Application-layer traffic processing with eBPF

extended Barkeley packet filter

Bertrone Matteo – Polytechnic of Turin

November 2015

BPF vs eBPF - *classical BPF*

- BPF - Berkeley Packet Filter
- Initially used as socket filter by packet capture tool tcpdump (via libpcap)
- Introduced in Linux in 1997 in kernel version 2.1.75

- Use cases:
- Mainly socket filters (drop or trim packet and pass to user space)
- used by tcpdump/libpcap, wireshark, nmap, dhcp ..

BPF vs eBPF - *extended BPF*

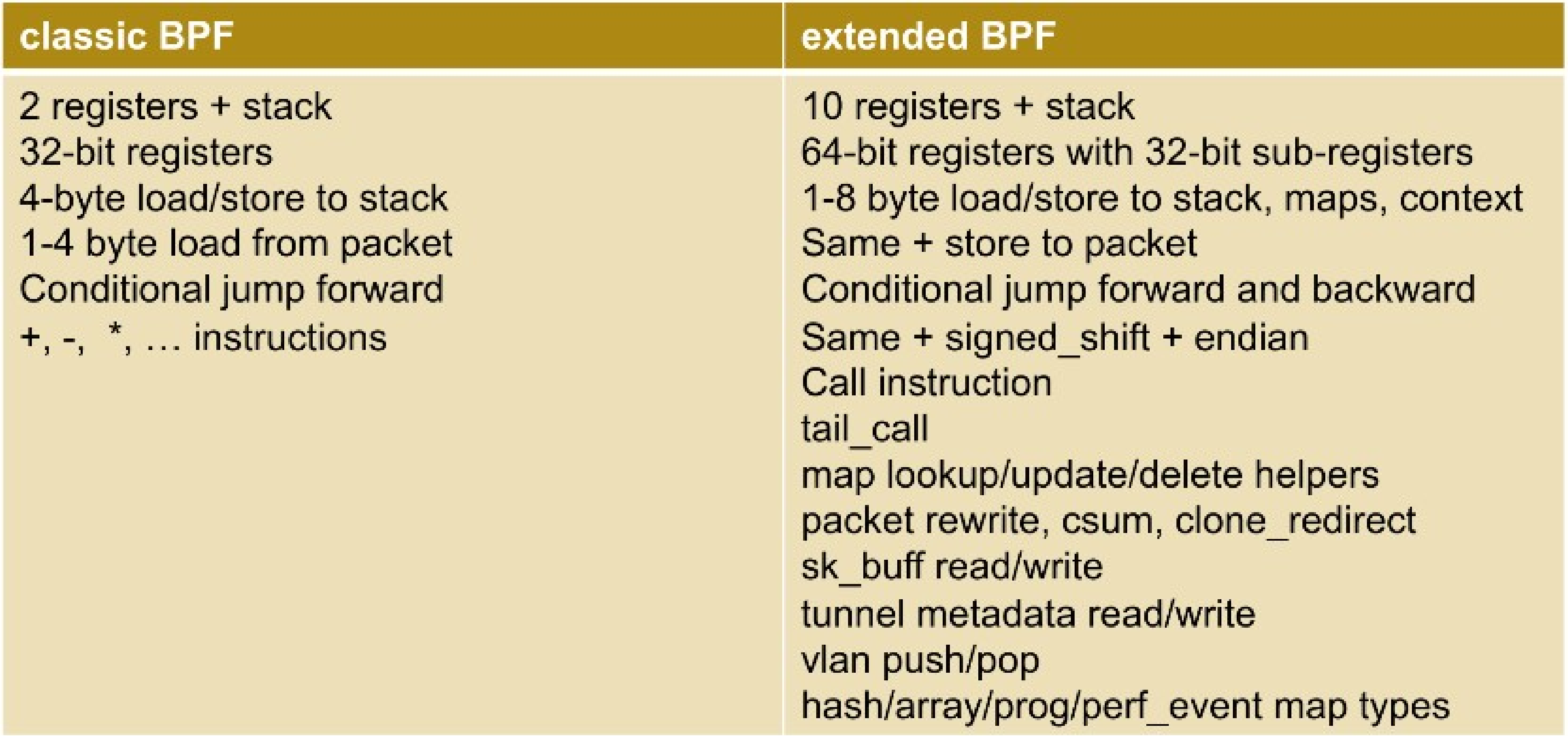
- New set of patches introduced in the Linux kernel since 3.15 (June 8th, 2014) and into 4.0 (April 12th, 2015) and into 4.1 and 4.3
- Current Linux kernel version 4.3.99 (November 2015)
- More registers (64 bit)
- In-kernel JIT compiler (safe) : x86, ARM64, s390 …
- “Universal in-kernel virtual machine”
- Portable – any platform that LLVM compiles into will work

- Use Cases:
- Networking (packet filtrering , network traffic control, etc ...)
- Tracing (analytics, monitoring, debugging)

Extended BPF

- Idea: improve and extend existing BPF infrastructure
- Programs can be written in C and translated into eBPF
- instructions using Clang/LLVM, loaded in kernel and executed
- LLVM backend available to compile eBPF programs (llvm 3.7)
- Safety checks performed by kernel
- Added arm64, arm, mips, powerpc, s390, sparc JITs
- ISA is close to x86-64 and arm64

eBPF - *low level VM architecture*



classic BPF	extended BPF
2 registers + stack	10 registers + stack
32-bit registers	64-bit registers with 32-bit sub-registers
4-byte load/store to stack	1-8 byte load/store to stack, maps, context
1-4 byte load from packet	Same + store to packet
Conditional jump forward	Conditional jump forward and backward
+, -, *, … instructions	Same + signed_shift + endian
	Call instruction
	tail_call
	map lookup/update/delete helpers
	packet rewrite, csum, clone_redirect
	sk_buff read/write
	tunnel metadata read/write
	vlan push/pop
	hash/array/prog/perf_event map types

eBPF new features - *maps*

- BPF maps are key/value storage of different types.
- Example
  value = bpf_table_lookup(table_id, key) – lookup key in a table

- Userspace can read/modify the tables
- Generic memory allocated
- Transfer data from userspace to kernel and vice versa
- Share data among many eBPF programs (see next)
- A map is identified by a file descriptor returned by a bpf() system call that creates the map
- Attributes: max elements, size of key, size of value
- Types of maps: BPF_MAP_TYPE_ARRAY, BPF_MAP_TYPE_HASH

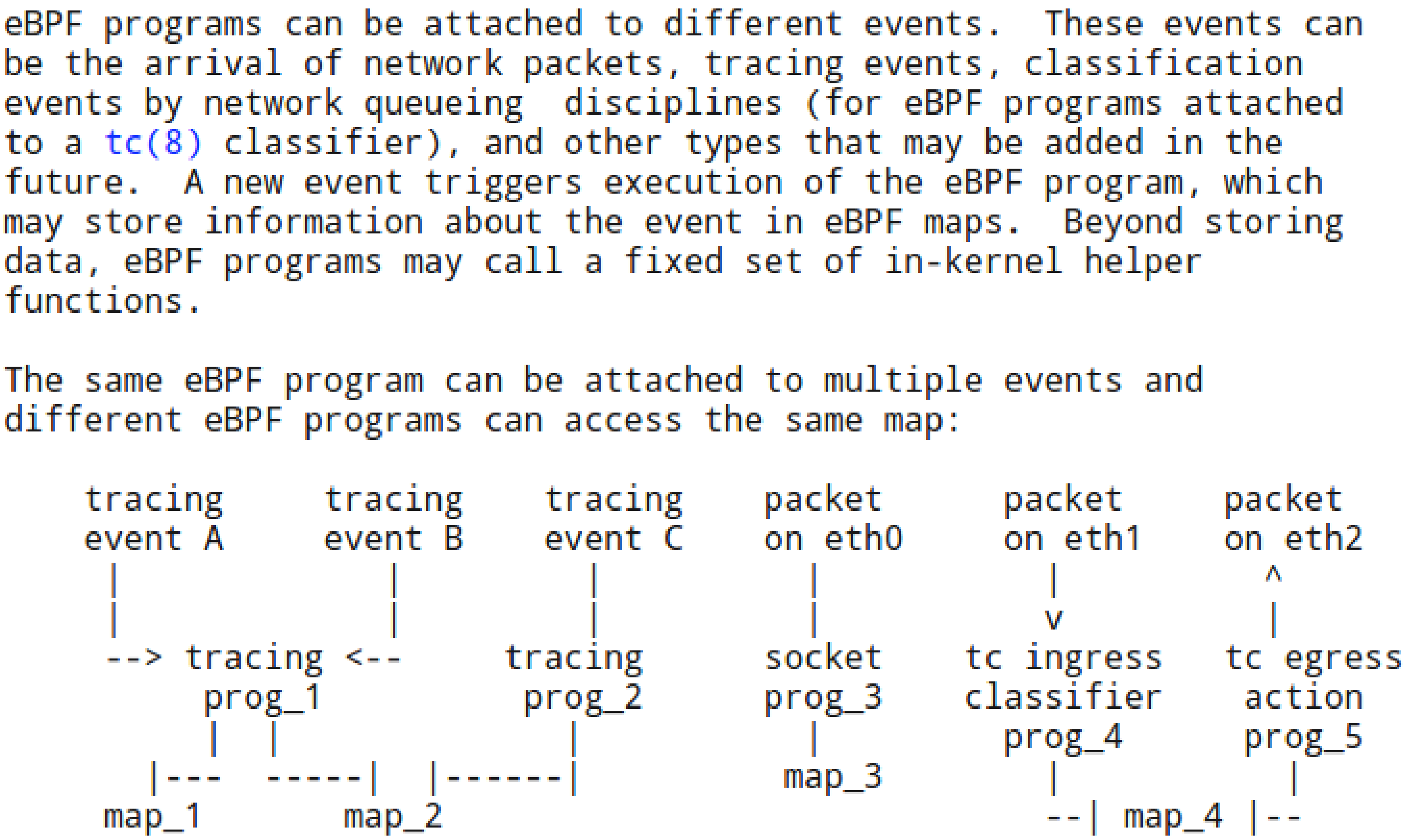
eBPF – *maps example*

- Restrictive C program to:
- obtain the protocol type (UDP, TCP, ICMP, …) from each packet
- keep a count for each protocol in a “map”:

```
int bpf_prog1(struct __sk_buff *skb)
 {
     int index = load_byte(skb, ETH_HLEN +
 offsetof(struct iphdr, protocol));                 //load in index protocol of current pkt
     long *value;

     value = bpf_map_lookup_elem(&my_map, &index);  //lookup in bpf map (my_map)
     if (value)
         __sync_fetch_and_add(value, 1);            //increment counter for current protocol
     return 0;
 }
```

eBPF – *maps sharing*



eBPF programs can be attached to different events. These events can be the arrival of network packets, tracing events, classification events by network queueing disciplines (for eBPF programs attached to a tc(8) classifier), and other types that may be added in the future. A new event triggers execution of the eBPF program, which may store information about the event in eBPF maps. Beyond storing data, eBPF programs may call a fixed set of in-kernel helper functions.

The same eBPF program can be attached to multiple events and different eBPF programs can access the same map:

```
    tracing     tracing    tracing    packet      packet     packet
    event A     event B    event C    on eth0     on eth1    on eth2
     |             |         |          |           |          ^
     |             |         |          |           v          |
     --> tracing <--     tracing      socket    tc ingress   tc egress
          prog_1          prog_2      prog_3    classifier    action
          |  |              |           |         prog_4      prog_5
       |---  -----|  |------|          map_3        |           |
     map_1       map_2                              --| map_4 |--
```

http://man7.org/linux/man-pages/man2/bpf.2.html

eBPF – function calls

- It's possible to use map index as function pointer and use it to jump to other ebpf functions.

https://github.com/iovisor/bcc/tree/master/examples/networking/tunnel_monitor

```
/*from tunnel_monitor/monitor.c*/
BPF_TABLE("prog", int, int, parser, 10);     //initialize map of type prog
...
int handle_ingress(struct __sk_buff *skb) {
  ...
  parser.call(skb, 1);                       // jump to function 1, using pointer present in map
  ...
}
int handle_egress(struct __sk_buff *skb) {
  ...
  parser.call(skb, 2);                       //jump to function 2, using pointer present in map
  ...
}
# from tunnel_monitor/monitor.py
outer_fn = b.load_func("handle_outer", BPF.SCHED_CLS)     #load bpf function
inner_fn = b.load_func("handle_inner", BPF.SCHED_CLS)     #load bpf function

# using jump table for inner and outer packet split
parser = b.get_table("parser")               #retrieve map handle
parser[c_int(1)] = c_int(outer_fn.fd)        #populate map with function pointers
parser[c_int(2)] = c_int(inner_fn.fd)        #populate map with function pointers
```

BCC – BPF Compiler Collection

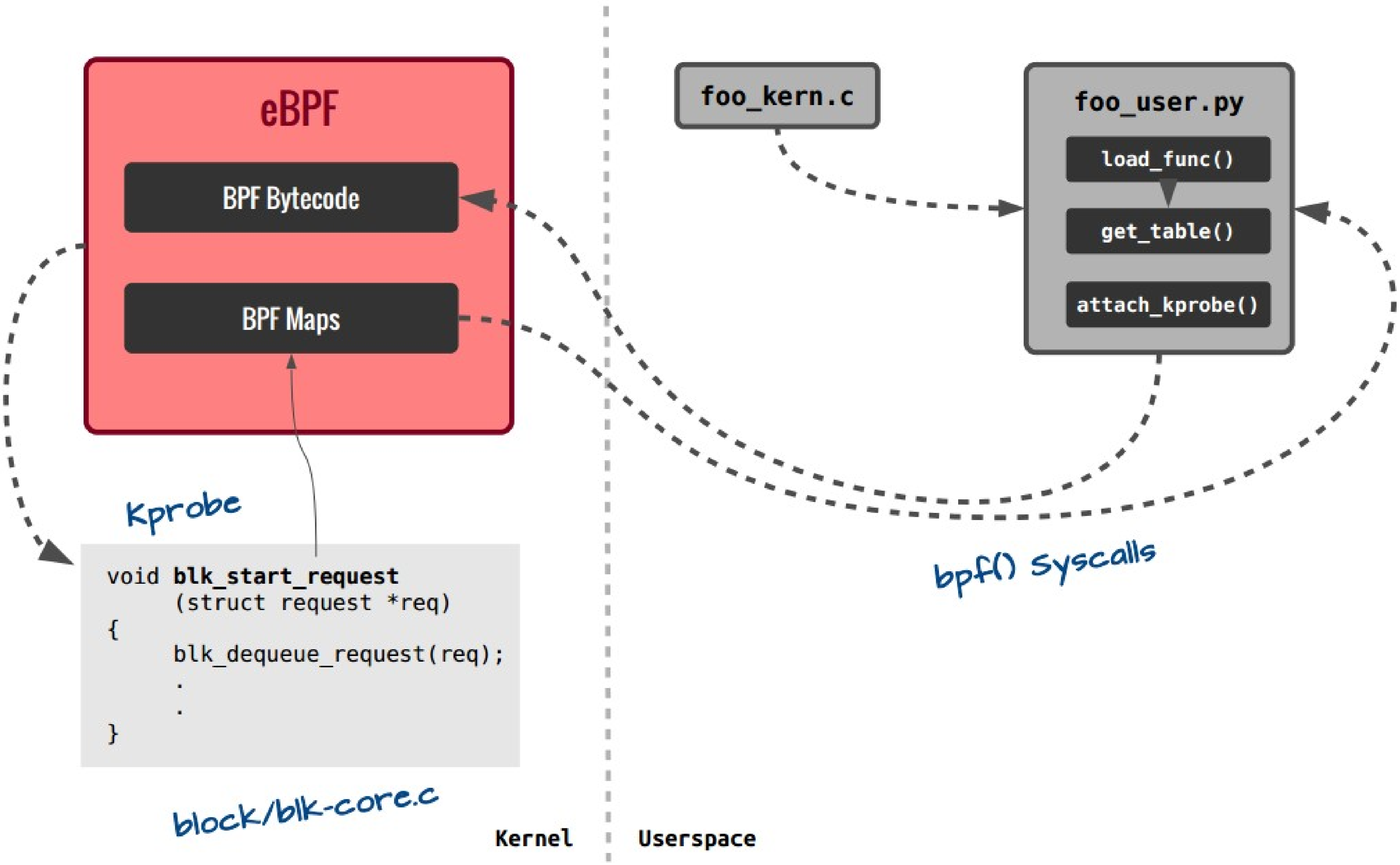
https://github.com/iovisor/bcc

BCC – BPF Compiler Collection

BCC is a toolkit for creating efficient kernel tracing and manipulation programs. It makes use of eBPF (Extended Berkeley Packet Filters)

BCC makes eBPF programs easier to write, with kernel instrumentation in C and a front-end in Python. It is suited for many tasks, including performance analysis and network traffic control.

EBPF – Session workflow (.py + BCC)



eBPF

BPF Bytecode

BPF Maps

foo_kern.c

foo_user.py

load_func()

get_table()

attach_kprobe()

Kprobe

```
void blk_start_request
     (struct request *req)
{
     blk_dequeue_request(req);
     .
     .
}
```

block/blk-core.c

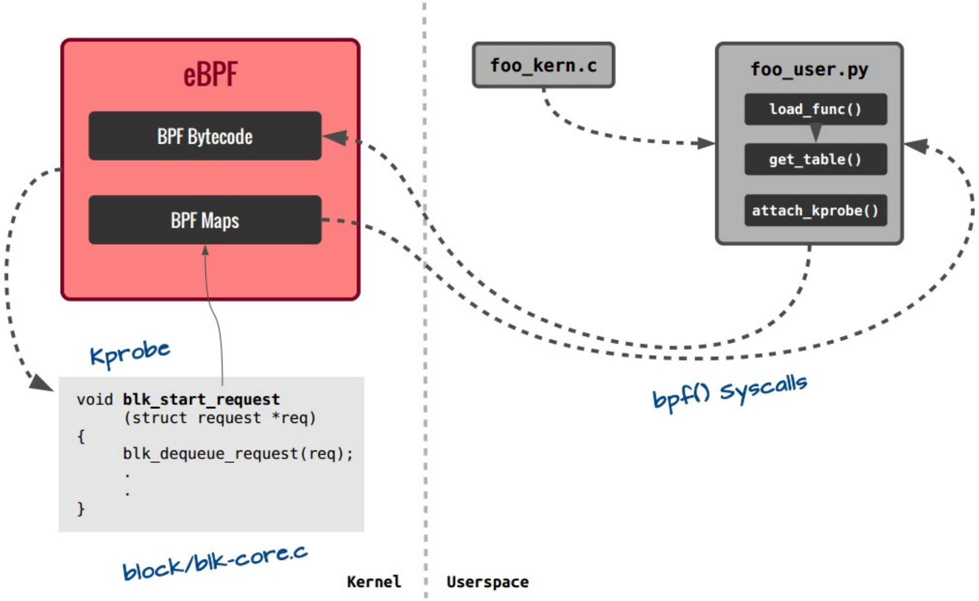
bpf() syscalls

Kernel

Userspace

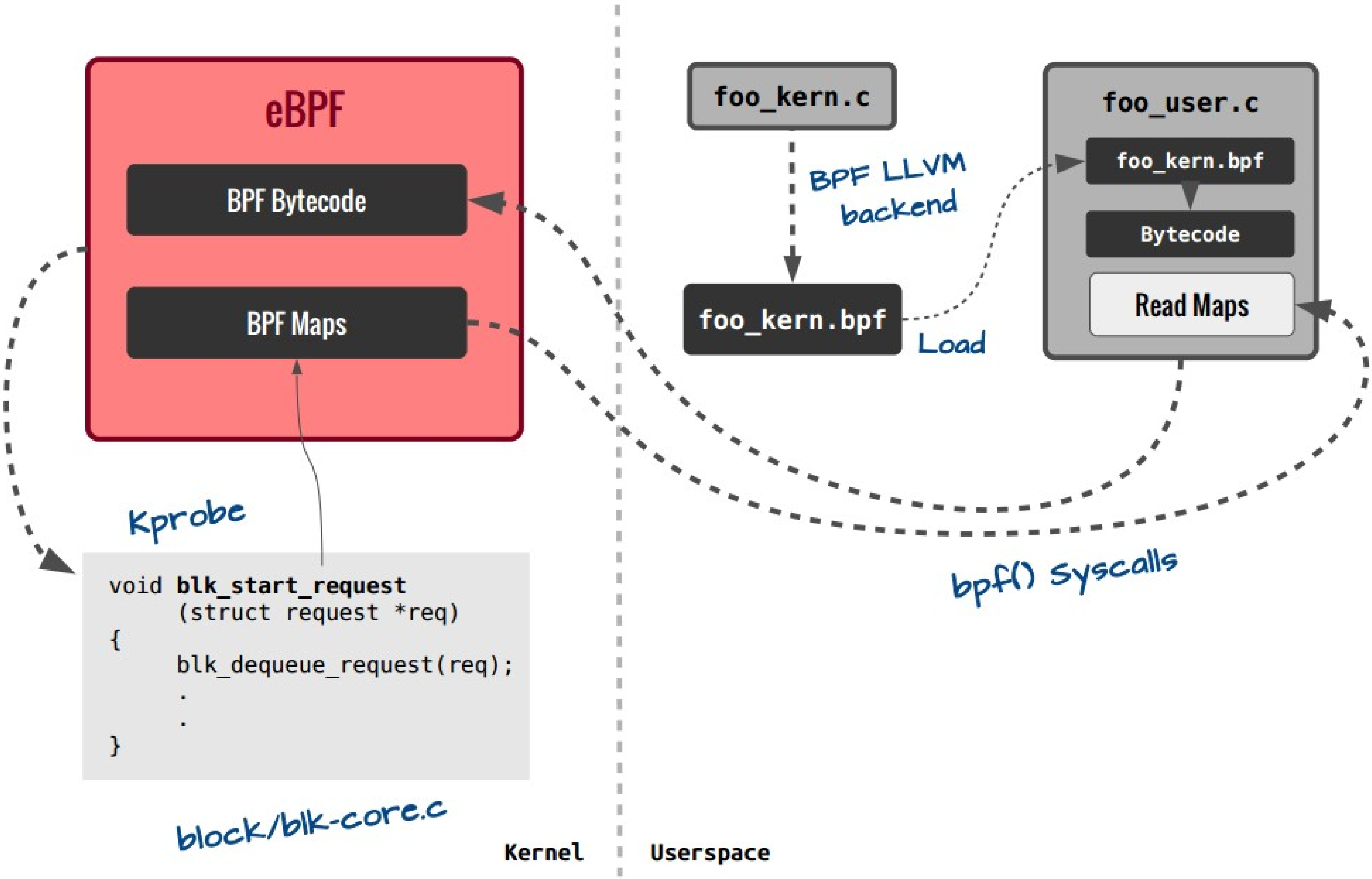
EBPF – Session workflow (.py + BCC)

- Write your BPF program in C, inline or in a separate file
- Write a python script that loads and interacts with your BPF program
- Attach the program to kprobes, socket, tc filter/action
- Read/update maps
- https://github.com/iovisor/bcc



```
void blk_start_request
     (struct request *req)
{
     blk_dequeue_request(req);
     .
     .
}
```

EBPF – Session workflow (c program)



eBPF

BPF Bytecode

BPF Maps

Kprobe

```
void blk_start_request
     (struct request *req)
{
     blk_dequeue_request(req);
     .
     .
}
```

block/blk-core.c

Kernel

Userspace

foo_kern.c

BPF LLVM backend

foo_kern.bpf

Load

foo_user.c

foo_kern.bpf

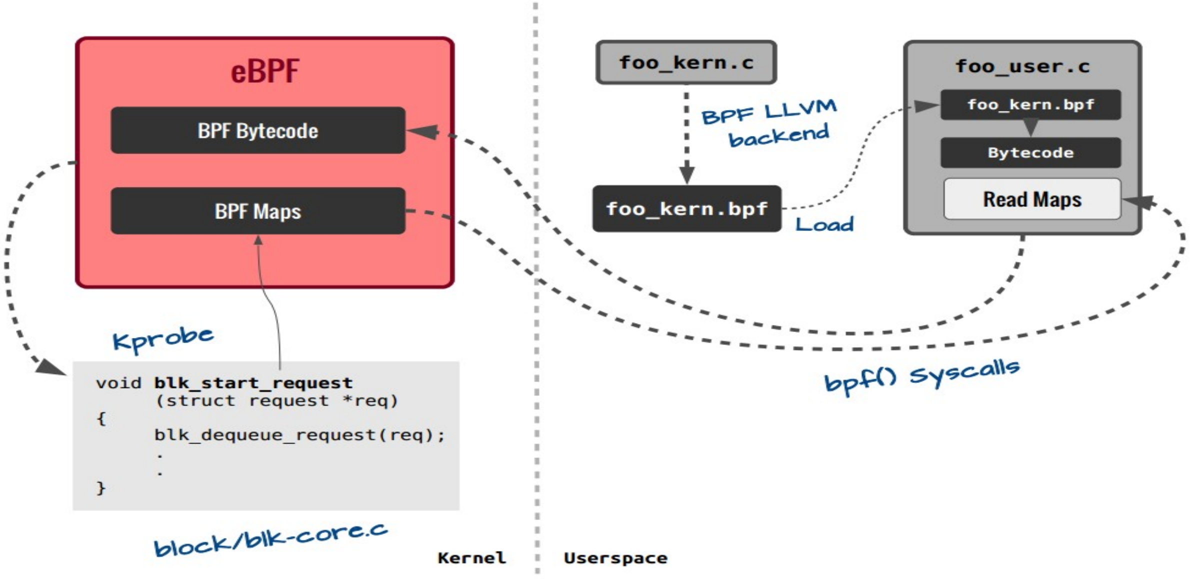
Bytecode

Read Maps

bpf() syscalls

EBPF – Session workflow (c program)

- C API for working with BPF programs - libbpfprog.so
- JIT compile a C source file to BPF bytecode (using clang+llvm)
- Load bytecode and maps to kernel with bpf() syscall
- Attach 1 or more BPF programs to 1 or more hook points
- kprobe, socket, tc classifier, tc action

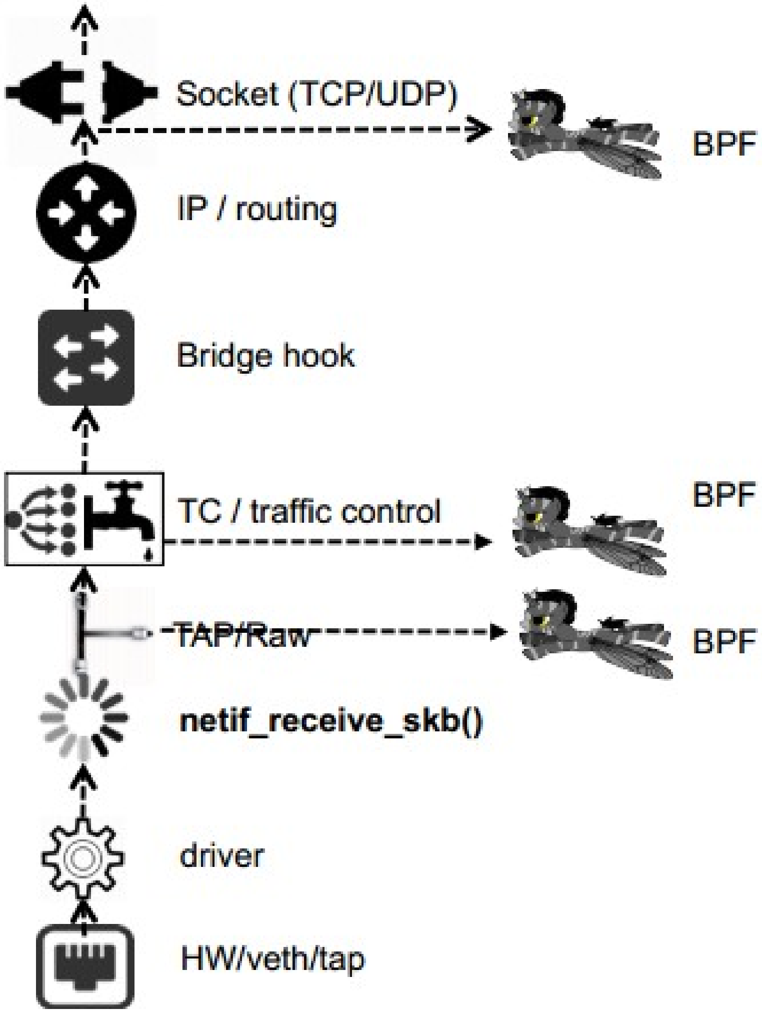


```
void blk_start_request
     (struct request *req)
{
     blk_dequeue_request(req);
     .
     .
}
```

eBPF & Networking

Hooking into Linux networking stack (RX)

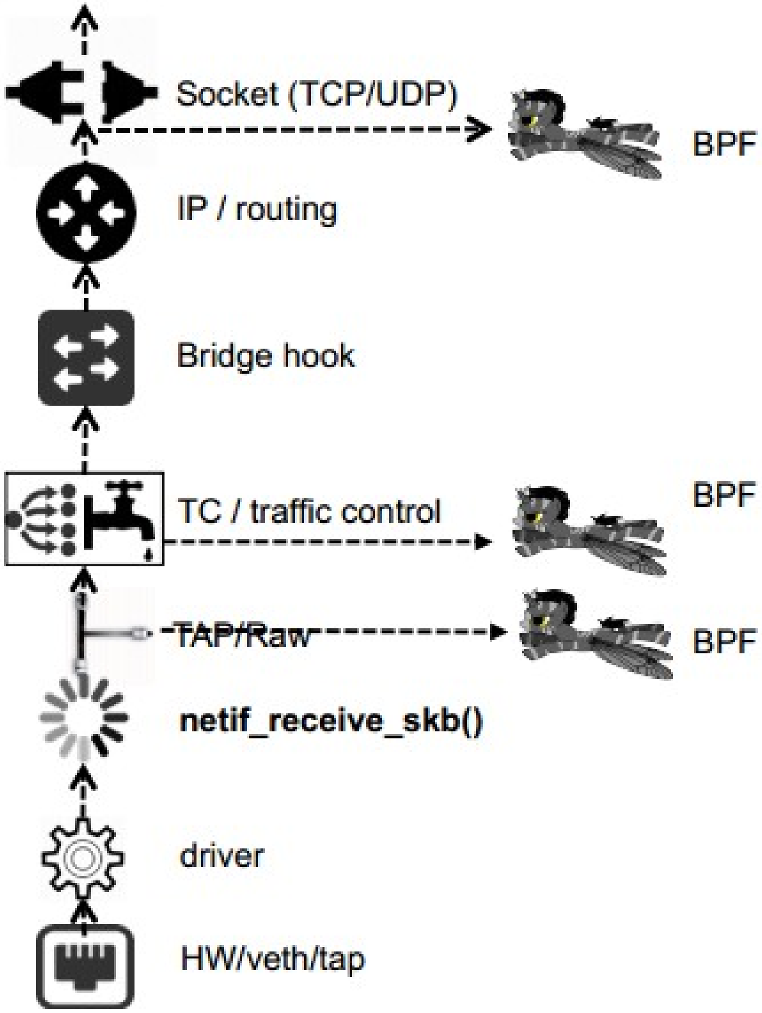
- BPF programs can attach to sockets or the traffic control (TC) subsystem, kprobes, syscalls, tracepoints ...
- sockets: STREAM (L4/UDP), DATAGRAM (L4/TCP) or RAW (TC)
- This allows to hook at different levels of the Linux networking stack, providing the ability to act on traffic that has or hasn’t been processed already by other pieces of the stack
- Opens up the possibility to implement network functions at different layers of the stack



eBPF & Networking

Hooking into Linux networking stack (TX)

- Opens up the possibility to implement network functions at different layers of the stack



eBPF Retrieving Data

How userspace program can retrieve data from eBPF program running in in-kernel vm ?

- Can read the <debugfs>/trace_pipe file from userspace (BCC wrap it to bpf_trace_printk())
- Can retrieve registers values (they are the ctx)
- Can read/write from maps
- Filtered packets, read from socket

eBPF Limitation & Safety

- Max 4096 instructions per program
- Stage 1 reject program if:
  - Loops and cyclic flow structure
  - Unreachable instructions
  - Bad jumps
- Stage 2 Static code analyzer:
  - Evaluate each path/instruction while keeping track of regs and stack states
  - Arguments validity in calls

EBPF – Some basic functions

```
BPF_FUNC_map_lookup_elem, /* void *map_lookup_elem(&map, &key) */
BPF_FUNC_map_update_elem, /* int map_update_elem(&map, &key, &value, flags) */
BPF_FUNC_map_delete_elem, /* int map_delete_elem(&map, &key) */
BPF_FUNC_probe_read,      /* int bpf_probe_read(void *dst, int size, void *src) */
BPF_FUNC_ktime_get_ns,    /* u64 bpf_ktime_get_ns(void) */
BPF_FUNC_trace_printk,    /* int bpf_trace_printk(const char *fmt, int fmt_size, ...) */
BPF_FUNC_get_prandom_u32, /* u32 prandom_u32(void) */
BPF_FUNC_get_smp_processor_id, /* u32 raw_smp_processor_id(void) */
BPF_FUNC_skb_store_bytes, /*store bytes into packet*/
BPF_FUNC_l3_csum_replace, /* recompute IP checksum*/
BPF_FUNC_l4_csum_replace,/*recompute TCP/UDP checksum*/
BPF_FUNC_tail_call, /*jump into another BPF program*/
BPF_FUNC_clone_redirect,/*redirect to another netdev*/
BPF_FUNC_get_current_pid_tgid, /*get current pid*/
BPF_FUNC_get_current_uid_gid, /*get current uid*/
BPF_FUNC_skb_vlan_push, /* bpf_skb_vlan_push(skb, vlan_proto, vlan_tci) */
BPF_FUNC_skb_vlan_pop,  /* bpf_skb_vlan_pop(skb) */
BPF_FUNC_perf_event_read,  /* u64 bpf_perf_event_read(&map, index) */
BPF_FUNC_redirect, /*redirect to another netdev*/
```

Application-layer traffic processing with eBPF

Write an eBPF application that parses HTTP packets and extracts (and prints on screen) the URL contained in the GET/POST request

Application-layer traffic processing with eBPF

http-parse v2

eBPF application that parses HTTP packets and extracts (and prints on screen) the URL contained in the GET/POST request. Complete version: manage also long urls splitted in multiple packets.

https://github.com/netgroup-polito/ebpf-test/blob/master/http-parse-v2.py
https://github.com/netgroup-polito/ebpf-test/blob/master/http-parse-v2.c

eBPF socket filter.

Filters IP and TCP packets, containing "HTTP", "GET", "POST" in payload and all subsequent packets belonging to the same session, having the same (ip_src,ip_dst,port_src,port_dst).
Program is loaded as PROG_TYPE_SOCKET_FILTER and attached to a socket, bind to eth0.
Matching packets are forwarded to user space, others dropped by the filter.

Python script reads filtered raw packets from the socket, if necessary reassembles packets belonging to the same session, and prints on stdout the first line of the HTTP GET/POST request.

Application-layer traffic processing with eBPF - Conclusions

EBPF is very powerful for some specific kind of analysis and processing. For Example statistics on traffic type, traffic control, kernel events end performance analysis etc...

In my case (Application-layer traffic) some constraints on eBPF language forced me to split this type of analysis in part in eBPF and in part in userspace.

This hybrid approach in this case is the only way because I can't perform complex HTTP payload analysis inside ebpf program, mainly because of string operation.

eBPF Usecases & Examples

- **Some eBPF example using BCC (from https://github.com/iovisor/bcc)**
- tools/tcpaccept: Trace TCP passive connections (accept()).
- tools/tcpconnect:Trace TCP active connections (connect()).
- examples/distributed_bridge/: Distributed bridge example.
- examples/simple_tc.py: Simple traffic control example.
- examples/tc_neighbor_sharing.py: Per-IP classification and rate limiting.
- examples/tunnel_monitor/: Efficiently monitor traffic flows.
- examples/vlan_learning.py: Demux Ethernet traffic into worker veth+namespaces.

Links

- https://github.com/iovisor/bcc
- https://github.com/iovisor/bpf-docs
- http://lwn.net/Articles/603984/
- http://lwn.net/Articles/603983/
- https://lwn.net/Articles/625224/
- https://www.kernel.org/doc/Documentation/networking/filter.txt
- http://man7.org/linux/man-pages/man2/bpf.2.html
- https://linuxplumbersconf.org/2015/ocw//system/presentations/3249/original/bpf_llvm_2015aug19.pdf
- https://videos.cdn.redhat.com/summit2015/presentations/13737_an-overview-of-linux-networking-subsystem-extended-bpf.pdf
- https://github.com/torvalds/linux/tree/master/samples/bpf
- https://suchakra.wordpress.com/2015/05/18/bpf-internals-i/
- https://suchakra.wordpress.com/2015/08/12/bpf-internals-ii/
- http://events.linuxfoundation.org/sites/events/files/slides/tracing-linux-ezannoni-linuxcon-ja-2015_0.pdf