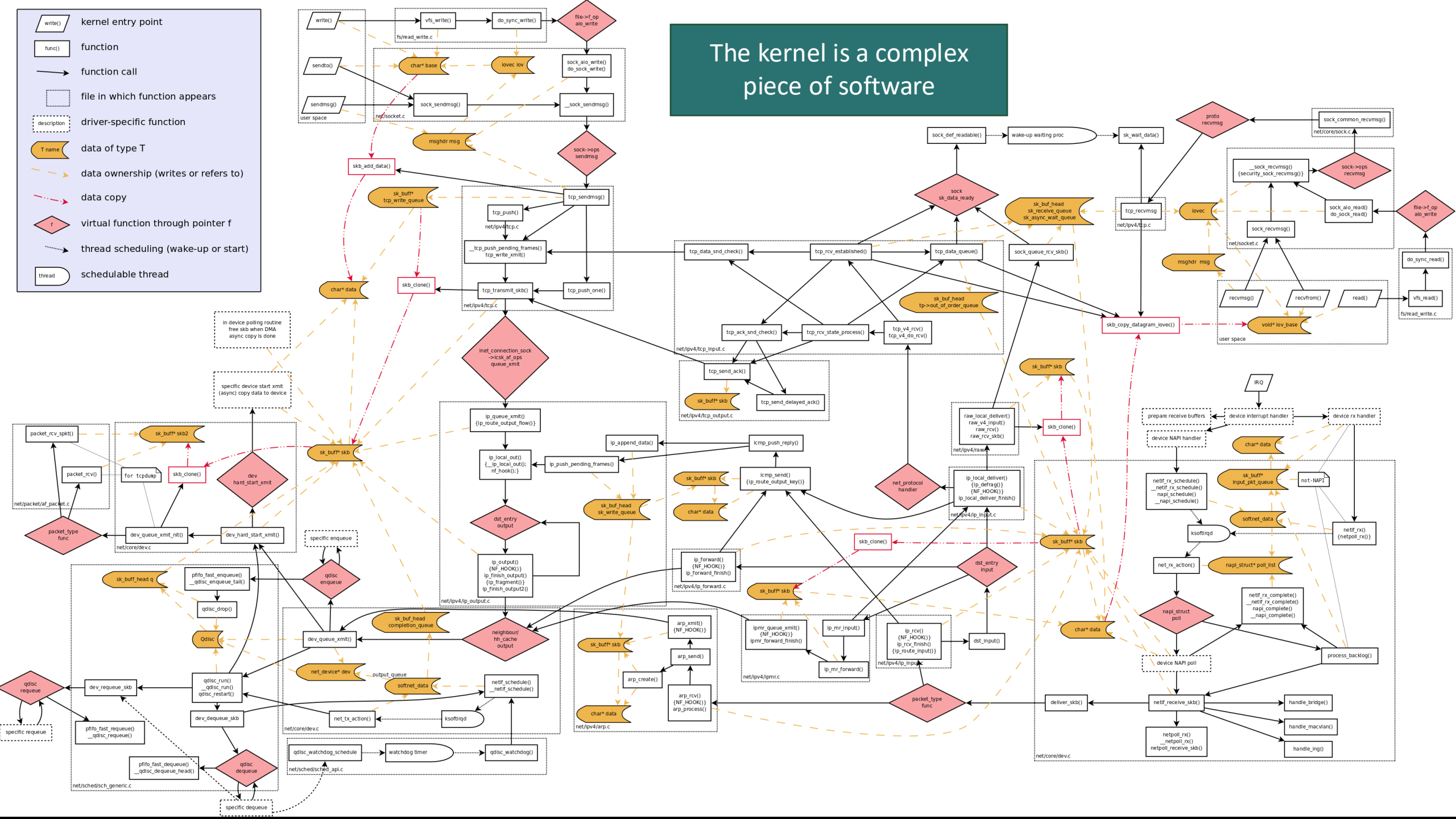
CLOUD POUR IOT – FAST KERNEL

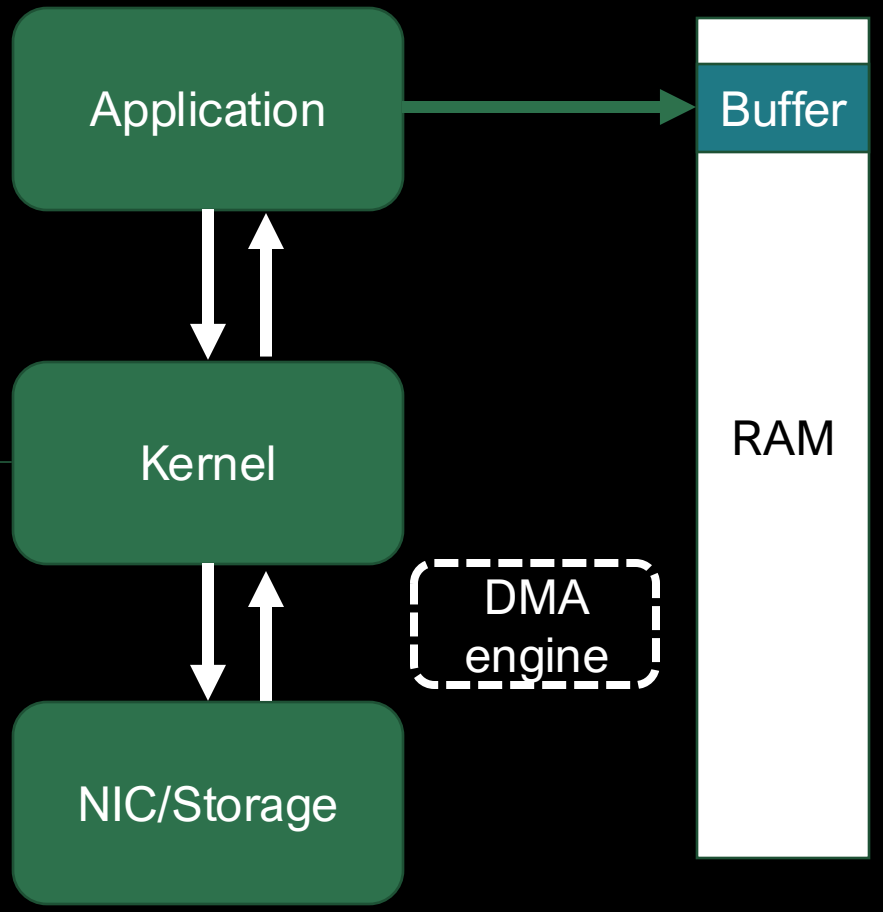
ESIR


write()
kernel entry point
func()
function
function call
file in which function appears
description
driver-specific function
T name
data of type T
data ownership (writes or refers to)
data copy
f
virtual function through pointer f
thread scheduling (wake-up or start)
thread
schedulable thread
The kernel is a complex piece of software
write()
vfs_write()
do_sync_write()
file->f_op aio_write
fs/read_write.c
sendto()
char* base
iovec iov
sock_aio_write() do_sock_write()
sendmsg()
sock_sendmsg()
__sock_sendmsg()
user space
net/socket.c
msghdr msg
sock->ops sendmsg
skb_add_data()
sk_buff* tcp_write_queue
tcp_sendmsg()
tcp_push()
net/ipv4/tcp.c
__tcp_push_pending_frames() tcp_write_xmit()
char* data
skb_clone()
tcp_transmit_skb()
tcp_push_one()
net/ipv4/tcp.c
In device polling routine free skb when DMA async copy is done
inet_connection_sock ->icsk_af_ops queue_xmit
specific device start xmit (async) copy data to device
ip_queue_xmit() {ip_route_output_flow()}
sk_buff* skb
ip_local_out() {__ip_local_out(); nf_hook();}
ip_push_pending_frames()
ip_append_data()
icmp_push_reply()
dst_entry output
sk_buff_head sk_write_queue
ip_output() {NF_HOOK()} ip_finish_output() {ip_fragment()} ip_finish_output2()
net/ipv4/ip_output.c
packet_rcv_spkt()
sk_buff* skb2
packet_rcv()
for tcpdump
skb_clone()
dev hard_start_xmit
packet_type func
dev_queue_xmit_nit()
dev_hard_start_xmit()
net/packet/af_packet.c
net/core/dev.c
specific enqueue
qdisc enqueue
sk_buff_head q
pfifo_fast_enqueue() __qdisc_enqueue_tail()
qdisc_drop()
Qdisc
dev_queue_xmit()
sk_buff_head completion_queue
neighbour/ hh_cache output
net_device* dev
output_queue
softnet_data
netif_schedule() __netif_schedule()
qdisc requeue
dev_requeue_skb
qdisc_run() __qdisc_run() qdisc_restart()
dev_dequeue_skb
net_tx_action()
ksoftirqd
specific requeue
pfifo_fast_requeue() __qdisc_requeue()
net/core/dev.c
qdisc_watchdog_schedule
watchdog timer
qdisc_watchdog()
net/sched/sch_generic.c
net/sched/sch_api.c
pfifo_fast_dequeue() __qdisc_dequeue_head()
qdisc dequeue
specific dequeue
arp_xmit() {NF_HOOK()}
arp_send()
arp_create()
arp_rcv() {NF_HOOK()} arp_process()
sk_buff* skb
char* data
net/ipv4/arp.c
ipmr_queue_xmit() {NF_HOOK()} ipmr_forward_finish()
ip_mr_input()
ip_mr_forward()
net/ipv4/ipmr.c
ip_forward() {NF_HOOK()} ip_forward_finish()
net/ipv4/ip_forward.c
skb_clone()
dst_entry input
dst_input()
ip_rcv() {NF_HOOK()} ip_rcv_finish() {ip_route_input()}
net/ipv4/ip_input.c
packet_type func
net_protocol handler
icmp_send() {ip_route_output_key()}
sock_def_readable()
wake-up waiting proc
sk_wait_data()
sock sk_data_ready
sk_buf_head sk_receive_queue sk_async_wait_queue
sock_queue_rcv_skb()
tcp_data_snd_check()
tcp_rcv_established()
tcp_data_queue()
sk_buf_head tp->out_of_order_queue
tcp_ack_snd_check()
tcp_rcv_state_process()
tcp_v4_rcv() tcp_v4_do_rcv()
net/ipv4/tcp_input.c
tcp_send_ack()
tcp_send_delayed_ack()
net/ipv4/tcp_output.c
raw_local_deliver() raw_v4_input() raw_rcv() raw_rcv_skb()
net/ipv4/raw.c
ip_local_deliver() {ip_defrag()} {NF_HOOK()} ip_local_deliver_finish()
net/ipv4/ip_input.c
skb_clone()
tcp_recvmsg
net/ipv4/tcp.c
iovec
skb_copy_datagram_iovec()
proto recvmsg
sock_common_recvmsg()
net/core/sock.c
__sock_recvmsg() {security_sock_recvmsg()}
sock->ops recvmsg
sock_aio_read() do_sock_read()
sock_recvmsg()
net/socket.c
file->f_op aio_write
do_sync_read()
vfs_read()
fs/read_write.c
msghdr msg
recvmsg()
recvfrom()
read()
void* iov_base
user space
IRQ
prepare receive buffers
device Interrupt handler
device rx handler
device NAPI handler
char* data
netif_rx_schedule() __netif_rx_schedule() napi_schedule() __napi_schedule()
sk_buff* input_pkt_queue
not-NAPI
softnet_data
ksoftirqd
netif_rx() {netpoll_rx()}
net_rx_action()
napi_struct* poll_list
napi_struct poll
netif_rx_complete() __netif_rx_complete() napi_complete() __napi_complete()
process_backlog()
device NAPI poll
deliver_skb()
netif_receive_skb()
handle_bridge()
handle_macvlan()
handle_ing()
netpoll_rx() __netpoll_rx() netpoll_receive_skb()
net/core/dev.c

CLOUD – FAST KERNEL

Traditionnellement, comment fonctionne la couche réseau d'un OS ?

- Contrôle des droits
- Traversée des couches protocolaires (L1-L4/5)
- Création des structures de données
- Contrôle du matériel



- Le noyau prend un temps considérable lors du traitement des paquets/blocks de données
- Le noyau reste relativement rigide lors qu'il faut des mise à jour à la volée

Alors comment faire ?

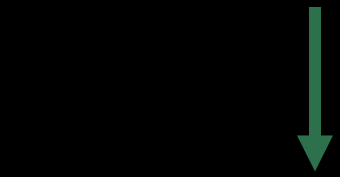
CLOUD – FAST KERNEL

Traditionnellement, comment fonctionne la couche réseau d'un OS ?

- Le noyau prend un temps considérable lors du traitement des paquets/blocks de données
- Le noyau reste relativement rigide lors qu'il faut des mise à jour à la volée

Alors comment faire ?

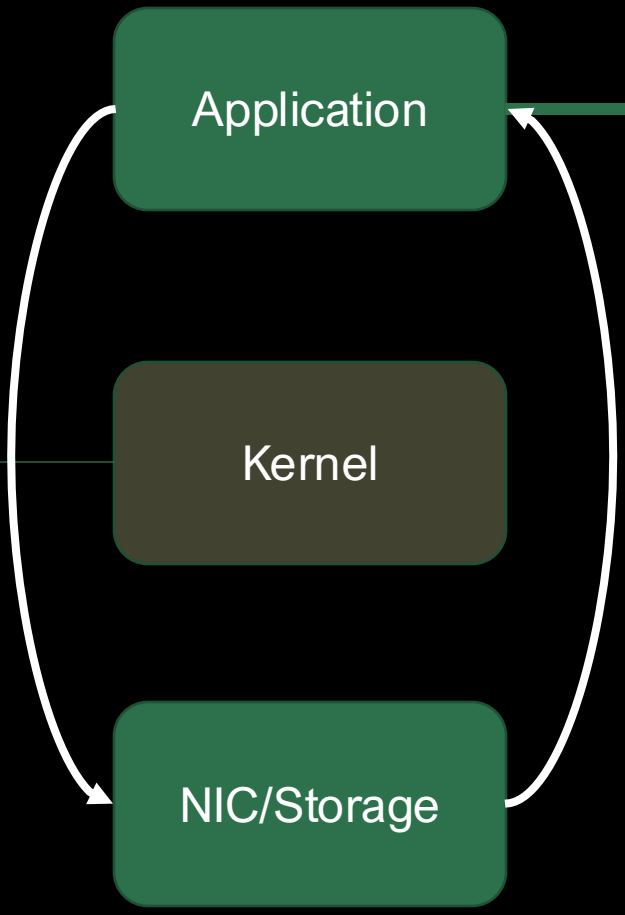
Et si on retirer le noyau de la chaine de traitement ?

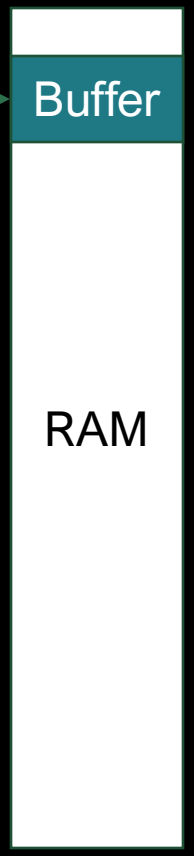


Kernel Bypass

CLOUD – FAST KERNEL

Kernel Bypass - Principles





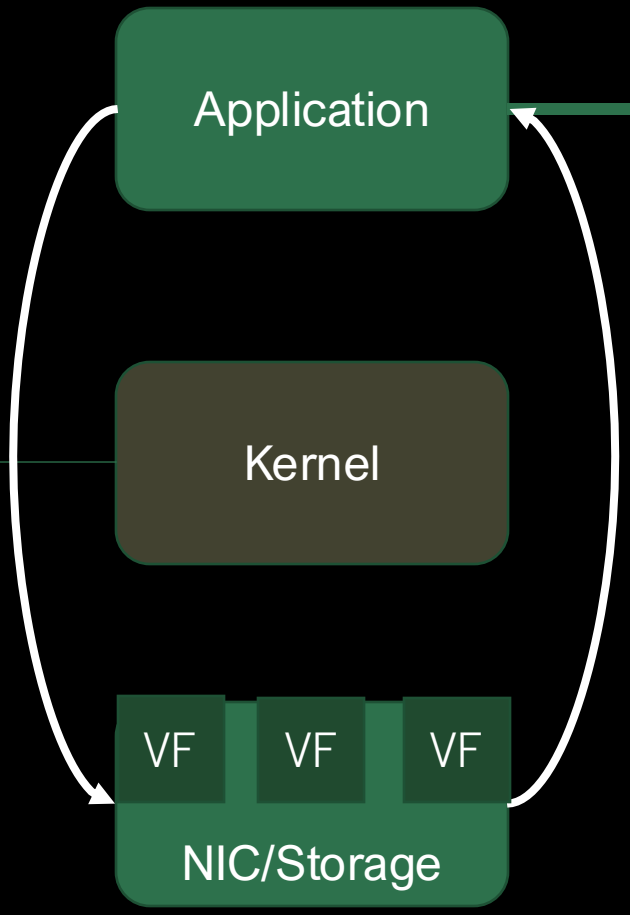
On va ignorer le noyau pour aller plus rapidement

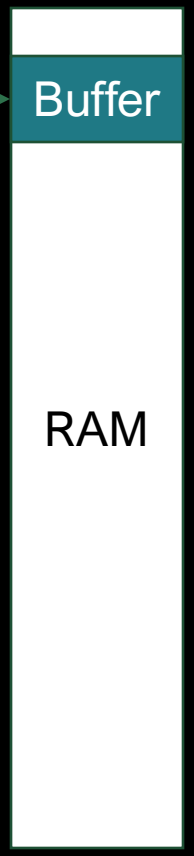
L'application communique directement avec le matériel.

Le matériel doit pouvoir gérer plusieurs requêtes venant de plusieurs applications

CLOUD – FAST KERNEL

Kernel Bypass - Principles





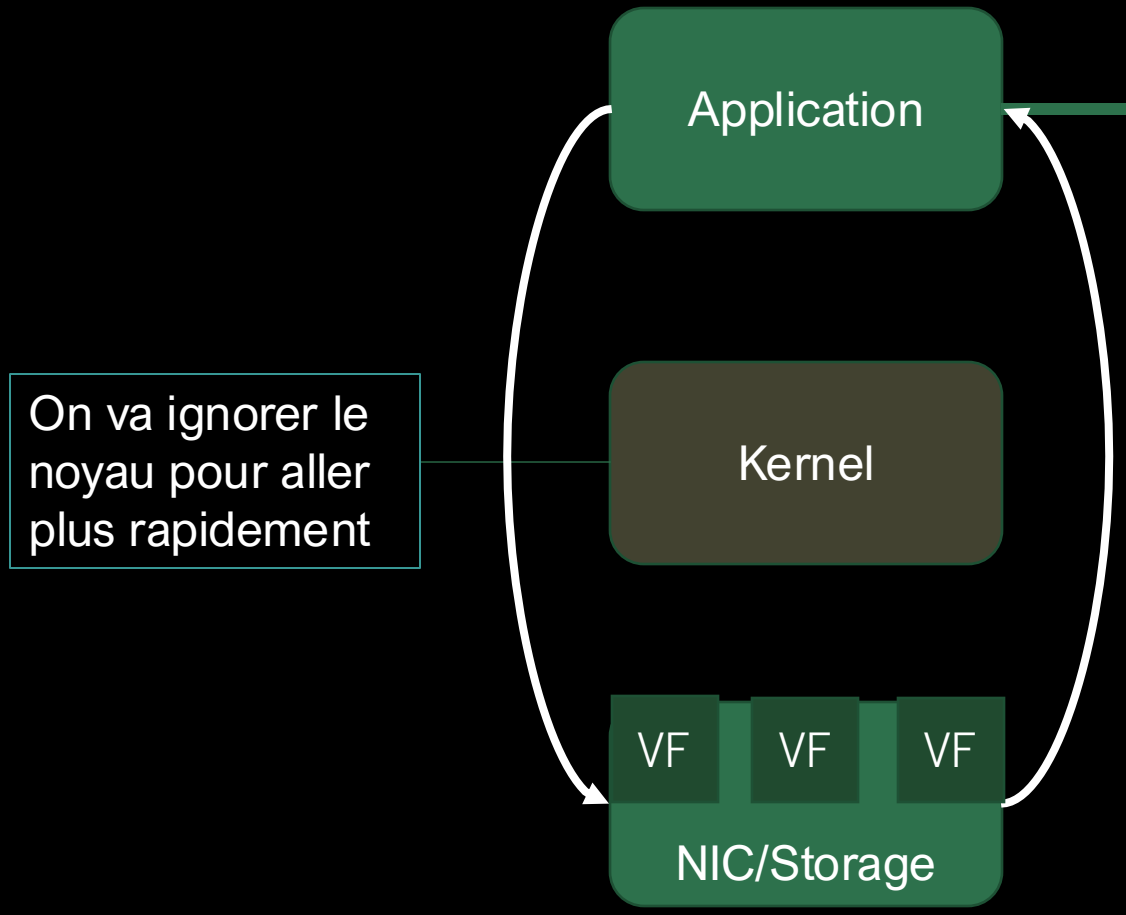
On va ignorer le noyau pour aller plus rapidement

L'application communique directement avec le matériel.

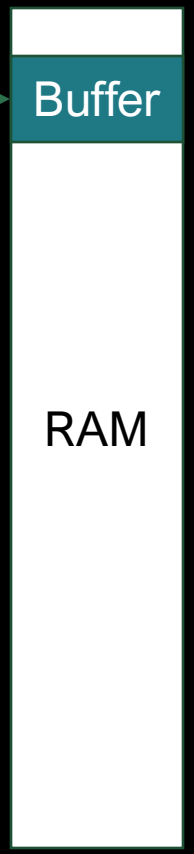
Afin de se multiplexer pour plusieurs applications, le matériel aura une interface quelle exposera pour une entité : VFs (virtual functions)

CLOUD – FAST KERNEL

Kernel Bypass – Le noyau avait quand même un rôle important



On va ignorer le noyau pour aller plus rapidement

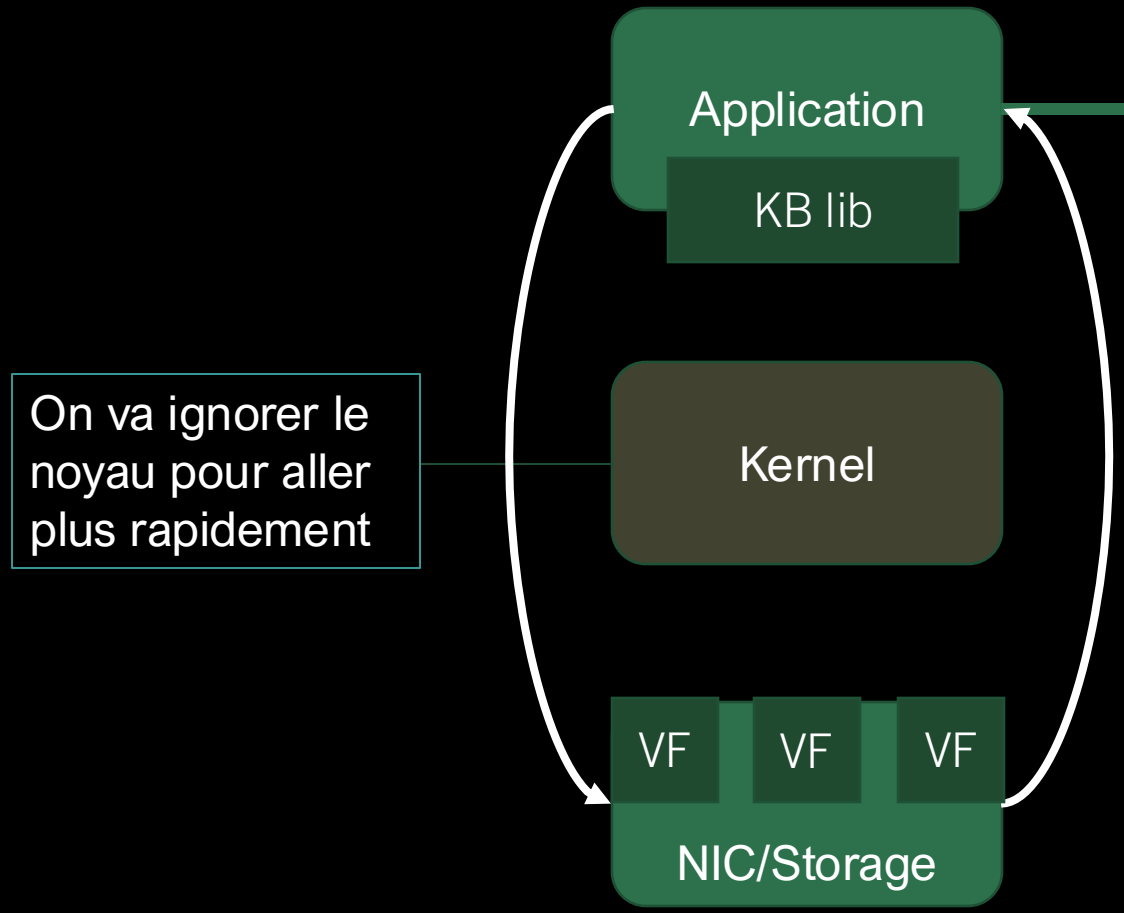


Toute l'abstraction du noyau disparait et les applications doivent réécrire plein de fonctionnalités

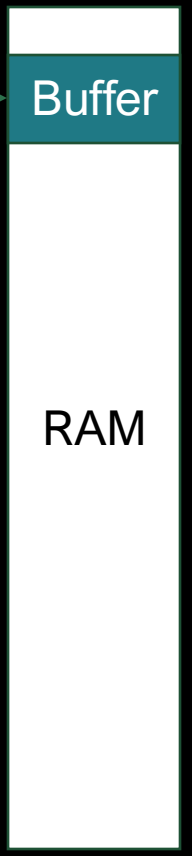
- Contrôle des requêtes
- Vérification des droits
- Tolérance aux fautes
- Isolation des pages mémoires

CLOUD – FAST KERNEL

Kernel Bypass – Le noyau avait quand même un rôle important



On va ignorer le noyau pour aller plus rapidement



Plusieurs librairies pour faire du kernel bypass existent mais sont complexe à utiliser

- Contrôle des requêtes
- Vérification des droits
- Tolérance aux fautes
- Isolation des pages

Irene Zhang et al. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems – SOSP'21

Timothy Stamler et al. zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO – ATC'22

CLOUD – FAST KERNEL

Traditionnellement, comment fonctionne la couche réseau d'un OS ?

- Le noyau prend un temps considérable lors du traitement des paquets/blocks de données

- Le noyau reste relativement rigide lors qu'il faut des mise à jour à la volée

Alors comment faire ?

Et si on changeait dynamiquement le comportement du noyau ?

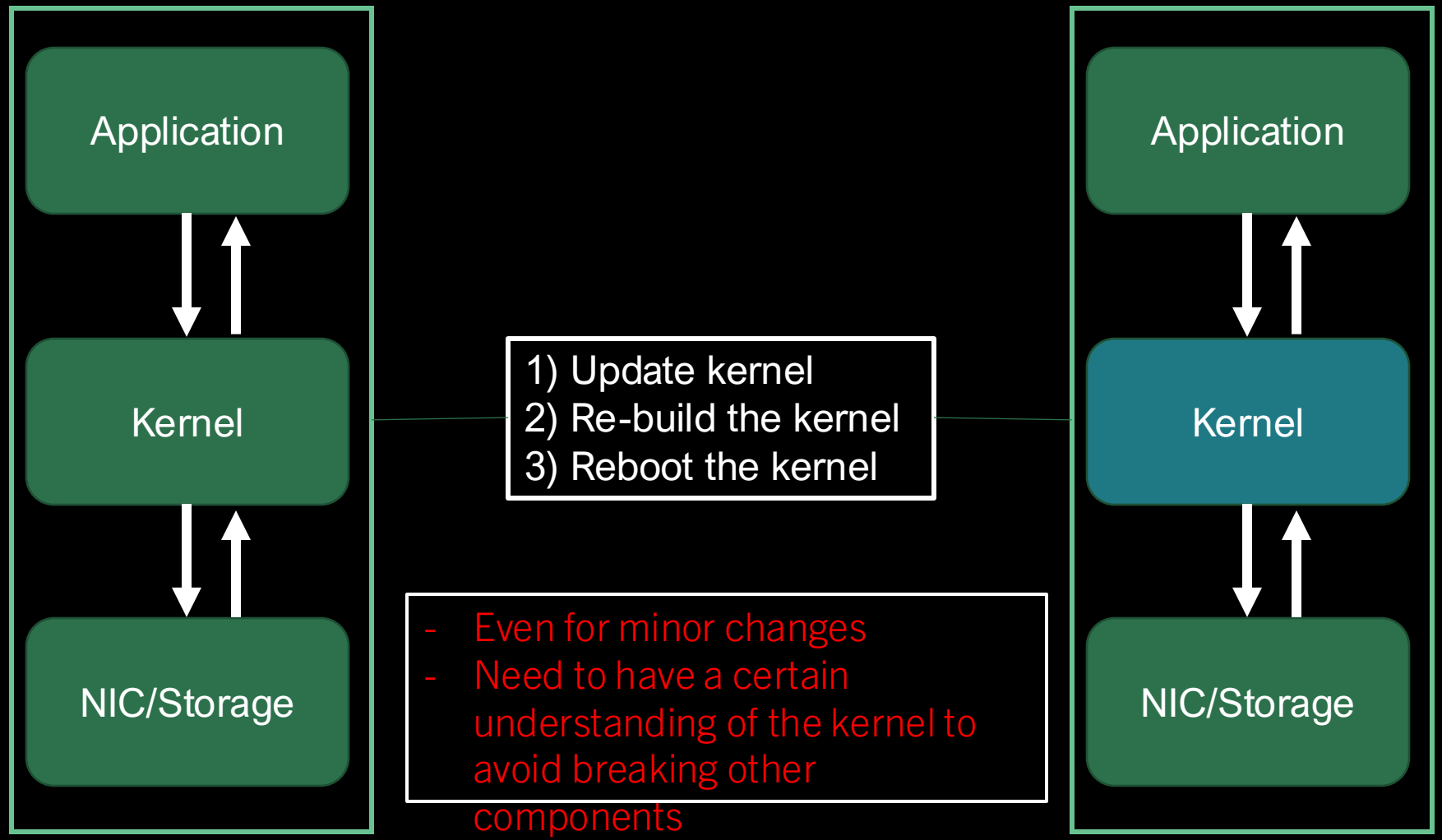
Modular kernel

Et si on retirer le noyau de la chaine de traitement ?

Kernel Bypass

CLOUD – FAST KERNEL

Comment faire si je peux optimiser ou personnaliser une partie du noyau ?



CLOUD – FAST KERNEL

Quelques mécanismes existent: Live patching, modules

- Live patching: dépends de ce qu'on modifie dû à kexec
- Les modules modifient le comportement de certaines fonctions mais ne permettent pas l'intéraction avec l'espace user (linux uio trop récent)
- Comment faire pour avoir plus de modularité ? De façon générique à partir de l'espace utilisateur ?

CLOUD – FAST KERNEL

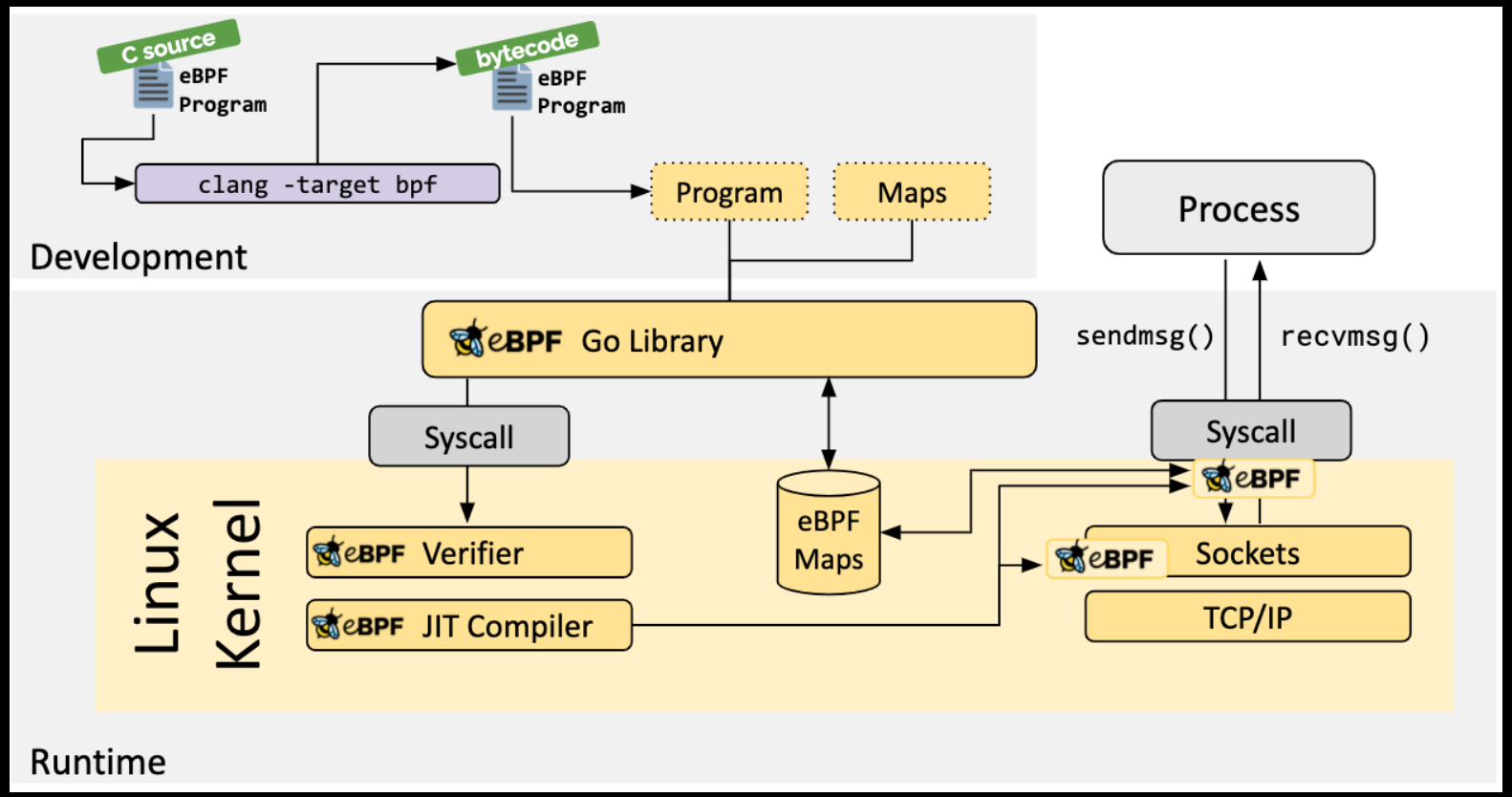
Quelques mécanismes existent: Live patching, modules

- Live patching: dépends de ce qu'on modifie dû à kexec
- Les modules modifient le comportement de certaines fonctions mais ne permettent pas l'intéraction avec l'espace user (linux uio trop récent)
- Comment faire pour avoir plus de modularité ? De façon générique à partir de l'espace utilisateur ?

eBPF (extended Berkeley Packet Filter)

CLOUD – FAST KERNEL

eBPF (extended Berkeley Packet Filter)



Une machine VM qui s'exécute lorsque votre noyau s'exécute en interprétant du bytecode.

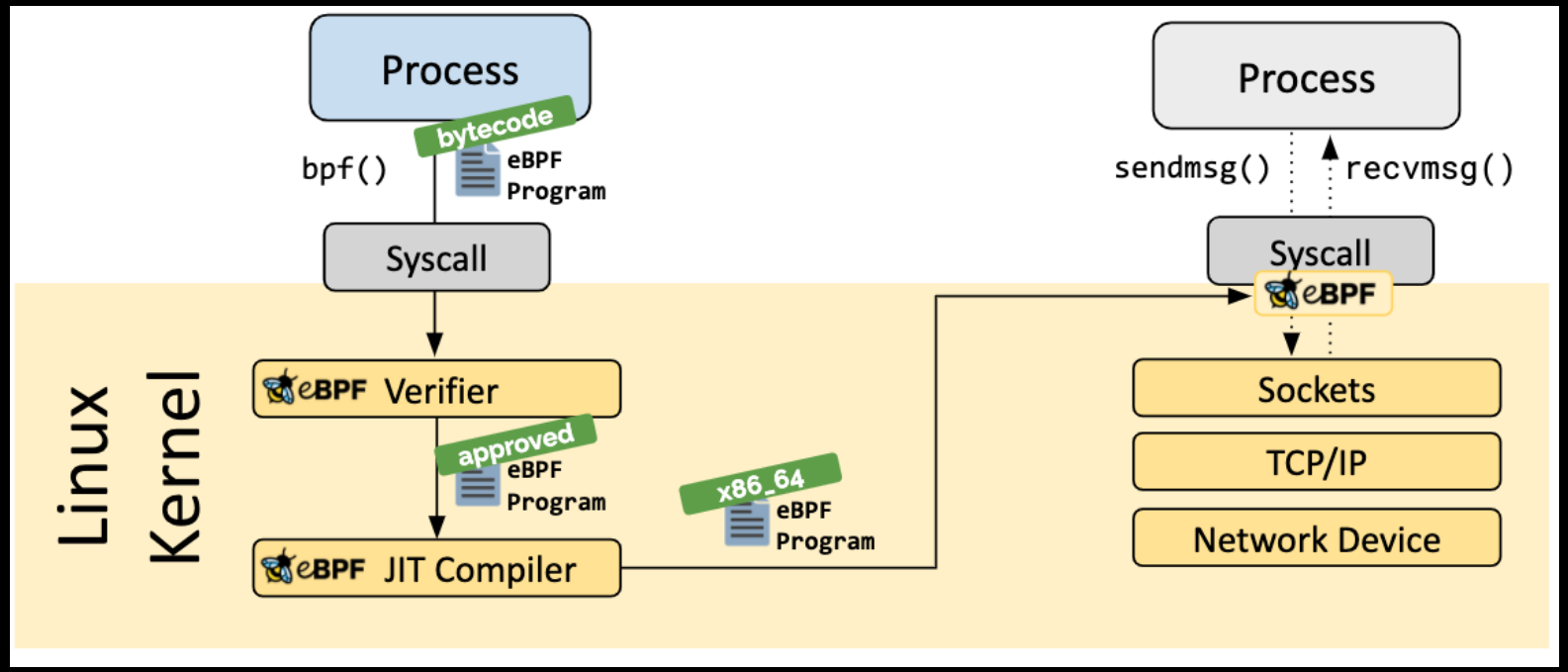
En fonction des hooks que vous définissez, le code de votre programme eBPF est introduit à l'execution via un **compilateur JIT (Just In Time)**

Votre programme s'exécute en mode noyau mais peut obtenir des information du mode utilisateur grâce aux **Maps**

https://ebpf.io/

CLOUD – FAST KERNEL

eBPF (extended Berkeley Packet Filter)



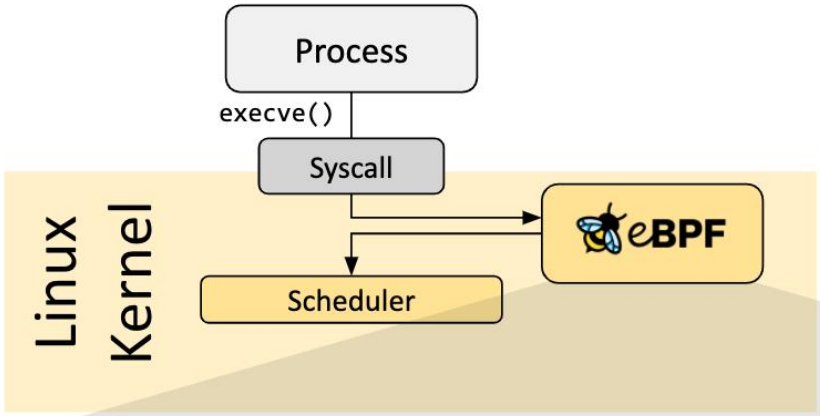
Chaque code eBPF est vérifié pour s'assurer de l'absence de bugs et de sa terminaison

Les structures de données sont optimisées avec les structure de contrôle comme les boucles (qui sont déroulés)

https://ebpf.io/

CLOUD – FAST KERNEL

eBPF (extended Berkeley Packet Filter)




```c
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

        bpf_get_current_comm(&event.comm, sizeof(event.comm));
        comm_events.perf_submit(ctx, &event, sizeof(event));

        return 0;
}
```

Des **helpers** sont à disposition de l'utilisateur pour réaliser certaines opérations

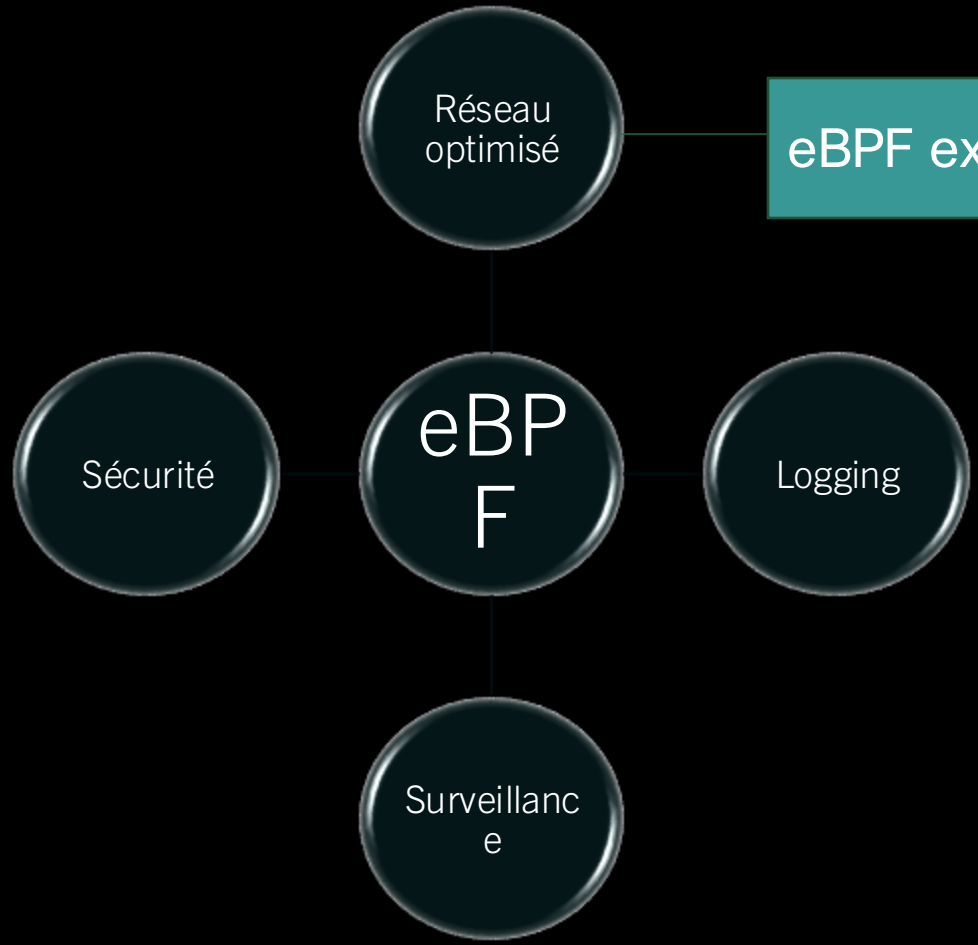
Une compréhension minimaliste du sous-système noyau cible est requise pour déterminer les **bons hooks**

man bpf-helpers

https://ebpf.io/

CLOUD – FAST KERNEL

eBPF (extended Berkeley Packet Filter)



CLOUD – FAST KERNEL

eBPF (extended Berkeley Packet Filter)


```c
struct bpf_map_def sec("maps") processes = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = sizeof(struct key_process_t),
    .value_size = sizeof(u64),
    .max_entries = PID_MAX_LIMIT,
};
```

```c
SEC("prog")
__u32 int main_func(struct rq *ctx)
{
    struct rt_rq *rt_rq = &ctx->rt;
    struct rt_prio_array *array = &rt_rq->active;
    struct sched_rt_entity *next = NULL;
    struct list_head *queue;
    int idx;

    return sched_find_first_bit(array->bitmap);

}
```

Map definition and code example exploiting ebpf.

CLOUD – FAST KERNEL

eBPF (extended Berkeley Packet Filter)


```
    ret = bpf_prog_load("bpf_prog_fifo.o", BPF_PROG_TYPE_SCHED, &obj,
&progfd);
    if (ret) {
        printf("Failed to load bpf program\n");
        exit(1);
    }

    ret = bpf_prog_attach(progfd, 0, BPF_SCHED, 0);
    if (ret) {
        printf("Failed to attach bpf\n");
        exit(1);
    }
```

```
make
clang bpf_load.c -lbpf
./a.out
```

Building the function that uses eBPF with clang and running it.
More details: https://github.com/djobiii2078/ebpf-sched-interface/