 4th, … packet) now match to the new entry in flowtable `f` in the Netfilter *ingress* hooks of `eth0` and `eth1`. Thus, those packets do not follow the classic forwarding path anymore and instead use the fastpath bypass. They are forwarded to where the flowtable entry specifies by means of its cached *routing decision* in terms of output network device and *nexthop*. You can indirectly observe this with rule (3) in the Nftables ruleset shown above, because its counter is only incremented for the first two packets, but not for those following packets anymore. In case of a flowtable misses (= for packets which do not match any entry in the flowtable `f`), those packet follows the classic forward path. You can observe the offloaded flow with the conntrack tool. It is now marked as `[OFFLOAD]`:

```
conntrack -L
tcp      6 82661 SYN_RECV src=192.168.1.2 dst=10.0.0.2 sport=47562 dport=80 \
 src=10.0.0.2 dst=192.168.1.2 sport=80 dport=47562 [OFFLOAD] mark=0 use=1
```

Be aware, that version 1.4.5 of the *conntrack-tools*, which is e.g. used in debian *buster*, is too old to show the offload status. Version 1.4.6 on debian *bullseye* or newer is required to see the `[OFFLOAD]` keyword in the output of command `conntrack -L`. That feature has been added in 2019-08-09 with this commit [https://git.netfilter.org/conntrack-tools/commit/?id=de12e29bf35b1da51944c826beb34acf48d90289]. As an alternative, you can also use the (deprecated) proc file `nf_conntrack`. It will also show you the offload status (tested with kernel v5.10):

```
cat /proc/net/nf_conntrack
tcp      6 82661 SYN_RECV src=192.168.1.2 .... [OFFLOAD] mark=0 use=1
```

The Tear Down Feature

The flowtables mechanism has a timeout for each flow, like the connection tracking system. If it expires (=if no packets are seen for `NF_FLOW_TIMEOUT` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L177] (30) seconds, which is the default timeout value), a garbage collector removes that entry from the flowtable. Thus, the remaining packets are thereby thrown back to the classic forwarding path and to connection tracking, which then re-starts its own timeout mechanism. If `TCP RST` or `FIN` packets are seen, those have the same effect. Thereby, if a flow only produces small bursts of packets here and there and the gaps between those bursts are bigger than 30 seconds, but still small enough so that connection tracking does not make the flow expire, then that flow can jump several times between slowpath and fastpath, being offloaded and torn down again and again. You can change the timeout individually for TCP and for UDP protocol by using one of the following sysctl's, which effectively change the used timeout for all offloaded flows in the current network namespace:

```
$ sudo sysctl -a -r nf_flowtable
net.netfilter.nf_flowtable_tcp_timeout = 30
net.netfilter.nf_flowtable_udp_timeout = 30
```

The Offload Action

Kernel function `nft_flow_offload_eval()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/netfilter/nft_flow_offload.c#L271] is the one being executed when the Nftables statement `flow add @f` is being evaluated for a network packet traversing the *forward* chain. (There is an older syntax `flow offload @f`, which is the reason why this is called the “offload action”. You might still see it sometimes. However, the newer syntax, the one which you actually should use, is `flow add @f`. Of course, `f` is just a variable name.) This function does a bunch of initial checks on the traversing network packet and will skip the offloading (=won't create a flowtable entry) and ignore the packet in the following cases:

- packet has an `skb_sec_path(skb)` [https://elixir.bootlin.com/linux/v5.14.7/source/include/linux/skbuff.h#L4457] extension attached[6)]
- IPv4 header contains IP options
- it is not a TCP or UDP packet
- TCP header has `FIN` or `RST` flags
- packet is not tracked by conntrack, e.g. state *invalid* or *untracked*
- conntrack helper extension exists for this flow
- conntrack status bit `IPS_SEQ_ADJUST` or `IPS_NAT_CLASH` is set for this flow
- conntrack status bit `IPS_CONFIRMED` is not yet set for this flow
- conntrack status bit `IPS_OFFLOAD` is already set for this flow
- … (no guarantee for completeness)

If the traversing network packet, however, passes all these initial checks, this function will set conntrack status bit `IPS_OFFLOAD` for this flow and create and add a new entry to the flowtable. The most essential information this new entry will contain is pointers to two instances of *routing decisions*, one for each flow direction (original and reply direction) of the flow. The network packet which triggered the call of function `nft_flow_offload_eval()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/netfilter/nft_flow_offload.c#L271], already delivers the attached *routing decision* for that packet's flow direction and this one is now saved within the new flowtable entry. However, to finish creation of this new flowtable entry, also the *routing decision* for the opposite flow direction is required. Thus, a routing lookup is performed. This is done by function `nft_flow_route()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/netfilter/nft_flow_offload.c#L216].

struct flow_offload

`struct flow_offload` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L168] represents a flowtable entry. Here some member variables:

- `struct flow_offload_tuple_rhash tuplehash[2];` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L169]
  array containing 2 instances of `struct flow_offload_tuple`

- `struct nf_conn *ct;` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L170]
  pointer to the `struct nf_conn` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_conntrack.h#L72] instance representing this connection/flow tracked by conntrack

- `unsigned long flags;` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L171]
  integer, containing one of the values defined in `enum nf_flow_flags` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L153]

- `u16 type;` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L172]
  is normally set to `NF_FLOW_OFFLOAD_ROUTE` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L165]

- `u32 timeout;` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L173]
  integer; works nearly the same way as the conntrack timeout… if no further network packet arrives within 30s, this entry in the flowtable is deleted again and thereby the flow is being thrown back to the slowpath and is handled by conntrack again

struct flow_offload_tuple

`struct flow_offload_tuple` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L103] represents one flow direction (original/reply). This struct has 2 purposes:

1. `struct flow_offload` contains an array of 2 instances of this struct, one for each flow direction. Those instances hold all flow-direction-specific data the flowtable needs to remember.
2. It serves as a key for doing the *flowtables* hash table lookup for network packets traversing the *ingress* hook. This lookup is implemented nearly in the same way than the conntrack lookup.

Thus, this struct is the flowtables equivalent of the conntrack `struct nf_conntrack_tuple` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_conntrack_tuple.h#L37]. Most member variables are the same. Well, their names differ, but content and purpose are the same:

- `struct in_addr src_v4;`
- `struct in6_addr src_v6;`
- `struct in_addr dst_v4;`
- `struct in6_addr dst_v6;`
- `__be16 src_port;`
- `__be16 dst_port;`
- `u8 l3proto;`
- `u8 l4proto;`
- `u8 dir;`

Some more member variables, which you won't find in the conntrack equivalent, are:

- `int iifidx;` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L117]
  index of input interface for this flow direction
- `u16 mtu;` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L133]
  remembering the MTU (maximum transmission unit) value
- `struct dst_entry *dst_cache` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L136]
  cached *routing decision* for this flow direction, as described above, see Figure 2

When you look into source code, you'll see that I simplified things a little in my description here. Several member variables are organized into *unions* or sub *structures*, but that does not change semantics. You'll also see some additional member variables I didn't mention. Those mostly are used in cases when you combine *flowtables* with features like *bridging*, *VLAN* or *PPPoE*. I won't dive into those topics in this article.

Fastpath function

Function `nf_flow_offload_ip_hook()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/netfilter/nf_flow_table_ip.c#L328] is the flowtables callback in the Netfilter *ingress* hook for traversing IPv4 packets. Let's take a look at what this function does.

- It does some initial checks and throws the traversing packet back to the slowpath in these cases:
  - packet is not IPv4
  - packet is an IPv4 fragment
  - packet IPv4 header contains IPv4 options
  - packet L4 protocol is not TCP or UDP
  - packet IPv4 TTL <= 1
  - packet has an incomplete Layer 3 or Layer 4 header
- It creates an instance of `struct flow_offload_tuple` (a “tuple”) and fills it with data from the traversing packet's IPv4 and TCP/UDP header: source and destination IP address, source and destination TCP/UDP port, l3proto, l4proto, iifidx.
- It calculates a hash from these (not all!) members of this “tuple” and performs the flowtable (hashtable) lookup. This works nearly identically to the conntrack lookup.
- After that, it does some further checks and throws the packet back to the slowpath in these cases:
  - no match is found
  - match is found, but flag `NF_FLOW_TEARDOWN` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L156] is set in this flowtable entry
  - match is found, but conntrack status bit `IPS_DYING` [https://elixir.bootlin.com/linux/v5.14.7/source/include/uapi/linux/netfilter/nf_conntrack_common.h#L86] is set for this flow
  - match is found, but packet size exceeds MTU set in flowtable entry
  - match is found, but packet TCP flag FIN or RST is set (this triggers Teardown)
- It refreshes the `timeout` of this flowtable entry, setting it to “now + 30 seconds”. This means teardown will happen for this flowtable entry, if no further matching packet arrives for >30 seconds.

- If the `flags` member of the flowtable entry contains bit `NF_FLOW_SNAT [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L154]` or `NF_FLOW_DNAT [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netfilter/nf_flow_table.h#L155]`, it performs NAT on the traversing packet.
- It decrements IPv4 TTL of the traversing packet by one.
- It updates the conntrack accounting packet/byte counters (if this feature is activated).
- In case of IPsec/Xfrm, it does some special handling here.
- It attaches the cached *routing decision* to the traversing packet.
- The end of the fastpath has been reached: the traversing packet is re-inserted into the neighbouring subsystem by calling function `neigh_xmit() [https://elixir.bootlin.com/linux/v5.14.7/source/net/core/neighbour.c#L2991]`.

Hardware fastpath

Network hardware offloading/acceleration has been implemented in countless ways and flavours by many chip vendors over the years. However, many have been implemented in form of individual solutions with no attempt to find potential synergy with other solutions of that kind or to create a common software interface. Further, software interfaces / drivers which made those features available to the Linux kernel, often didn't find their way into the mainline kernel sources. *Flowtables* has been created with the goal to not only create a software fastpath, but additionally to provide a common mainline interface for hardware-based solutions, which intend to offload entire flows. However, as things like this take time, by the time of writing it seems not many chip vendors have jumped on that train yet[7]. The Nftables wiki [https://wiki.nftables.org/wiki-nftables/index.php/Flowtables] mentions that support for hardware from vendor *Mellanox* has been added [https://lwn.net/Articles/804384/]. Nevertheless, when a flow is actually offloaded into hardware, the conntrack status bit `IPS_HW_OFFLOAD` is set and this flow is marked as `[HW_OFFLOAD]` in the output of command `conntrack -L`.

Flowtable Accounting

The *connection tracking* system supports accounting, which means counting packets and bytes for each flow and for each flow direction. This feature is deactivated by default. You can activate it with this sysctl:

```
sudo sysctl -w net.netfilter.nf_conntrack_acct=1
```

Then you can observe packet and byte counters for each flow and each flow direction e.g. with command `conntrack -L`. By default, packets and bytes which travel on the *flowtables* fastpath are not counted here, just packets on the slowpath are counted. However, you can also activate that, by adding keyword `counter` when creating your flowtable:

```
nft add flowtable ip filter f \
   { hook ingress priority 0\; devices = { eth0, eth1 }\; counter\; }
```

After that, conntrack accounting will count both packets (and bytes) on the slowpath and on the fastpath.

Context

The described behavior and implementation has been observed on a Debian 11 (bullseye) system with using Debian *backports* on *amd64* architecture.

- kernel: `5.14.7`
- nftables: `0.9.8`
- conntrack: `1.4.6`

Feedback

Feedback to this article is very welcome! Please be aware that I did not develop or contribute to any of the software components described here. I'm merely some developer who took a look at the source code and did some practical experimenting. If you find something which I might have misunderstood or described incorrectly here, then I would be very grateful, if you bring this to my attention and of course I'll then fix my content asap accordingly.

References

- man nft(8) [https://www.netfilter.org/projects/nftables/manpage.html]
- kernel.org: Netfilter's flowtable infrastructure [https://www.kernel.org/doc/html/latest/networking/nf_flowtable.html]
- wiki.nftables.org: Flowtables [https://wiki.nftables.org/wiki-nftables/index.php/Flowtables]

Continue with next article

Flowtables - Part 2: IPsec gateway in tunnel-mode

published 2022-08-08, *last modified 2022-12-28*

1)
It works the same way for IPv6, just the functions handling the packets on OSI layer 3 are different.
2)
Actually, to be more precise: It is being placed in the *ingress* hooks of all network interfaces which are involved in this forwarded traffic. More on that later…
3)
if existing
4)
especially article 3 of the series
5)
Well… "globally" within a network namespace, but that's another topic.
6)
This is a tricky special case, which occurs when using *flowtables* together with IPsec in tunnel-mode. I'll describe it in the 2nd article of this series.
7)
Someone correct me here please, in case that is wrong. I did not investigate this deeply.