Toward Flexible and Efficient In-Kernel Network Function Chaining with IOVisor

Fulvio Risso

Abstract

The eBPF Linux module, which represents the main component of the IOVisor technology, became part of the Linux kernel in 2013. This module enables arbitrary code to be dynamically injected and executed in the Linux kernel while at the same time providing hard safety guarantees in order to preserve the integrity of the system.

While, so far, this component has been used mainly for tracing, monitoring and statistics (in fact, several tools exist that extract information from network traffic and other kernel events such as page faults, system calls, and more), recent projects proposed its usage also for the creation of complex network functions.

This tutorial focuses on the high performance network processing capabilities of IOVisor and it presents the state of the art of the above technology, including XDP (eXpress Data Path), which enables a vanilla Linux kernel to sustain a 10Gbps wire-rate throughput. In addition, it presents the recent extensions of the Iovisor technology that allow the creation of complex network functions (switch, router, NAT, load balancer, firewall, etc.), including both data and control plane. This enables the creation of arbitrary modules, dynamically injectable at run-time, which can be used to create complex service chains and datacenter-wide services (such as the Cilium project).

Finally, this tutorial will summarize the possible interactions of IOVisor with other emerging technologies, such as OpenFlow/OpenState, P4, and SmartNICs.

Why is eBPF cool?

June 2018, Layer 4 Load Balancing at Facebook
https://atscaleconference.com/videos/networking-scale-2018-layer-4-load-balancing-at-facebook/

February 2018, BPF comes through firewalls
https://lwn.net/Articles/747551/
https://lwn.net/Articles/747504/
https://www.netronome.com/blog/frnog-30-faster-networking-la-francaise/

NETRONOME

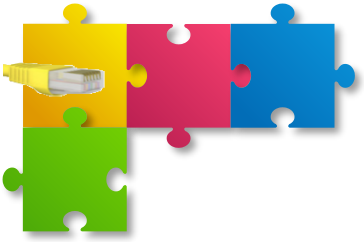
redhat

March 2018, Introducing AF_XDP support (to bring packets from NIC driver directly to userspace)
https://lwn.net/Articles/750293/
http://mails.dpdk.org/archives/dev/2018-March/092164.html
https://twitter.com/DPDKProject/status/1004020084308836357

DPDK
DATA PLANE DEVELOPMENT KIT

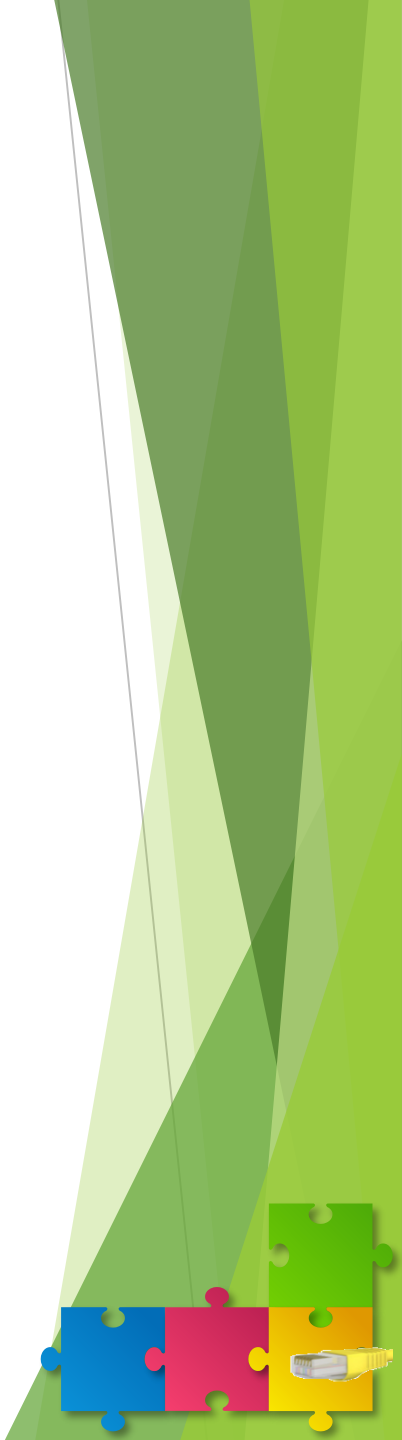
CUMULUS

April 2018, Add examples of ipv4 and ipv6 forwarding in XDP (to exploit the Linux routing table to forward packets in eBPF)
https://patchwork.ozlabs.org/patch/904674/



Introduction to eBPF

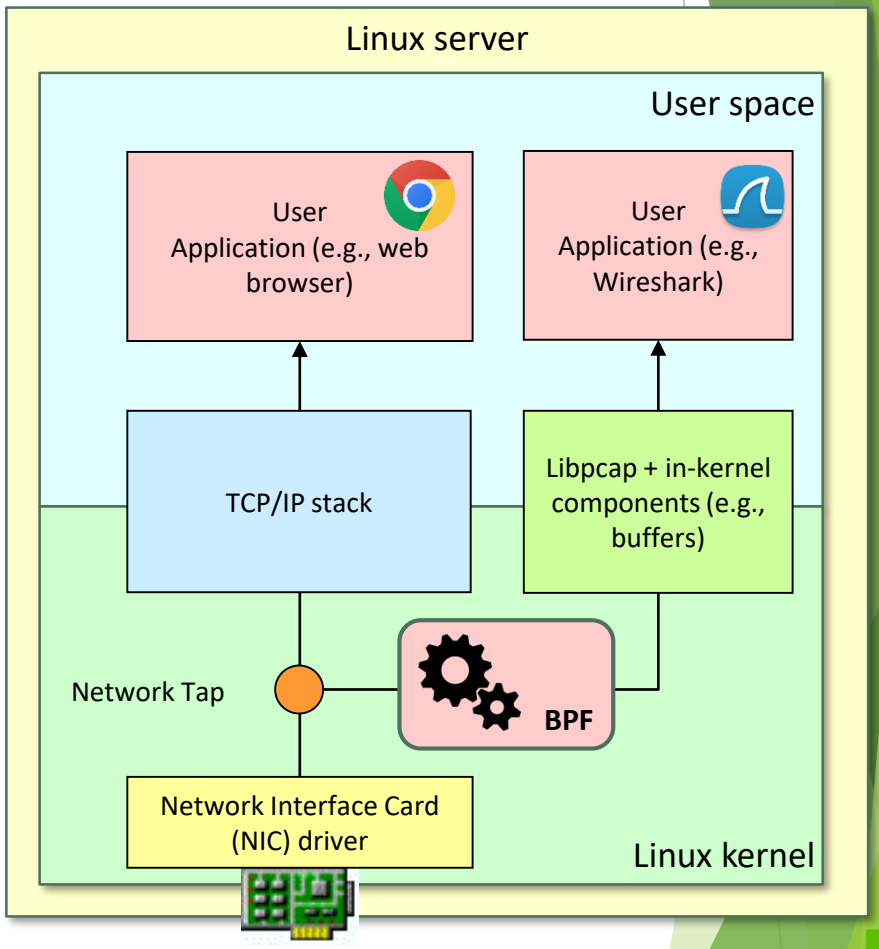
Part I



eBPF

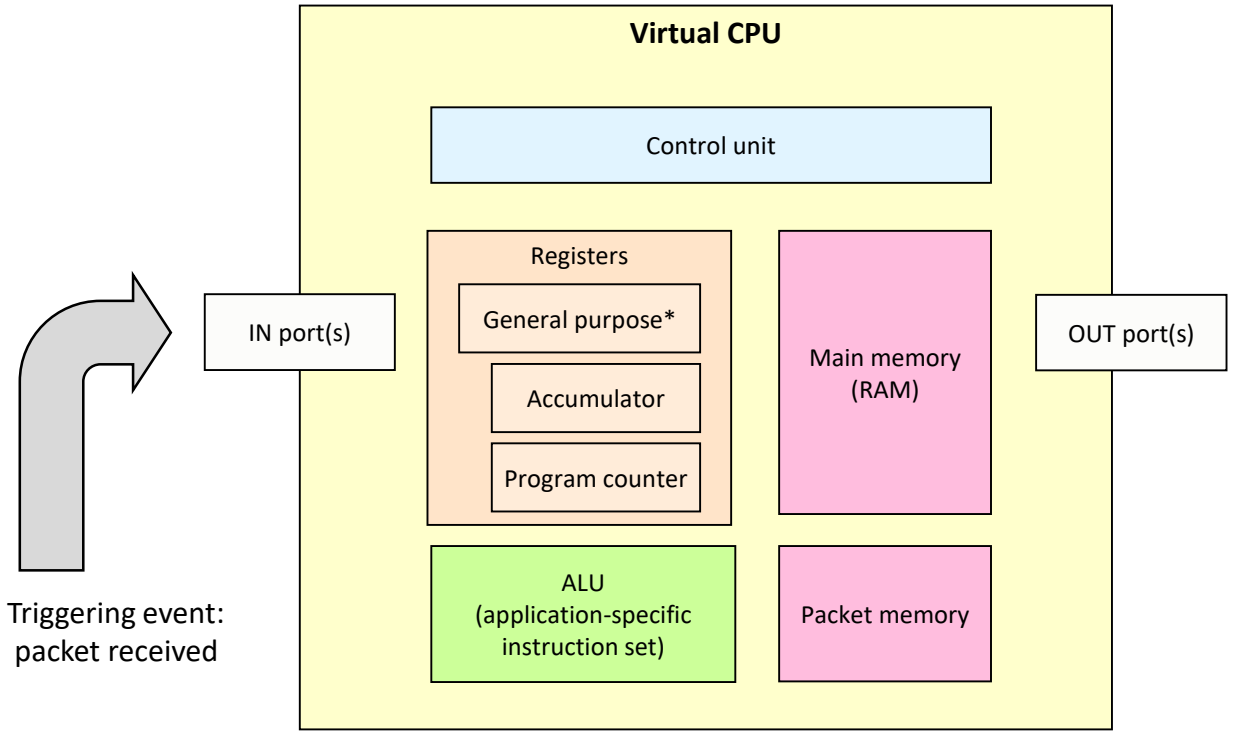
Berkeley Packet Filter (BPF)

- Generic **in-kernel**, **event-based virtual CPU**
  - Introduced in Linux kernel 2.1.75 (1997)
  - Initially used as packet filter by packet capture tool tcpdump (via libpcap)
- In-kernel
  - No syscalls overhead, kernel/user context switching
  - Process as soon as the event comes (e.g., early packets discard in DDOS)
- Event-based
  - Network packets
- Virtual CPU
  - Sandbox



Special purpose Virtual CPU

- An-hoc execution environment specially crafted for packet filtering purposes
  - E.g., specific memory for packets (separated from the main RAM)




```
Example of BPF injected code

Filter: "ip" (with simple Ethernet frames)

(000) ldh   [12]
(001) jeq   #0x800   jt 2   jf 3
(002) ret   #96
(003) ret   #0
```

* Since this is a virtual (not real) CPU, we can have either **infinite registers**, or a **stack-based** architecture (operands are pushed on the stack before calling the operator), such as the Java Virtual Machine, which is easier to implement.
The actual implementation has to take care of mapping the infinite number of registers to the existing ones (*register allocation*), which is one of the steps of modern compilers.
Please note also that the **stack pointer** is missing in this picture. On some embedded-oriented architectures, functions calls are not supported, hence the stack pointer may not be present.

Virtual CPU vs. Virtual Machine

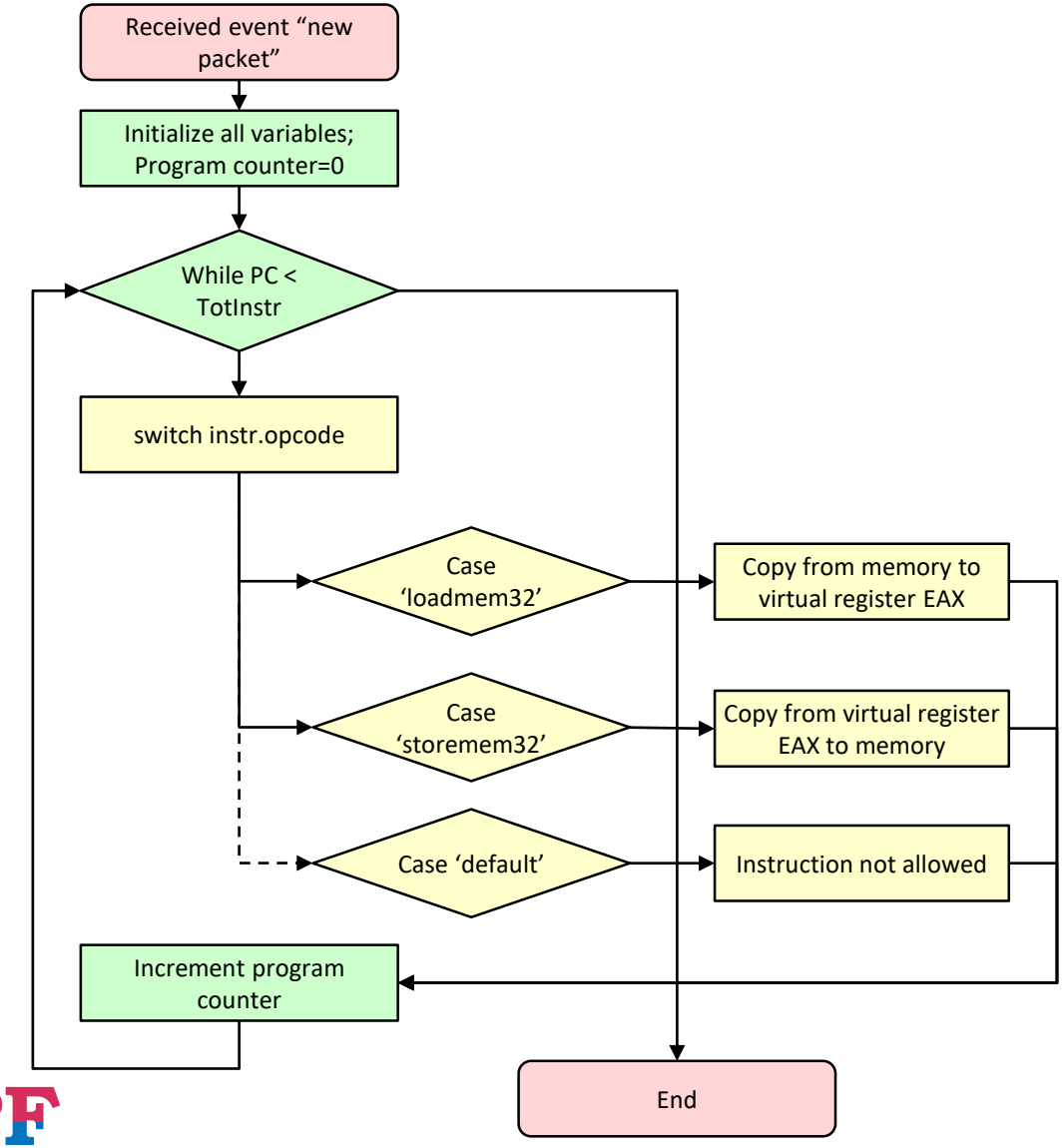
Special purpose vCPU

- Software architecture that emulates a specific HW component (e.g., CPU)
- Much easier to emulate
  - Just the HW, no need to support unmodified Operating Systems
  - Actually, several types of implementation are possible
- vCPU are also known a “Virtual Machines”, although this may be confusing given the current use of VMs in computing virtualization
  - Special purpose “VMs” are oriented to solve a specific problem (e.g., packet filtering)

Full-fledged VM

- Software architecture that emulates a full-fledged HW (e.g., CPU, memory, NICs, screen, I/O devices, etc.) and that is designed to virtualize a full computing system, starting with the OS
- Several HW to be emulated at high speed
  - Need to support un-modified Operating Systems, according to the full virtualization model

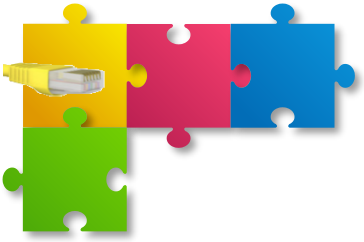
vCPU interpreter and BPF asm code examples




```
// Example of BPF programs
user@linux$ tcpdump -d ip
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 3
(002) ret      #96
(003) ret      #0

user@linux$ tcpdump -d ip6
(000) ldh      [12]
(001) jeq      #0x86dd          jt 2    jf 3
(002) ret      #96
(003) ret      #0

user@linux$ tcpdump -d tcp
(000) ldh      [12]
(001) jeq      #0x86dd          jt 2    jf 4
(002) ldb      [20]
(003) jeq      #0x6             jt 7    jf 8
(004) jeq      #0x800           jt 5    jf 8
(005) ldb      [23]
(006) jeq      #0x6             jt 7    jf 8
(007) ret      #96
(008) ret      #0
```

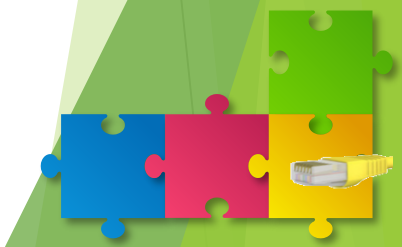
Extended Berkeley Packet Filter (eBPF)

- Initially proposed by Alexei Starovoitov (PLUMgrid) in 2013
  - https://lkml.org/lkml/2013/12/2/1066
- eBPF officially part of the Linux kernel since 3.15
  - In practice, kernel 4.x are required to take advantages of the more advanced features
  - List of features with the associated kernel version:
    - https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md
  - Continuously evolving platform
- Naming:
  - Initially, new BPF identified with “eBPF”, and old BPF called “Classic BPF” or “cBPF”
  - Recently (2018), people tend to refer to this technology simply as “BPF”
    - This may cause confusion, as “cBPF” is available on many operating systems (Windows, BSD, MAC OS, etc), while “eBPF” is available only on Linux



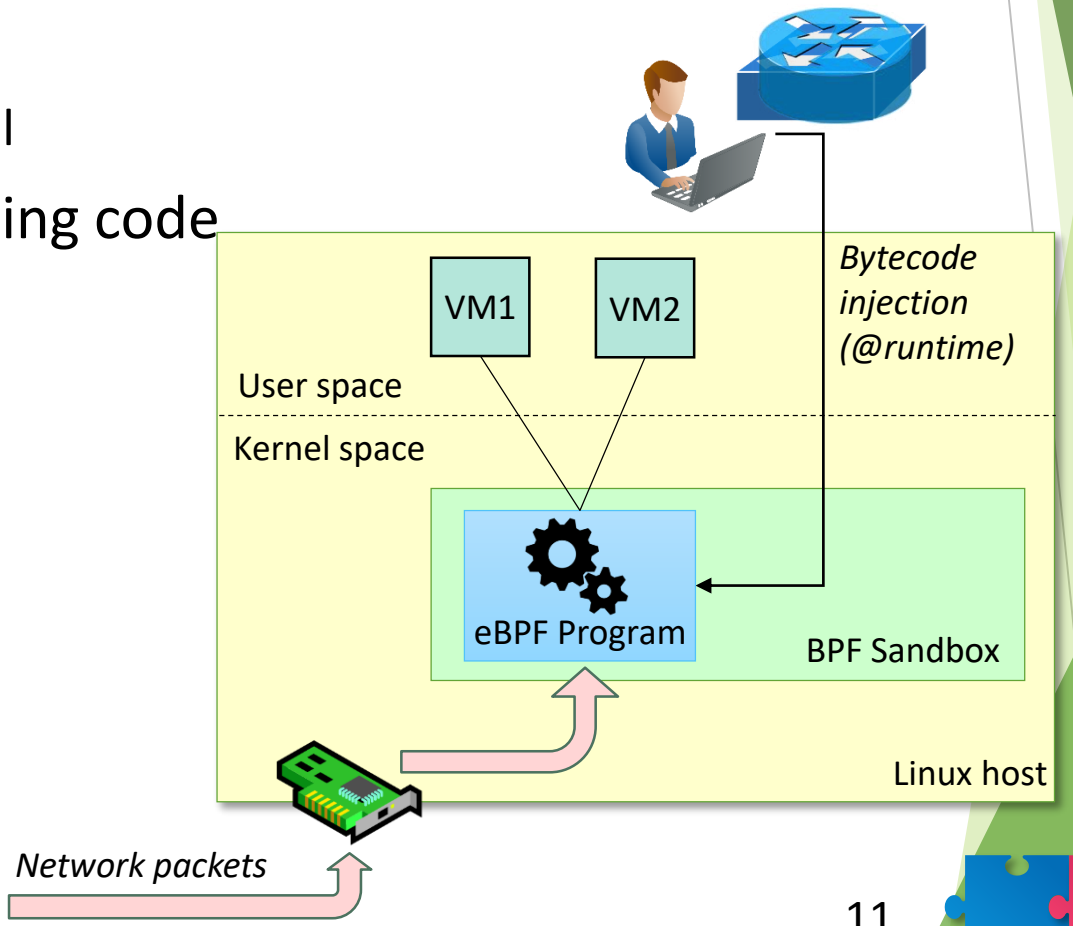
eBPF Key Features

Part II



#1: Runtime bytecode injection

- eBPF programs can be dynamically created and injected in the kernel at run-time
  - Vanilla Linux kernel, without any patch
  - No additional kernel module
  - Obviously, no need to recompile the kernel
- Enables dynamic fine tuning of processing code
  - E.g., “return all packets”, then
    “return ip packets”, then
    “return tcp packets on destport 80”, etc.



#2: Safe

- Linux kernel must be protected from erroneous or malicious injected programs
- Achieved with a **sandbox** (that prevents possible critical conditions at run-time) and a **verifier** (which checks the code and can refuse to inject it in the sandbox)
- Sandbox
  - No invalid memory access
- Verifier
  - Bounded program size
  - Bounded max number of instructions (in total, and per each possible execution path)
    - Hence, code cannot have unbounded loops (e.g., loops are unrolled)
- Consequence: eBPF does not support completely arbitrary programs
  - Not Turing complete

#3: Efficient

- BPF runtime consumes a little amount of resources
  - Cannot be used to generate a possible “denial of service” attack in the kernel because of its limited resource consumption
- BPF runtime is close to kernel events that have to be processed (e.g., packet received) – See the XDP technology later
- Assembly BPF bytecode is either interpreted or (in recent kernels) translated into native assembly code (e.g., x64) at run-time with a Just-in-time translator (JIT)
  - Interpreter has been removed starting from kernel 4.17

<u>1. Restricted C</u>

eBPF Code

```
static void init_array()
{
    int key;
    for (key = 0; key < 1000; key++) {
        bpf_update_elem(map_fd[0],
          &key, &value1, BPF_ANY);
    }
}
```

Compiler (userspace tool)

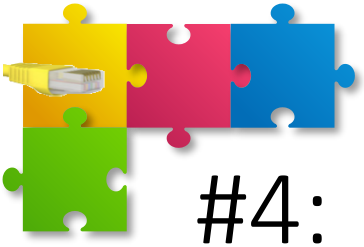
<u>2.eBPF bytecode</u>

```
l0: ldh [12]
l1: jeq #0x800, l3, l2
l2: jeq #0x805, l3, l8
l3: ld [26]
l4: jeq #SRC, l4, l8
l5: ld len
l6: jlt 0x400, l7, l8
l7: ret #0xffff
l8: ret #0
```

JIT (standard in-kernel feature)

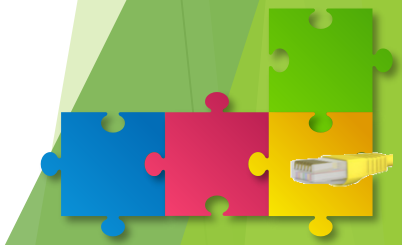
<u>3. x86 assembly</u>

```
mov eax, [ebp+8]
mov esi, [ebp+12]
mov edi, [ebp+16]

mov [ebp-4], edi
add [ebp-4], esi
add eax, [ebp-4]
```



#4: React to generic kernel events

- eBPF code is hooked to a kernel event
  - When fired, your code (associated to an *event handler*) is executed
- Some possible events:
  - Network packet received
  - Message (socket-layer) received
  - Data written to disk
  - Page fault in memory
  - File in /etc folder being modified
- In general, any kernel event can be potentially intercepted
  - Kprobes, Uprobes, syscalls, tracepoints



Caveats and limitations

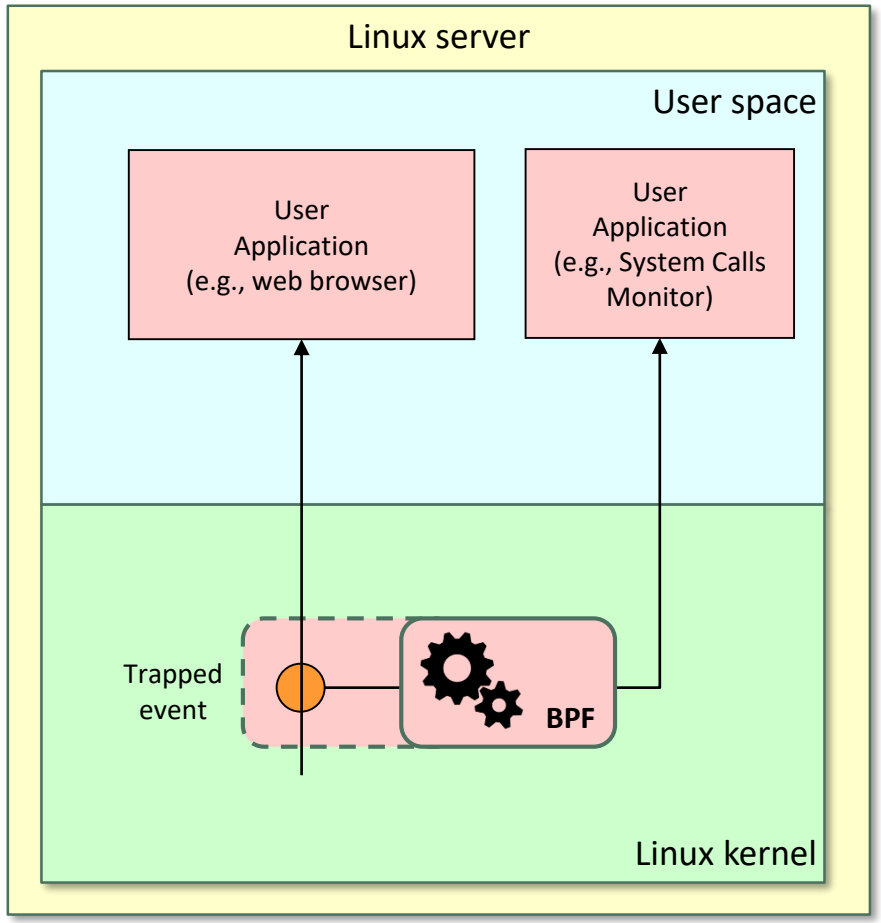
- Both syntax and semantic of the *eBPF_event_handler()* are event-dependent

- Parameters (a.k.a. metadata) delivered to the *eBPF_event_handler()* may be different based on the hook point as well
  - E.g., when operating on network packets, in some cases the programmer has the *skb*, in other cases he doesn't

- The actions allowed on the packet may be different (next slide)
  - In some cases the handler receives a *copy* of the event
  - In other cases it receives the *original* event, hence enabling to *modify* the event itself (e.g., change the content of the network packet)

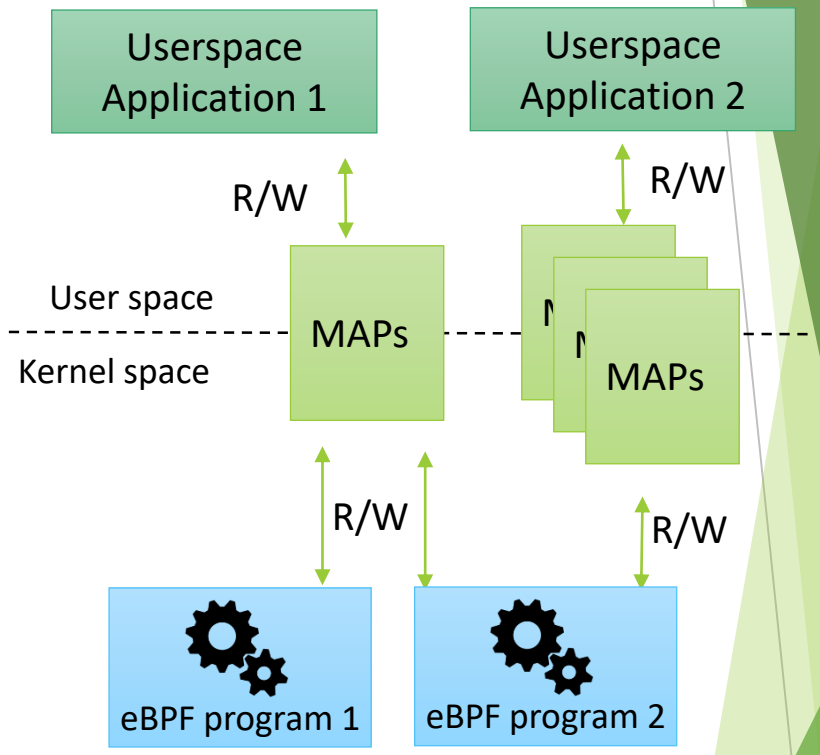
#5: Intercepting events: On-path vs. aside

- Depending on the type of event occurring
  - The **actual** event can be delivered to eBPF, which can potentially manipulate it (e.g., modify network packets)
  - A **copy** of the event can be delivered to eBPF, but the original event will continue its way in the kernel (e.g., write to disk), completely unmodidied



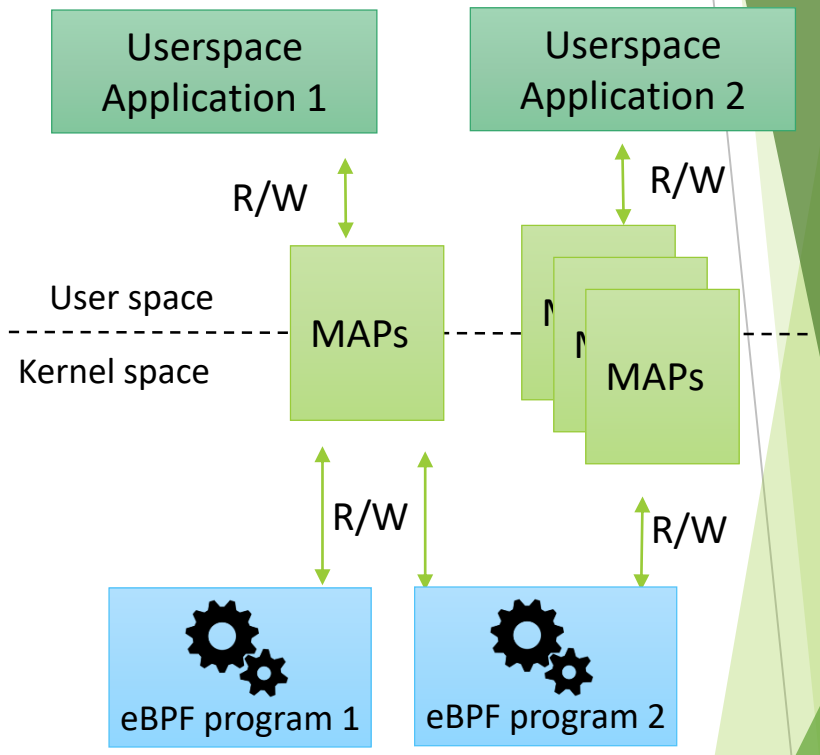
#6: Shared memory

- Shared between
  - User and kernel space
  - Different eBPF programs
- Three main purposes
  - Export data from kernel to userspace
    - E.g., kernel updates statistics about network traffic, userland program can (periodically) show them
  - Data pushed by userspace to kernel
    - E.g., user application configures eBPF program behavior (e.g., routing table)
  - Data shared between different eBPF programs
    - E.g., packet parser, sharing pointers to relevant protocol fields to all following modules



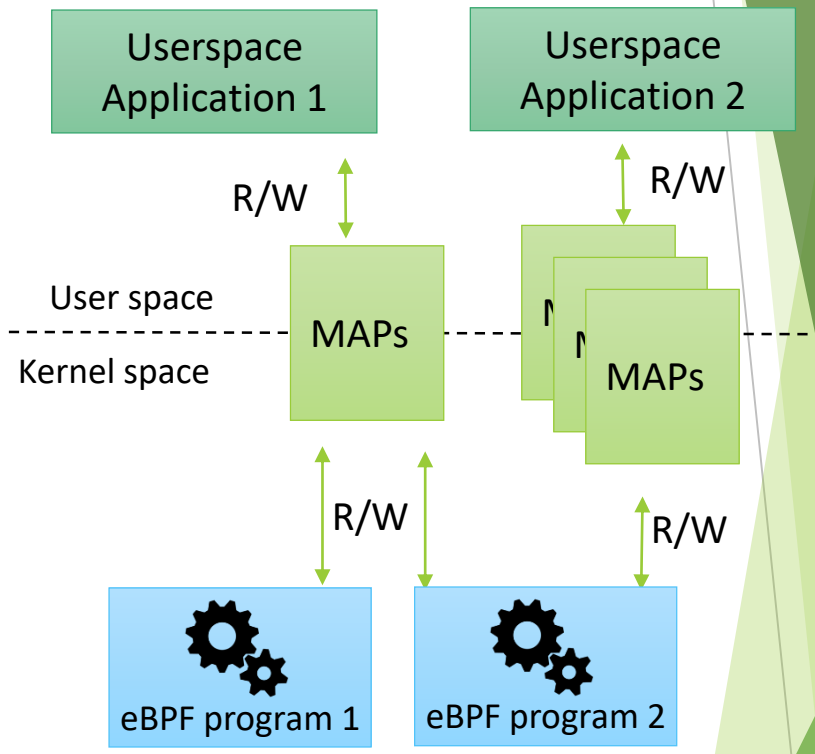
Shared memory: architecture

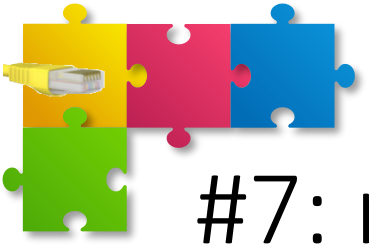
- Preformatted memory, instead of unstructured vanilla memory page
  - Data access arbitrated by proper structures, called **maps**
    - Key/value storage of different types
    - Array, HashMap, LRUMap, …
  - Cannot access to a specific offset in a map
  - Avoids concurrency issues when multiple entities are trying to access the same data



Shared memory: additional considerations

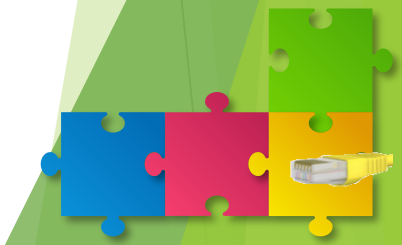
- **eBPF decouples the state from the code**
  - eBPF bytecode is stateless
  - Persistent data is saved in maps, which contain the state/configuration of a program
- Efficient (no copy needed)
- Per-CPU maps
  - Multiple instances of the same map, private of each CPU
  - Much more efficient (avoids data syncronization between CPU caches) but needs manual sync (if required)

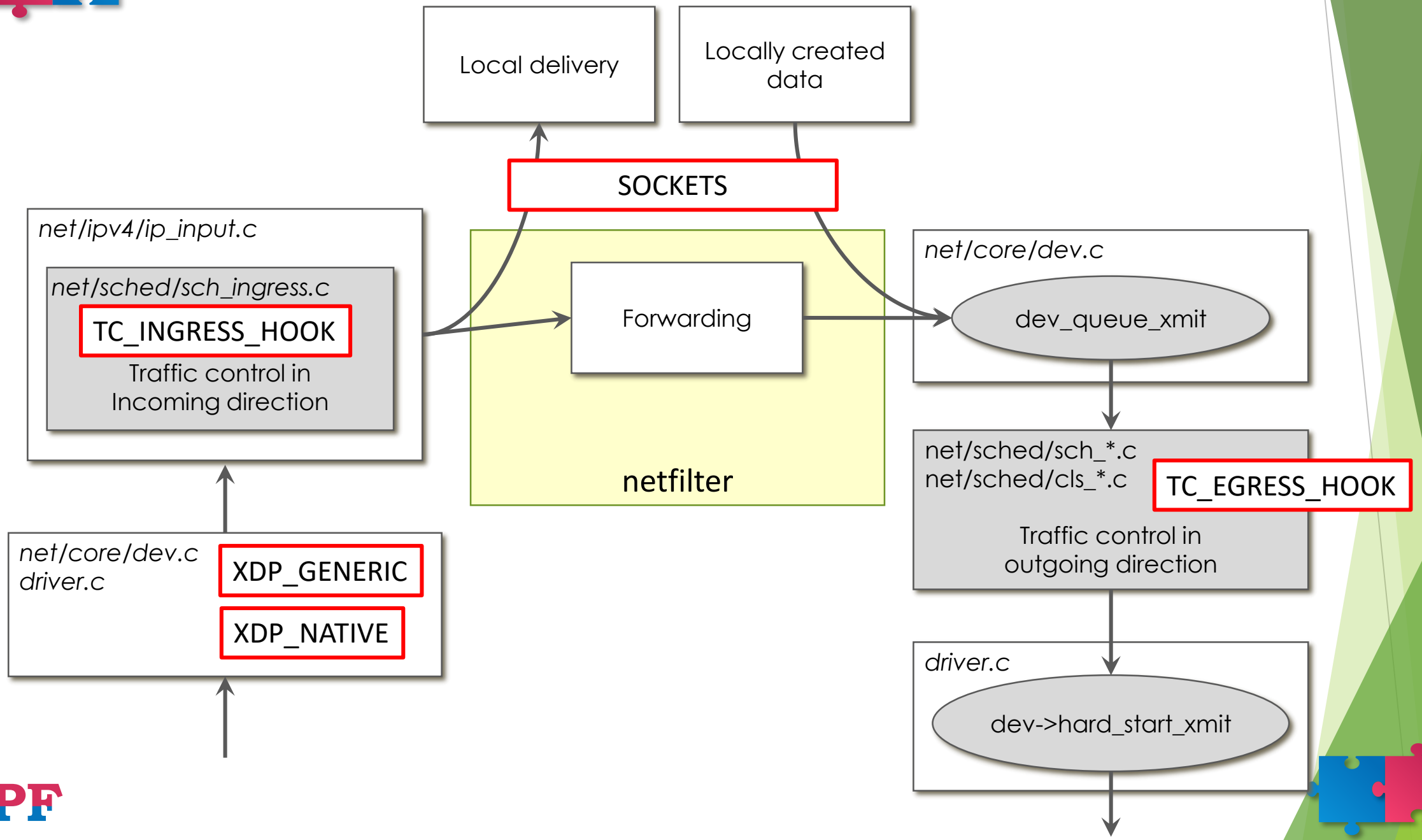




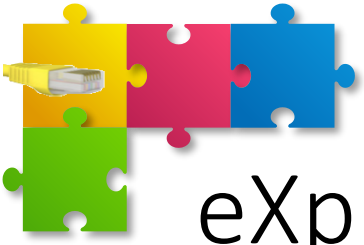
#7: multiple hook points for networking

- Several hook points (a.k.a. *kernel events*) for networking are available
  - Located at different levels of the Linux networking stack
  - Provide the ability to act on traffic that has or has not been processed already by other pieces of the stack
    - Opens up the possibility to implement network functions at different layers of the stack
  - Sockets
  - Traffic Control
  - RAW (traditional BPF)
- Metadata associated to the packet (and dispatched to the eBPF program) and allowed actions change according to the hook point used




Local delivery
Locally created data
SOCKETS
net/ipv4/ip_input.c
net/sched/sch_ingress.c
TC_INGRESS_HOOK
Traffic control in Incoming direction
Forwarding
netfilter
net/core/dev.c
dev_queue_xmit
net/sched/sch_*.c
net/sched/cls_*.c
TC_EGRESS_HOOK
Traffic control in outgoing direction
driver.c
dev->hard_start_xmit
net/core/dev.c
driver.c
XDP_GENERIC
XDP_NATIVE

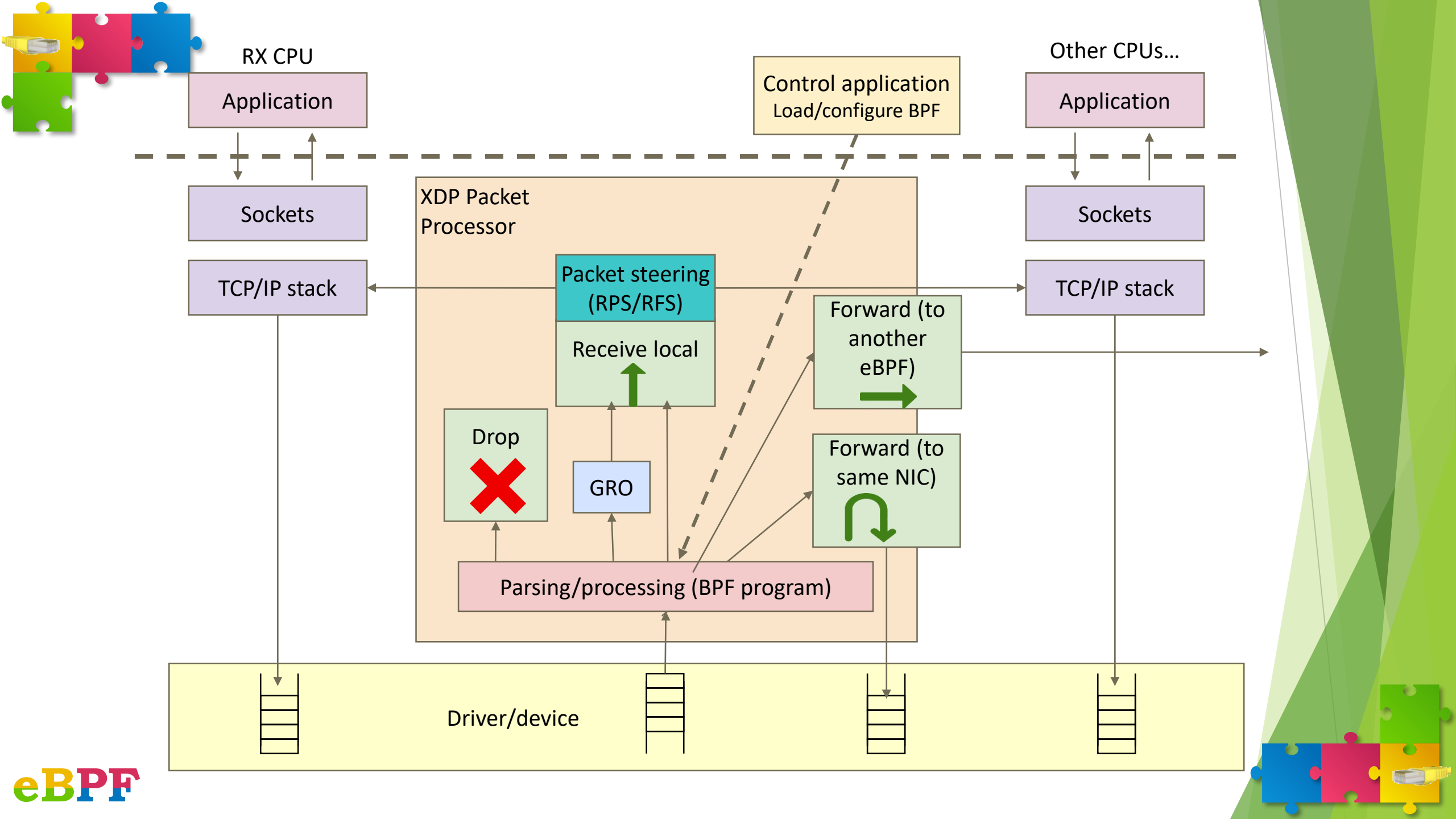
eBPF



eXpress Data Path (XDP)

- High performance hook point
  - Allows to run eBPF programs at the earliest networking driver stage (e.g., ixgbe [1])
- Possible use cases
  - Early packet discard (e.g., DDoS mitigation)
  - Firewalling
  - Load balancing
  - Forwarding

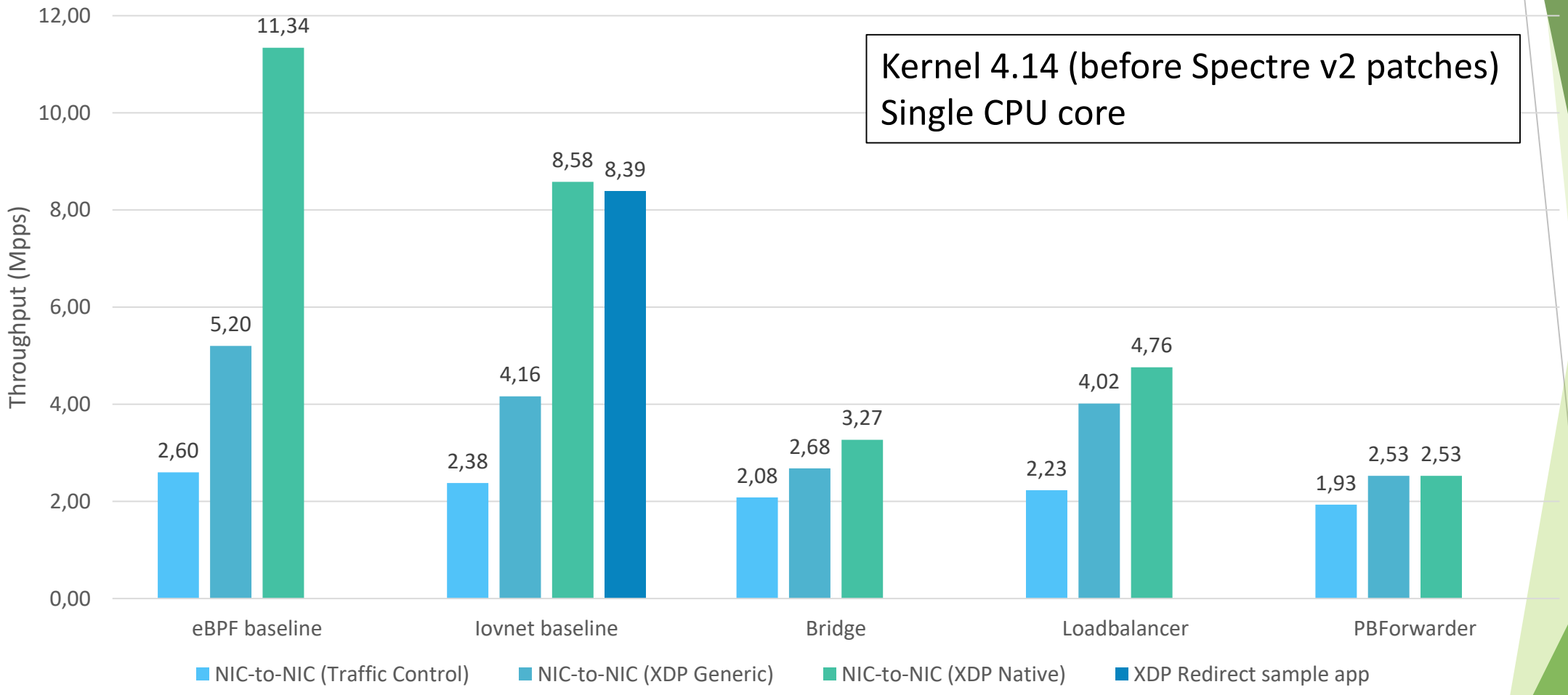
[1] https://github.com/torvalds/linux/blob/80cee03bf1d626db0278271b505d7f5febb37bba/drivers/net/ethernet/intel/ixgbe/ixgbe_main.c#L2333


RX CPU
Application
Control application
Load/configure BPF
Other CPUs...
Application
Sockets
XDP Packet
Processor
Sockets
TCP/IP stack
Packet steering
(RPS/RFS)
TCP/IP stack
Receive local
Forward (to
another
eBPF)
Drop
GRO
Forward (to
same NIC)
Parsing/processing (BPF program)
Driver/device

XDP generic vs XDP native

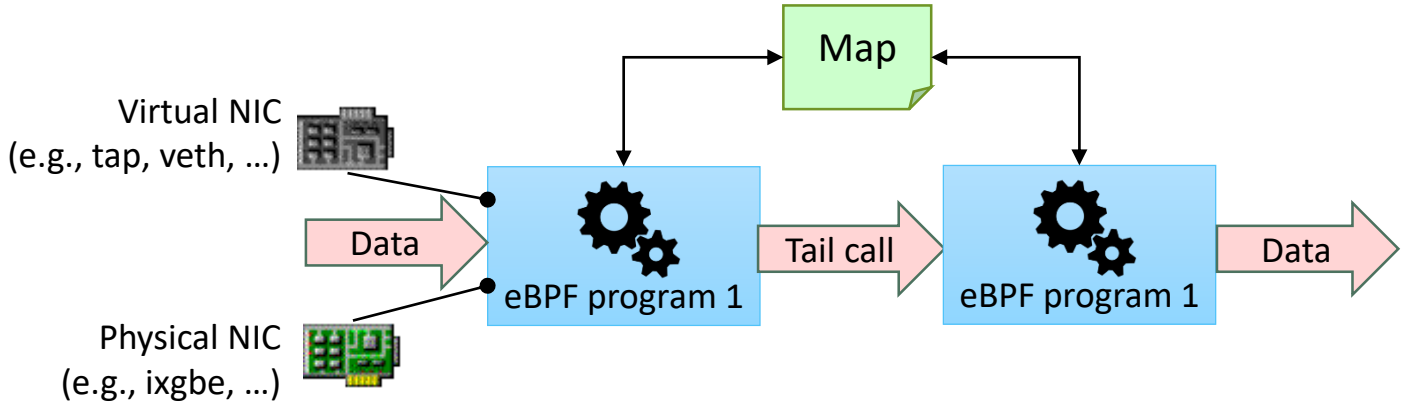
- Two possible working modes
  - XDP Native: XDP support provided in the network device driver; eBPF is called with “raw” packet, before Linux creates the *skb*
  - XDP Generic: offers XDP support for drivers that do not provide XDP native
    - User can request this mode by setting the XDP_FLAGS_SKB_MODE
    - A program attached to *XDP generic* is faster than TC, but slower than *XDP native*.
- A device that runs in XDP native can redirect a packet only to another device running XDP native
  - Redirect, in XDP native, sends the “raw” packet (without skb), so we need the other device to be XDP native as well
- A device that runs in XDP generic can redirect a packet to all the devices that run in XDP generic mode

XDP real apps tests



#8: Service chains

- eBPF programs can be connected to
  - Any physical/virtual NIC
  - Any tunnel adapter (e.g., GRE, IPsec, Vxlan, …)
- eBPF programs can be chained together to create complex service chains
  - Enable to split a complex function in multiple components
    - E.g., routing can be split in ARP handling, ICMP handling, forwarding process
  - Communication can happen through “tail calls”
    - Like functions calls (without return), which jump to other eBPF program
  - Although less efficient, eBPF programs can be connected through virtual adapters (e.g., veth) as well

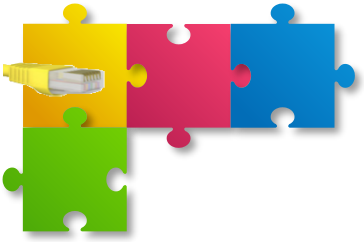


#9: Portable

- BPF bytecode is hardware independent, hence it can be executed on any architecture

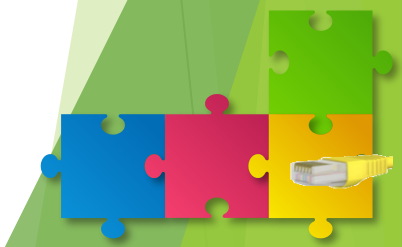
MIPS/ARM CPU
~128MB RAM
50-100 USD

Intel x64CPU
~64-128GB RAM
5000 USD



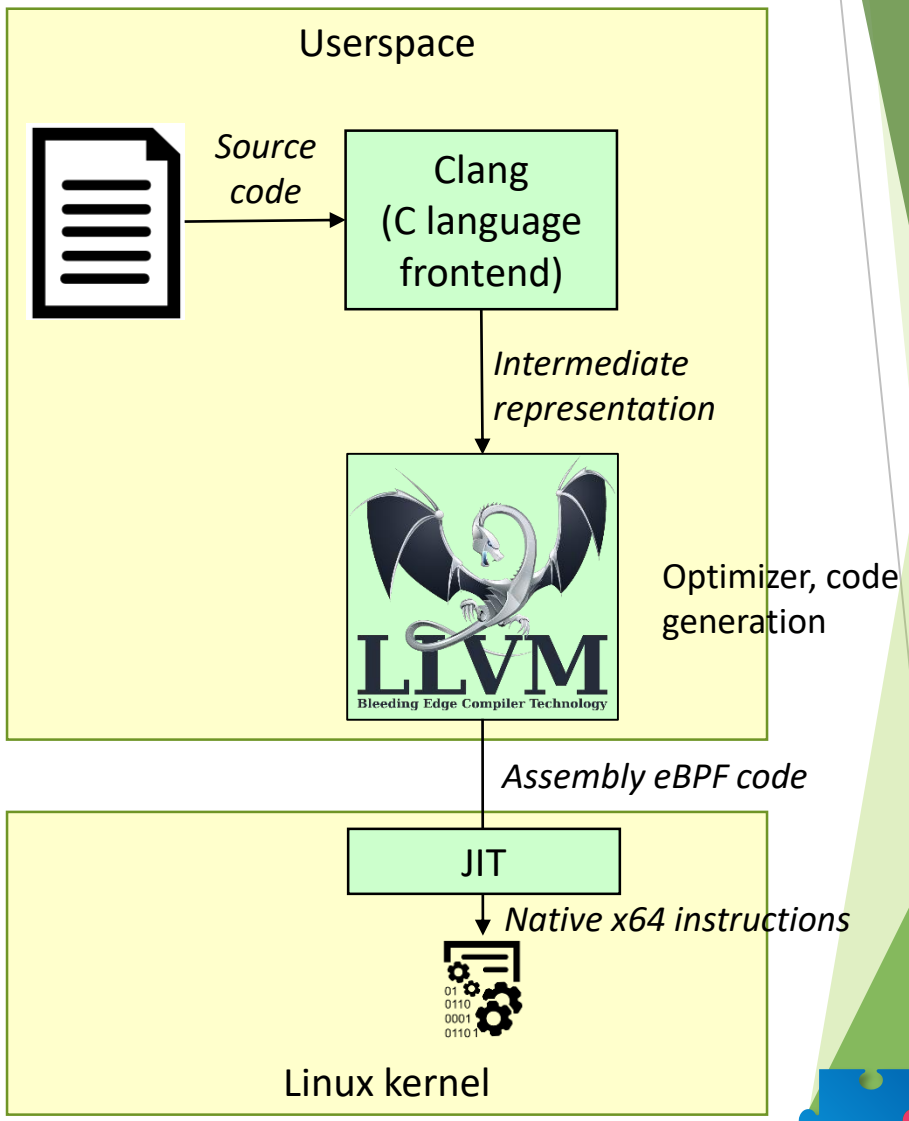
Creating eBPF Programs

Part III



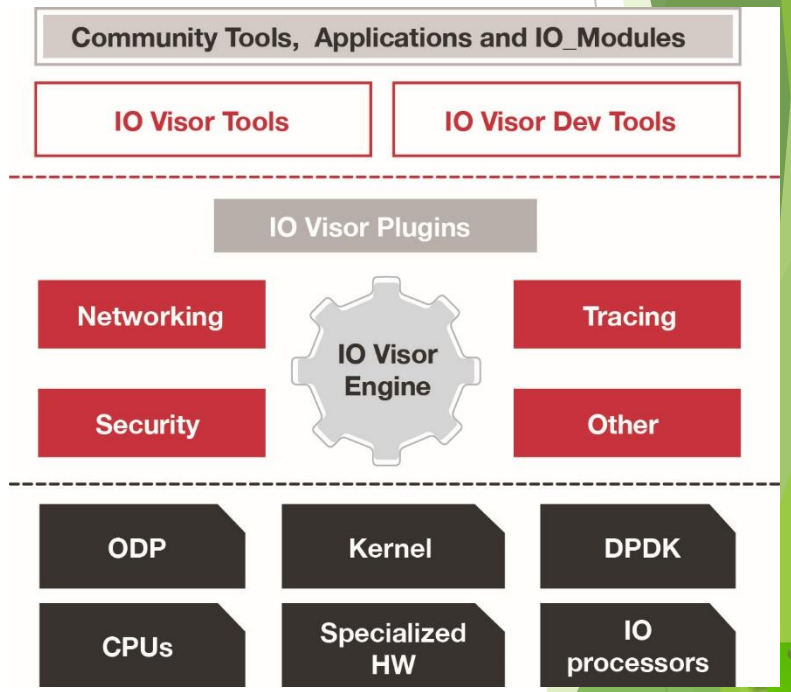
C-based programming

- eBPF code can be written in (restricted) C
  - Albeit with the limitations due to the verifier (safety)
- Extended the CLANG frontend compiler to accept the “restricted C” of the eBPF
- Leveraging the LLVM core for general optimizations
- Extended the LLVM backend to generate eBPF code
- Introduced a JIT (in Linux kernel) to translate eBPF assembly instructions into native x64 code



IOVisor

- IOVisor is (was?) a community driven open source project that opens up new ways to innovate, develop and share I/O and networking functions
- A collection of Open Source components that, combined together, allows to create IOModules that can be used to build networking, security, tracing applications
- It provides a programmable data plane and development tools to simplify the creation and sharing of dynamic “IO Modules”
- IOVisor stays to eBPF such as libpcap with BPF (and even more)



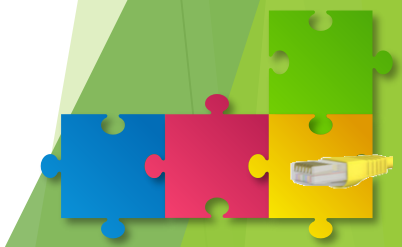
BCC – BPF Compiler Collection

- BCC is a toolkit for creating efficient kernel tracing and manipulation programs
- BCC makes eBPF programs easier to write, with kernel instrumentation in C and a front-end in Python and Lua
- It is suited for several tasks, including performance analysis, network traffic control and packet filtering
- Mostly used for tracing (monitoring)

Helpers

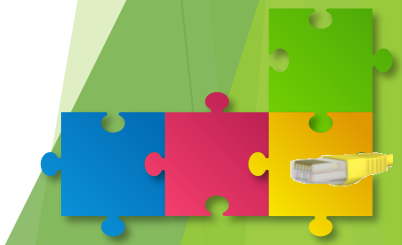
- eBPF assembly instructions are limited for safety reasons
  - We may need more instructions
  - We may need to perform some complex tasks, not allowed by the eBPF asm
- Possible solution: Helpers
  - Functions that are implemented natively in the Linux kernel, which are available (as an assembly call) from the bytecode
- A (long) list of helpers is available (and growing):
  - https://github.com/torvalds/linux/blob/master/tools/include/uapi/linux/bpf (from Linux kernel)
  - https://github.com/iovisor/bcc/blob/master/src/cc/export/helpers.h (from Iovisor project)
  - (they are obviously the same)

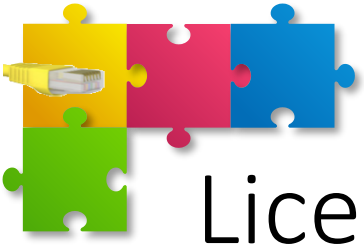
```
Example

// do some processing in assembly
(000) ...
(001) call bpf_ktime_get_ns()
(002) ...
```

Helpers and maps

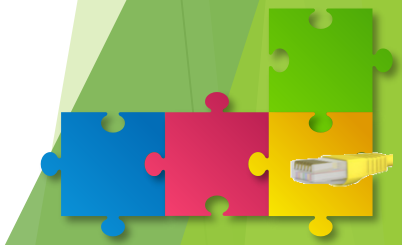
- Maps, in fact, are not made with “plain” memory, but are structured according to different types
  - E.g., Hash, Array, LongestPrefixMatch, etc.
  - https://github.com/torvalds/linux/blob/master/tools/include/uapi/linux/bpf.h
- Interation with map is possible through helpers, such as:
  - bpf_map_lookup_elem()
  - bpf_map_update_elem()
  - bpf_map_delete_elem()
- Helpers also arbitrate access to maps (i.e., mutual exclusion)





Licensing

- Kernel code in Linux has to be GPL
- Injected eBPF code may be non-GPL
  - Injected code require the explicit indication of the license
  - Some kernel helpers accept only GPL code, refusing to serve non-GPL programs
- Given the above limitation, eBPF enables non-GPL code to be executed at kernel level

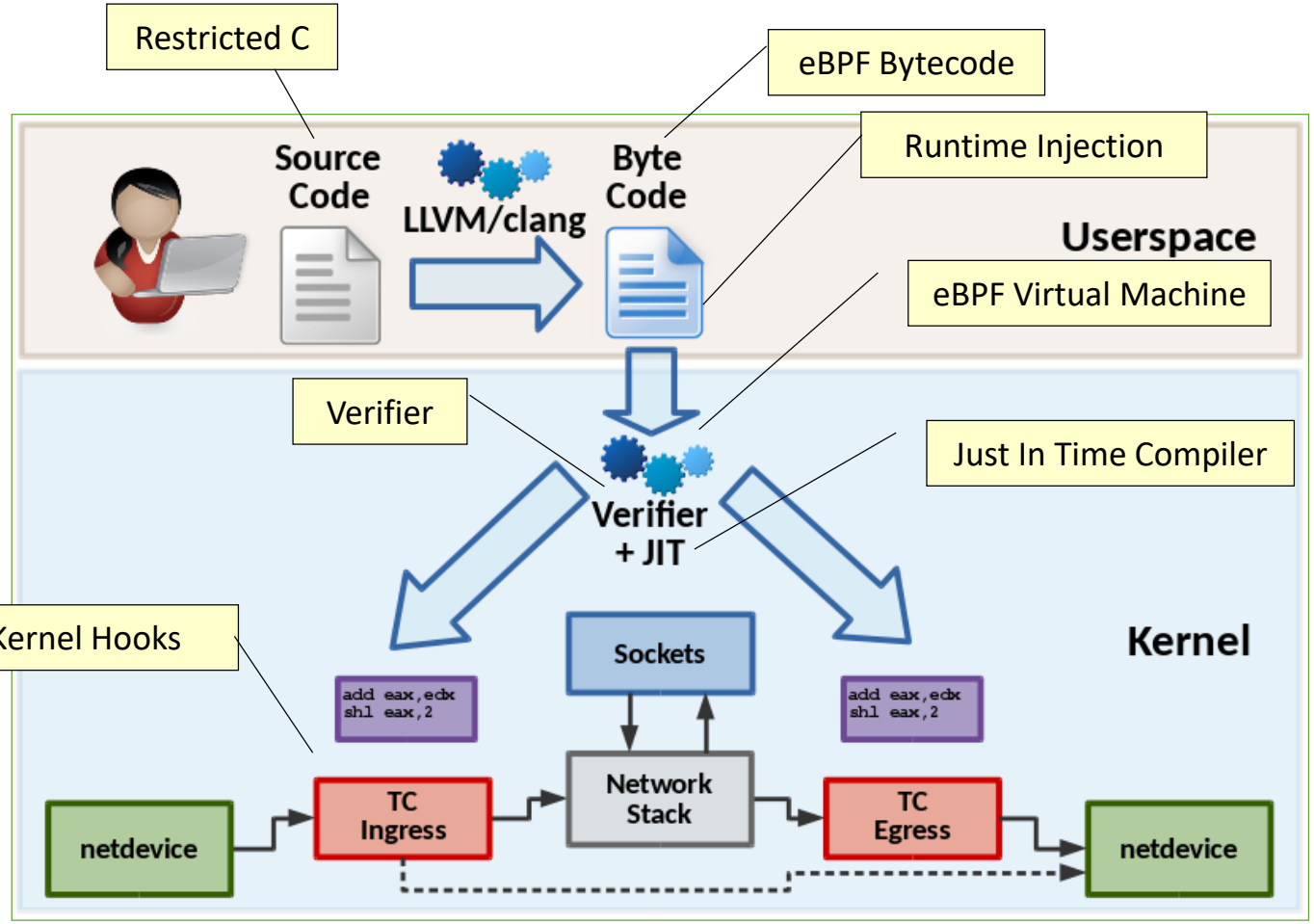


eBPF: overview of the runtime architecture

1. Restricted C

eBPF Code

```
static void init_array()
{
   int key;
   for (key = 0; key < 1000; key++) {
      bpf_update_elem(map_fd[0], &key,
&value1, BPF_ANY);
   }
}
```

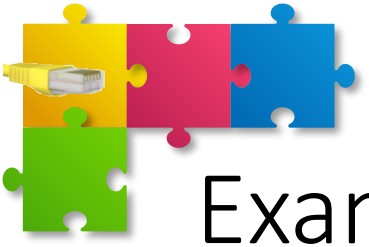


2.eBPF bytecode

```
l0: ldh [12]
l1: jeq #0x800, l3, l2
l2: jeq #0x805, l3, l8
l3: ld [26]
l4: jeq #SRC, l4, l8
l5: ld len
l6: jlt 0x400, l7, l8
l7: ret #0xffff
l8: ret #0
```

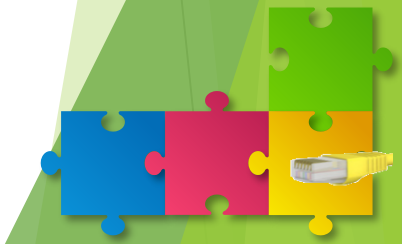
3. x86 Native Code

```
mov eax, [ebp+8]
mov esi, [ebp+12]
mov edi, [ebp+16]

mov [ebp-4], edi
add [ebp-4], esi
add eax, [ebp-4]
```



Example: eBPF program operating on sockets (1)

- High-level workflow
  - Create an eBPF program
  - Call new new bpf() syscall to inject the program in the kernel and obtain a reference to it
  - Attach the program to a socket (with the new SO_ATTACH_BPF setsockopt() option):
    - setsockopt(socket, SOL_SOCKET, SO_ATTACH_BPF, &fd, sizeof(fd));
    - where “socket” represents the socket of interest, and “fd” holds the file descriptor for the loaded eBPF program
- Once the program is loaded, it will be fired on each packet that shows up on the given socket
- Limitation: programs cannot do anything to influence the delivery or contents of the packet
  - These programs are not actually “filters”; all they can do is store information in eBPF “maps” for consumption by user space

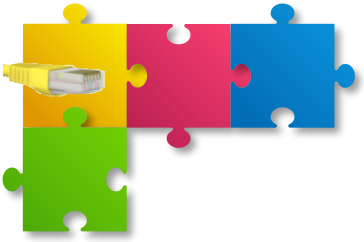


Example: eBPF program operating on sockets (2)

- This code obtains the low-level protocol (UDP, TCP, ICMP, …) from each packet and maintains a count for each protocol in an eBPF map

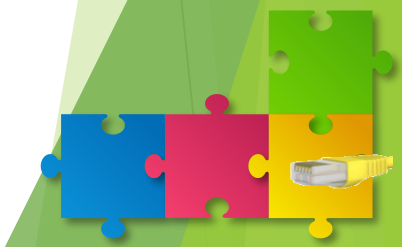
```
int bpf_rx_handler(struct sk_buff *skb)
    {
        /* 14 is the length of the Ethernet header; */
        /* 9 is the offset of the protocol type field in IP */
        /* No need to check if the packet contains IP or ARP or ... */
        /*   because we bind it to a socket */
        int index = load_byte(skb, 14 + 9);
        long *value;

        value = bpf_map_lookup_elem(&my_map, &index);
        if (value)
            __sync_fetch_and_add(value, 1);

        return 0;
    }
```

Piece of code that is translating into eBPF instructions and injected in the Linux kernel



eBPF: Gold, Iron, or Paper?

Part IV



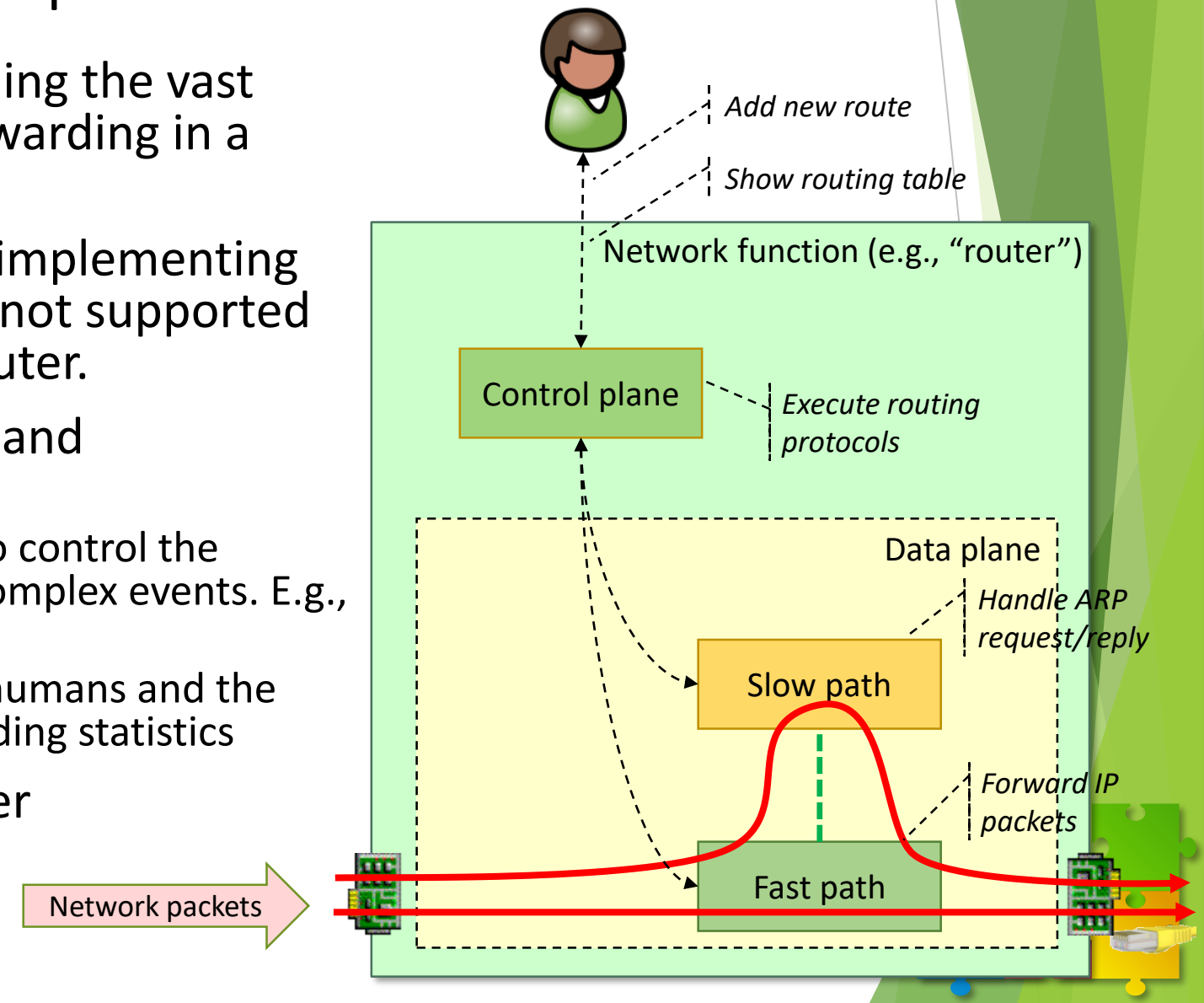
eBPF

The nature of the eBPF substrate

- **Gold**: we found a way to solve all our problems
- **Iron**: Nice concept, suitable to become the foundation of a bright technology, but need some more work
- **Paper**: Good for academia, not useful (or usable) in reality
- Let’s present some of the most important areas in which we need some additional work

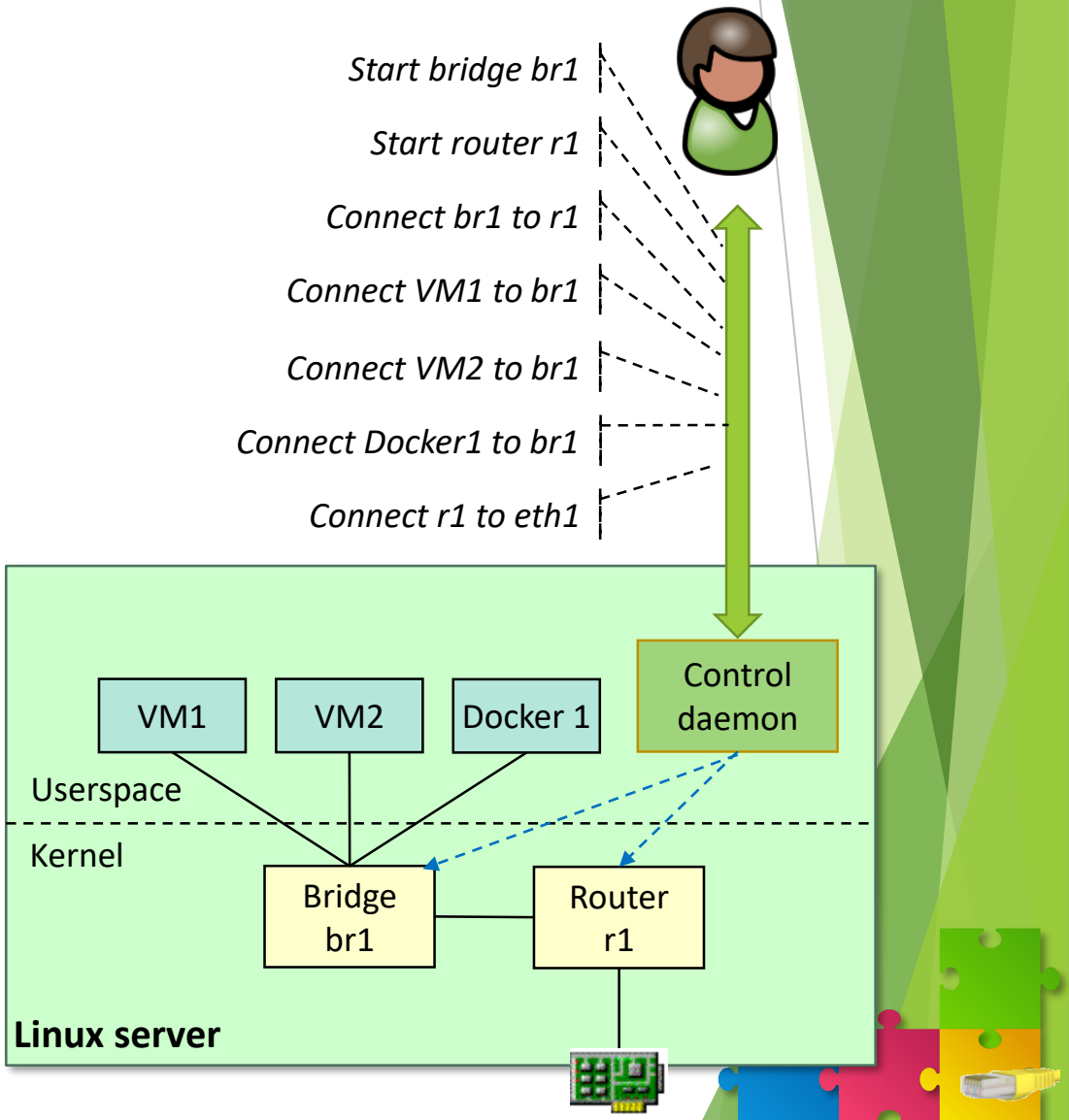
Fast, Slow, and Control path

- **Fast path**: running in kernel, handling the vast majority of packets. E.g., plain forwarding in a router.
- **Slow path**: running in user space, implementing complex data plane tasks that are not supported by eBPF. E.g., ARP handling in a router.
- **Control path**: implements **control** and **management** tasks
  - Control: out-of-band tasks needed to control the dataplane and to react to possible complex events. E.g., Routing Protocols, Spanning Tree
  - Management: interaction between humans and the software, e.g., for configuration, reading statistics
- Not necessarily on the same server



Single point of control and service chain

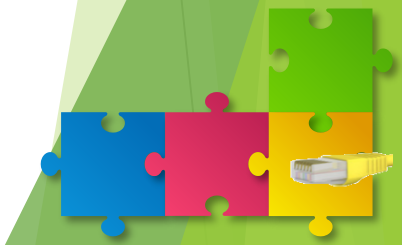
- Missing a single point of entry to the system
- Several reasons
  - Handle the lifecycle of each running service (and all virtual networks)
  - Facilitate the setup of the virtual infrastructure
    - Connects services together (and disconnect)
    - Connects/disconnects endpoints (veth, VMs, Docker, …) to/from services
  - Configures services
- Not easy to create a service chain
  - Technically is possible, in practice programmers have to deal with a lot of minor details
  - Interface name (and numbering)
  - Differences in XDP and TC programs…

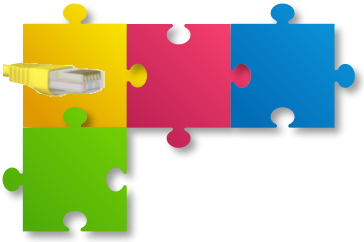


Other possible issues…

- … and how to possible solve them:

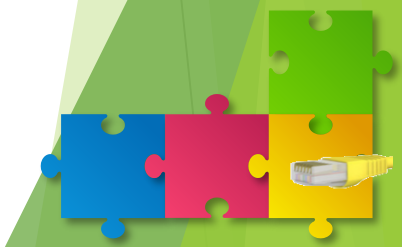
Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vasquez Bernal, Massimo Tumolo, “Creating Complex Network Services with eBPF: Experience and Lessons Learned,” Proceedings of IEEE International Conference on High Performance Switching and Routing, Bucharest, Romania, June 2018.

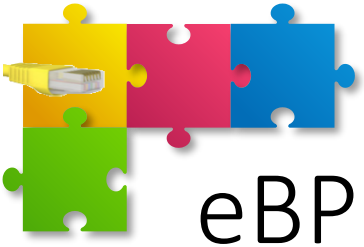




Additional Topics

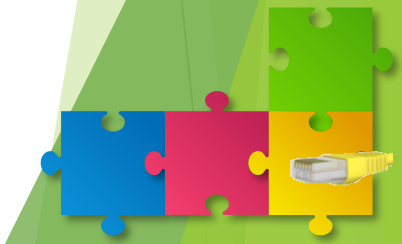
Part V





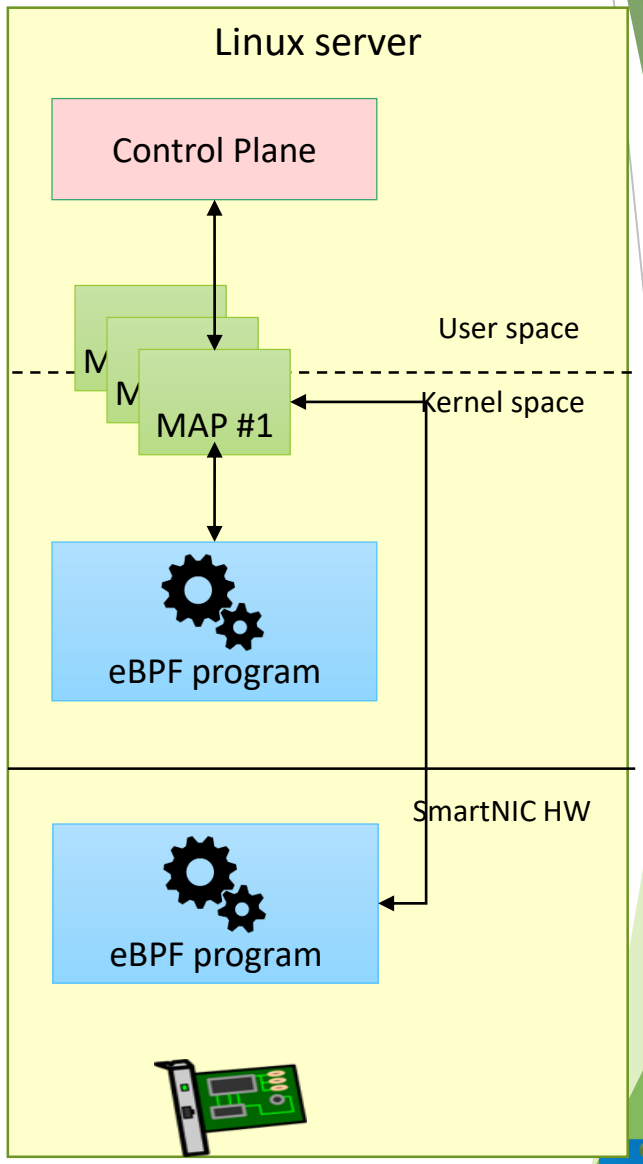
eBPF and P4

- P4: *protocol independent* packet processing architecture
  - eBPF does not say anything about “protocol independency”
- P4 is a high level programming language for packet forwarding dataplanes
  - It can be compiled for a different set of targets
  - Based on the concept of match and action
- P4 can be compiled to eBPF
  - https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4
- eBPF
  - Follows a “evolutionary” approach, supporting programs written in “restricted C”
  - Supports generic events, not just network packets
  - Completely software-based (no hardware implementations), hence limited to the edge of the network (Linux servers-based)

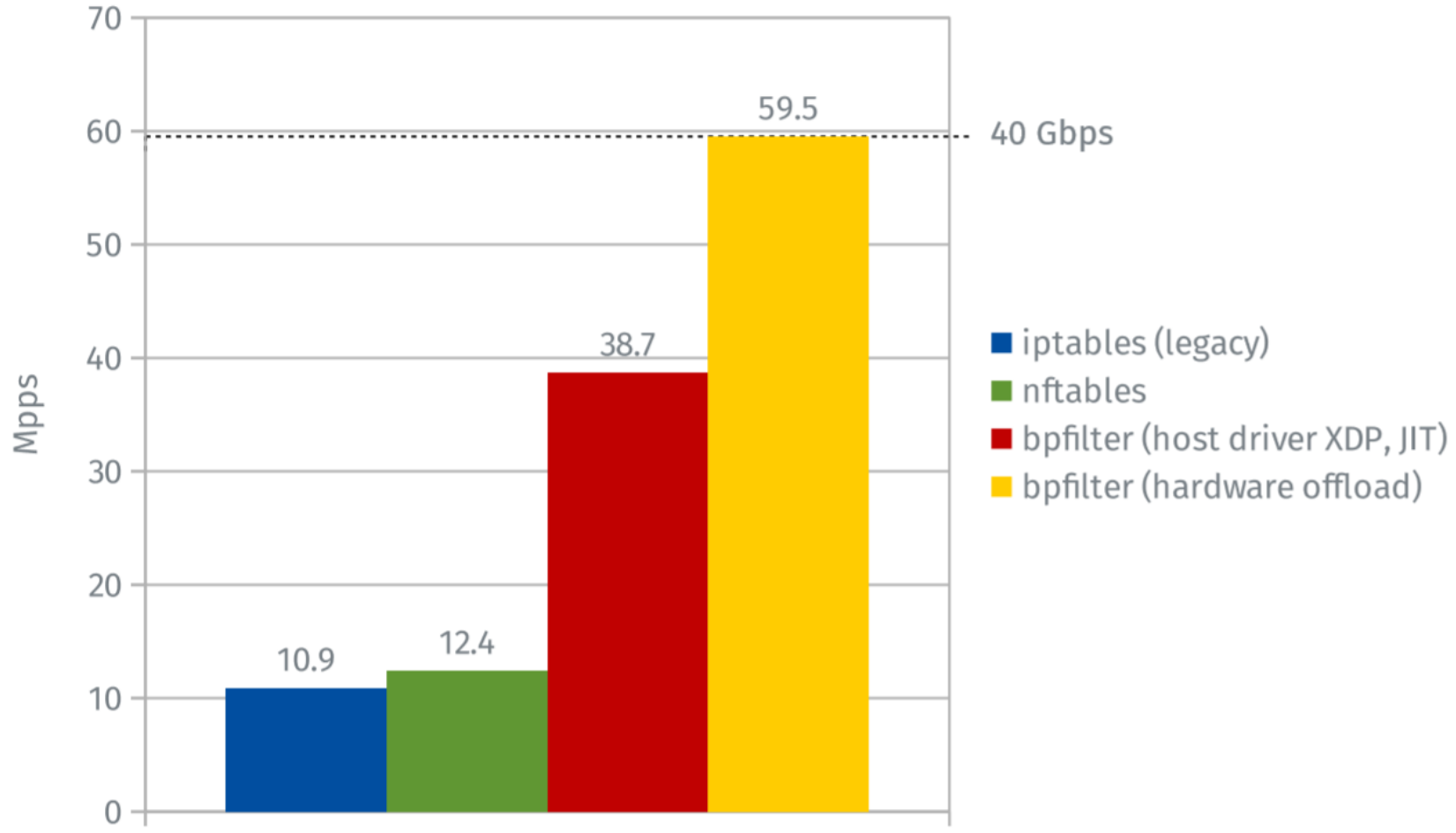


SmartNICs

- Typical architecture
  - General purpose CPU: can be used to offload eBPF (e.g., XDP) programs
  - Additional hardware acceletators: can be used to delegate specific tasks to the NIC
- Opportunities, but also challenges
  - On-board CPU usually slower than Intel chips
    - Can save server CPU cycles, but at the same time it can be slower than main CPU
  - Hardware accelerators are powerful, but how can we create eBPF programs that run seamlessly on both SmartNIC-equipped servers and vanilla machines?
  - Are maps (which must be shared between SmartNIC and userspace), introducing any performance penalty?



Performance: XDP vs SmartNIC



Quentin Monnet, FRnOG30, “bpfilter, pare-feu Linux à la sauce eBPF,” March 2018. https://www.qmo.fr/docs/talk_20180316_frnog_bpfilter.pdf

Is eBPF/XDP an alternative to DPDK?

eBPF/XDP

- Very fast (particularly coupled with XDP)
- Runs on vanilla kernels (no additional kernel modules, no kernel patches, etc.)
- Limited licensing issues
- Arbitrary processing only supported in “slow path” (userspace)

DPDK

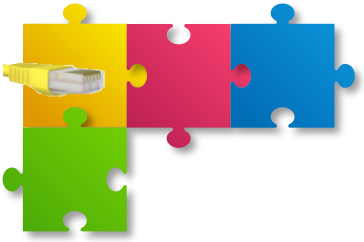
- Very fast
- Requires a modified system (additional kernel modules)
- No licensing issues
- Supports generic programming (Turing complete)
- Sophisticated and efficient libraries (e.g. for memory management, etc.)

March 2018, Introducing AF_XDP support
  Brings packets from NIC driver directly to userspace
https://lwn.net/Articles/750293/
http://mails.dpdk.org/archives/dev/2018-March/092164.html
https://twitter.com/DPDKProject/status/1004020084308836357

redhat

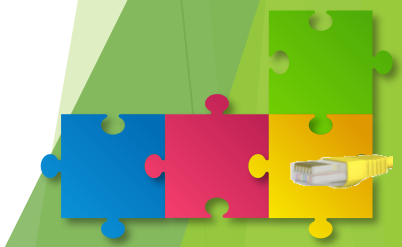
DPDK
DATA PLANE DEVELOPMENT KIT

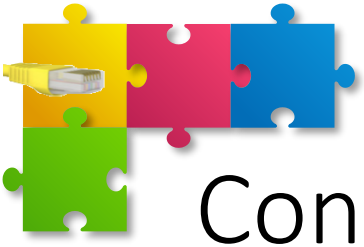
eBPF



Conclusions

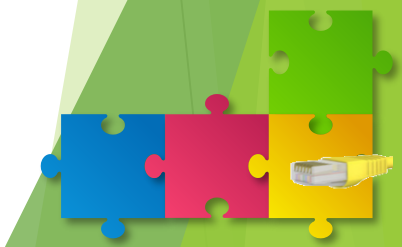
Part VI





Conclusions

- Fast (relatively) easy to use, potentially very powerful
  - Monitoring and (likely) network processing
- Many use cases
  - Packet filters (copy packet and pass to user space)
    - Used by tcpdump/libpcap, wireshark, nmap, dhcp, arpd, ...
  - In-kernel networking subsystems
    - cls_bpf (TC classifier) – QoS subsystem- , xt_bpf, ppp,...
  - seccomp (chrome sandboxing)
    - Introduced in 2012 to filter syscall arguments with bpf program
  - Tracing, Networking, Security, …
- Several “big names” here
- Need to enlarge the community, particularly with respect to end-users and application (e.g., non-kernel) developers



eBPF