eBPF
extended Barkeley packet filter

Brief Introduction

Bertrone Matteo - Polytechnic of Turin

December 2015

BPF vs eBPF - Classic 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

- Idea: improve and extend existing BPF infrastructure.
- Programs can be written in C and translated into eBPF.
- New set of patches introduced in the Linux kernel since 3.15 (June 8th, 2014).
- Current Linux kernel version 4.3.99 (November 2015).

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

eBPF kernel internals

- Internally: instruction set format with similar underlying principles from BPF.
- This new ISA is called 'eBPF'.
- It is designed to be JITed with one to one mapping.
- It opens up the possibility for GCC/LLVM compilers to generate optimized eBPF code through an eBPF backend that performs almost as fast as natively compiled code.
- Purpose: Write programs in "restricted C" and compile into eBPF, so that it can just-in-time map to modern 64-bit CPUs with minimal performance overhead over two steps:
- C -> eBPF -> native code.

eBPF LLVM back-end

- It's a very small and simple backend.
- There is no support for global variables, arbitrary function calls, floating point, varargs, exceptions, indirect jumps, arbitrary pointer arithmetic, alloca, etc.
- From C front-end point of view it's very restricted.
- It's done on purpose, since kernel rejects all programs that it cannot prove safe.
- It rejects programs with loops and with memory accesses via arbitrary pointers.
- When kernel accepts the program it is guaranteed that program will terminate and will not crash the kernel.

eBPF Maps

- Generic memory allocated.
- Transfer data from userspace to kernel and vice versa.
- key/value storage of different types.
- A map is identified by a file descriptor returned by a bpf() system call that creates the map
- Types of maps: BPF_MAP_TYPE_ARRAY, BPF_MAP_TYPE_HASH
- Share data among many eBPF programs (see next)

eBPF Maps - Example

Mantain a statistic on different traffic type (TCP,UDP,ICMP…)

(In efficient way, with a map)

```c
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 |--
```

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

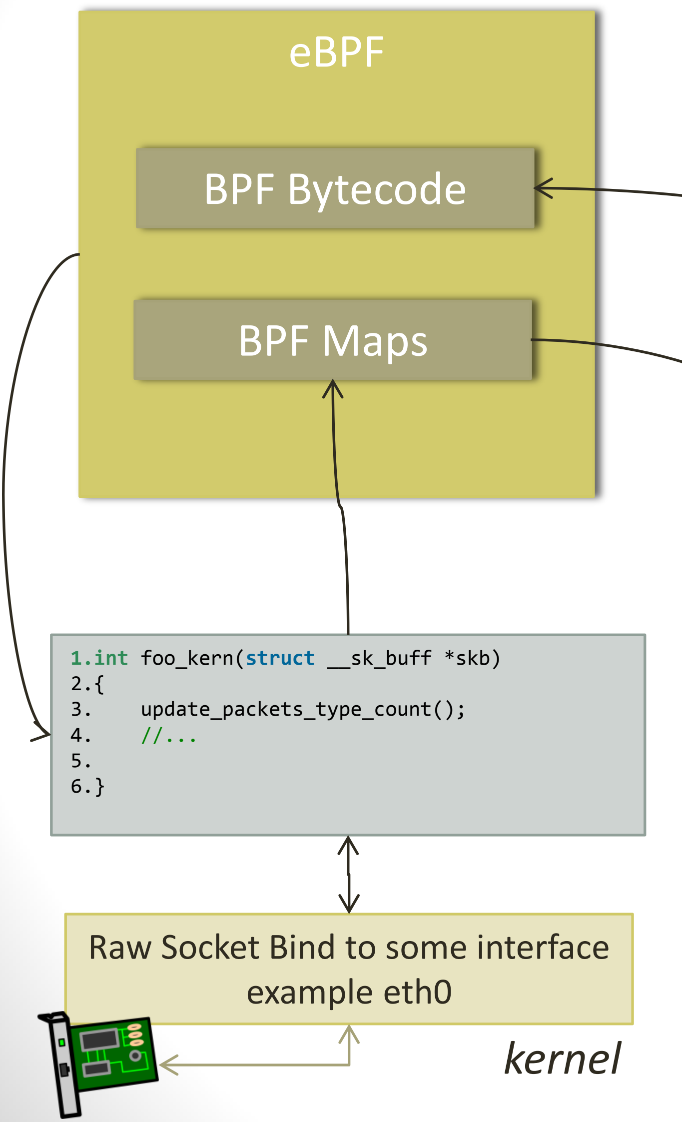
https://github.com/iovisor/bcc

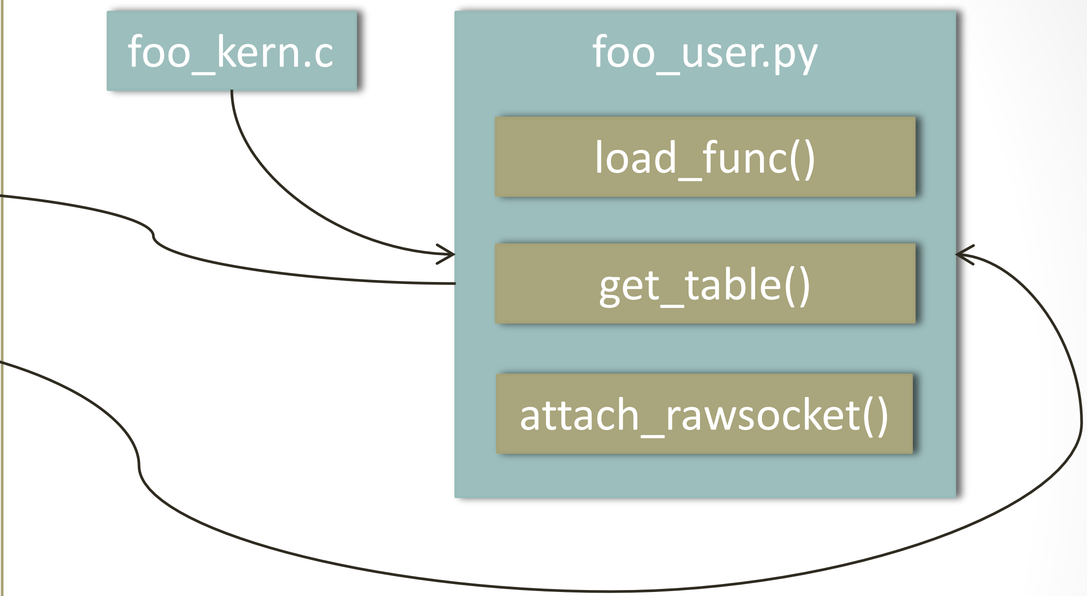
BCC - BPF Compiler Collection

- BCC is a toolkit for creating efficient kernel tracing and manipulation programs.

- 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, network traffic control and packet filtering.

BCC - BPF Compiler Collection



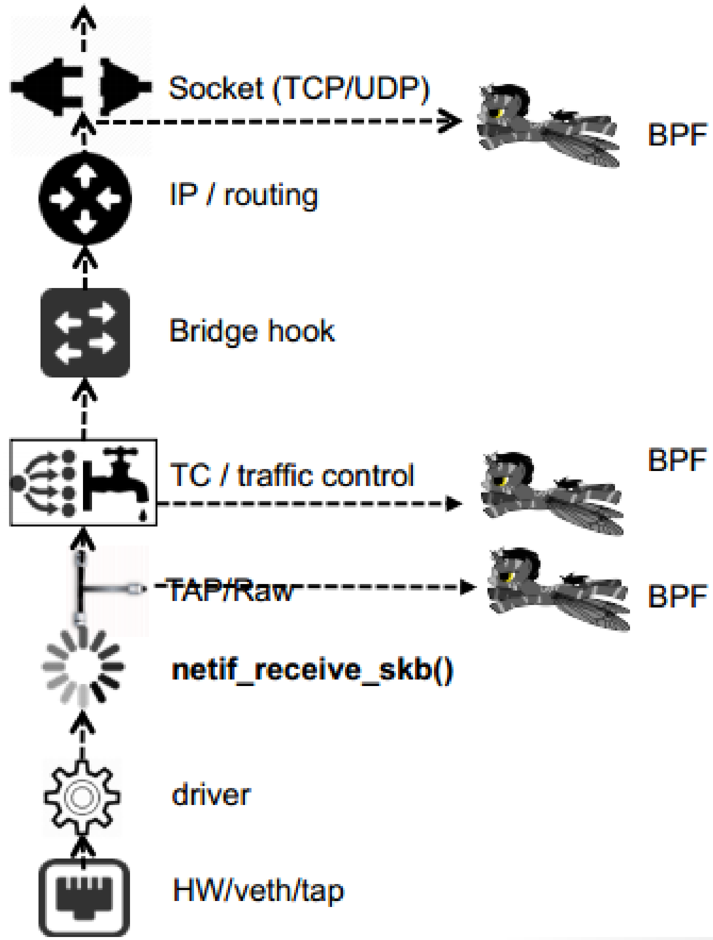


- Write your BPF program in C
- Write a python script that loads and interacts with your BPF program
- Attach the program to kprobes, socket, tc filter/action
- Read/update maps

userspace

eBPF & Networking
Hooking into Linux networking stack

- BPF programs can attach to sockets or the traffic control (TC) subsystem, kprobes, syscalls, tracepoints ...
- sockets: STREAM (L4/TCP), DATAGRAM (L4/UDP) 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 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()).Debug solution, not stable.
- Can retrieve registers values (they are the ctx)
- Can read/write from maps
- Filtered packets, read from socket

eBPF Limitation and 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

Bertrone Matteo - Polytechnic of Turin

December 2015

Project purpose

The eBPF has been recently proposed as an extension of the BPF virtual machine, defined many years ago and still used for packet filtering. The eBPF comes with additional features (e.g., more powerful virtual machines) as well as an accompanying compiler (LLVM) that can generate directly eBPF code. Furthermore, eBPF is now part of the standard Linux kernel, named as "BPF".

- This project aims at:
  - studying the architecture of the eBPF
  - evaluating the possible applications of the eBPF (e.g., through the available samples) and its degree of interaction with the LLVM compiler
  - making a proof of concept of an eBPF application that parses HTTP packets and extracts (and prints on screen) the URL contained in the GET/POST request.

Filter HTTP traffic (version I)

eBPF

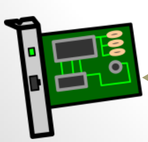
BPF program of type SOCKET_FILTER

http-parse-v1.c

```
/*eBPF program.
Filter IP and TCP packets, having payload not e
mpty and containing "HTTP", "GET", "POST" ... a
s first bytes of payload.
*/
int http_filter(struct __sk_buff *skb) {
    //code ...
    //send packet to userspace
    KEEP:
    return -1;

    //drop the packet
    DROP:
    return 0;
}
```

Raw Socket Bind to eth0



Filtered packets on socket

http-parse-v1.py

- Load eBPF program.
- Attach it to raw socket bind to eth0.
- Read filtered packet
- print the url contained in HTTP GET/POST request.

GET /path/to/file/index.html HTTP/1.0
POST /path/to/file/index.html HTTP/1.0

- *This version truncates long urls splitted in more packets*

kernel | *userspace*

Filter HTTP traffic (version II)

eBPF

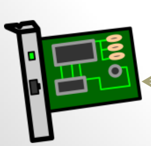
BPF program of type SOCKET_FILTER

BPF session Map

http-parse-v2.c

```
/*eBPF program.
Filter IP and TCP packets, having payload not empty
and containing "HTTP", "GET", "POST"  as first bytes
of payload AND ALL the other packets having same
(src_ip,dst_ip,src_port,dst_port) this means
belonging to the same "session“.
this additional check avoids url truncation, if url
is too long  userspace script, if necessary,
reassembles urls splitted in more packets.
*/
int http_filter(struct __sk_buff *skb) {
    //code ...
    //send packet to userspace
    KEEP:
    return -1;
    //drop the packet
    DROP:
    return 0;
}
```

Raw Socket Bind to eth0



kernel

Filtered packets on socket

userspace

http-parse-v2.py

- Load eBPF program.
- Attach it to raw socket bind to eth0.
- Initialize sessions Map.
    - Key:(ip_src,ip_dst,port_src,port_dst)
    - Value: timestamp
- Read filtered packet (all packets of HTTP session)
- Perform some check to eventually reassemble splitted packets.
- print the url contained in HTTP GET/POST request.

GET /path/to/file/index.html HTTP/1.0
POST /path/to/file/index.html HTTP/1.0

- *This version solve the problem of long urls splitted in more packets*

Filter HTTP traffic (version II)

http-parse-v2.c

```
struct Key {
    u32 src_ip;               //source ip
    u32 dst_ip;               //destination ip
    unsigned short src_port;  //source port
    unsigned short dst_port;  //destination port
};

struct Leaf {
    int timestamp;            //timestamp in ns
};

//BPF_TABLE(map_type, key_type, leaf_type, table_name, num_entry)
//map <Key, Leaf>
//tracing sessions having same Key(dst_ip, src_ip, dst_port,src_port)
BPF_TABLE("hash", struct Key, struct Leaf, sessions, 1024);

int http_filter(struct __sk_buff *skb) {

    struct ethernet_t *ethernet = cursor_advance(cursor, sizeof(*ethernet));
    //filter IP packets (ethernet type = 0x0800)
    if (!(ethernet->type == 0x0800)){
        goto DROP;
    }

    struct ip_t *ip = cursor_advance(cursor, sizeof(*ip));
    //filter TCP packets (ip next protocol = 0x06)
    if (ip->nextp != IP_TCP) {
        goto DROP;
    }

    //retrieve ip src/dest and port src/dest of current packet
    //and save it into struct Key
    key.dst_ip = ip->dst;
    key.src_ip = ip->src;
    key.dst_port = tcp->dst_port;
    key.src_port = tcp->src_port;
```

Filter HTTP traffic (version II)

http-parse-v2.c

```
//find a match with an HTTP message
    //HTTP
    if ( (payload_array[0] == 'H') && (payload_array[1] == 'T') && (payload_array[2] == 'T') && (payload_array[3] == 'P')){
        goto HTTP_MATCH;
    }
    //GET
    if ( (payload_array[0] == 'G') && (payload_array[1] == 'E') && (payload_array[2] == 'T') ){
        goto HTTP_MATCH;
    }
    //POST
    if ( (payload_array[0] == 'P') && (payload_array[1] == 'O') && (payload_array[2] == 'S') && (payload_array[3] == 'T')){
        goto HTTP_MATCH;
    }

    //no HTTP match
    //check if packet belong to an HTTP session
    struct Leaf * lookup_leaf = sessions.lookup(&key);
    if(lookup_leaf){
        //send packet to userspace
        goto KEEP;
    }
    goto DROP;

    //keep the packet and send it to userspace retruning -1
    HTTP_MATCH:
    //if not already present, insert into map <Key, Leaf>
    leaf.timestamp = 0;
    sessions.lookup_or_init(&key, &leaf);
    sessions.update(&key,&leaf);

    //send packet to userspace returning -1
    KEEP:
    return -1;

    //drop the packet returning 0
    DROP:
    return 0;

}
```

Filter HTTP traffic (version II)

http-parse-v2.py

```
# initialize BPF - load source code from http-parse.c
bpf = BPF(src_file = "http-parse-v2.c",debug = 0)

#load eBPF program http_filter of type SOCKET_FILTER into the kernel eBPF vm
#more info about eBPF program types
#http://man7.org/linux/man-pages/man2/bpf.2.html
function_http_filter = bpf.load_func("http_filter", BPF.SOCKET_FILTER)

#create raw socket, bind it to eth0
#attach bpf program to socket created
BPF.attach_raw_socket(function_http_filter, "eth0")

#get file descriptor of the socket previously created inside BPF.attach_raw_socket
socket_fd = function_http_filter.sock

#create python socket object, from the file descriptor
sock = socket.fromfd(socket_fd,socket.PF_PACKET,socket.SOCK_RAW,socket.IPPROTO_IP)
#set it as blocking socket
sock.setblocking(True)

#get pointer to bpf map of type hash
bpf_sessions = bpf.get_table("sessions")

#packets counter
packet_count = 0

#dictionary containing association <key(ipsrc,ipdst,portsrc,portdst),payload_string>
#if url is not entirely contained in only one packet, save the firt part of it in this local dict
#when I find \r\n in a next pkt, append and print all the url
local_dictionary = {}
```

Filter HTTP traffic (version II)

http-parse-v2.py

```
#retrieve raw packet from socket
packet_str = os.read(socket_fd,4096) #set packet lenght to max packet lenght on the interface
packet_count += 1

#convert packet into bytearray
packet_bytearray = bytearray(packet_str)

#ethernet header length
ETH_HLEN = 14

 #IP HEADER
 #https://tools.ietf.org/html/rfc791
 # 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 # +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 # |Version|  IHL  |Type of Service|          Total Length         |
 # +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 #
 #IHL : Internet Header Length is the length of the internet header
 #value to multiply * 4 byte
 #e.g. IHL = 5 ; IP Header Length = 5 * 4 byte = 20 byte
 #
 #Total Lenght: This 16-bit field defines the entire packet size,
 #including header and data, in bytes.

 #calculate packet total lenght
 total_lenght = packet_bytearray[ETH_HLEN + 2]               #load MSB
 total_lenght = total_lenght << 8                            #shift MSB
 total_lenght = total_lenght + packet_bytearray[ETH_HLEN+3]  #add LSB

 #calculate ip header lenght
 ip_header_length = packet_bytearray[ETH_HLEN]               #load Byte
 ip_header_length = ip_header_length & 0x0F                  #mask bits 0..3
 ip_header_length = ip_header_length << 2                    #shift to obtain lenght

 #retrieve ip source/dest
 ip_src_str = packet_str[ETH_HLEN+12:ETH_HLEN+16]                #ip source offset 12..15
 ip_dst_str = packet_str[ETH_HLEN+16:ETH_HLEN+20]                #ip dest   offset 16..19

 ip_src = int(toHex(ip_src_str),16)
 ip_dst = int(toHex(ip_dst_str),16)
```

Filter HTTP traffic (version II)

http-parse-v2.py

```
#TCP HEADER
#https://www.rfc-editor.org/rfc/rfc793.txt
#  12              13              14              15
#  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
# +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
# |  Data |           |U|A|P|R|S|F|                               |
# | Offset| Reserved  |R|C|S|S|Y|I|            Window             |
# |       |           |G|K|H|T|N|N|                               |
# +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 #
 #Data Offset: This indicates where the data begins.
 #The TCP header is an integral number of 32 bits long.
 #value to multiply * 4 byte
 #e.g. DataOffset = 5 ; TCP Header Length = 5 * 4 byte = 20 byte

 #calculate tcp header lenght
 tcp_header_lenght = packet_bytearray[ETH_HLEN + ip_header_length + 12]  #load Byte
 tcp_header_lenght = tcp_header_lenght & 0xF0                            #mask bit 4..7
 tcp_header_lenght = tcp_header_lenght >> 2                              #SHR 4 ; SHL 2 -> SHR 2

 #retrieve port source/dest
 port_src_str = packet_str[ETH_HLEN+ip_header_length:ETH_HLEN+ip_header_length+2]
 port_dst_str = packet_str[ETH_HLEN+ip_header_length+2:ETH_HLEN+ip_header_length+4]

 port_src = int(toHex(port_src_str),16)
 port_dst = int(toHex(port_dst_str),16)

 #calculate payload offset
 payload_offset = ETH_HLEN + ip_header_length + tcp_header_lenght

 #payload_string contains only packet payload
 payload_string = packet_str[(payload_offset):(len(packet_bytearray))]

 #CR + LF (substring to find)
 crlf = "\r\n"

 #current_Key contains ip source/dest and port source/map
 #useful for direct bpf_sessions map access
 current_Key = bpf_sessions.Key(ip_src,ip_dst,port_src,port_dst)
```

```
#looking for HTTP GET/POST request
if ((payload_string[:3] == "GET") or (payload_string[:4] == "POST")   or (payload_string[:4] == "HTTP")  \
or ( payload_string[:3] == "PUT") or (payload_string[:6] == "DELETE") or (payload_string[:4] == "HEAD") ):
  #match: HTTP GET/POST packet found
  if (crlf in payload_string):
    #url entirely contained in first packet -> print it all
    printUntilCRLF(payload_string)

    #delete current_Key from bpf_sessions, url already printed. current session not useful anymore
     try:
       del bpf_sessions[current_Key]
     except:
       print ("error during delete from bpf map ")
   else:
     #url NOT entirely contained in first packet
     #not found \r\n in payload.
     #save current part of the payload_string in dictionary <key(ips,ipd,ports,portd),payload_string>
     local_dictionary[binascii.hexlify(current_Key)] = payload_string
 else:
   #NO match: HTTP GET/POST  NOT found

   #check if the packet belong to a session saved in bpf_sessions
   if (current_Key in bpf_sessions):
     #check id the packet belong to a session saved in local_dictionary
     #(local_dictionary mantains HTTP GET/POST url not printed yet because splitted in N packets)
     if (binascii.hexlify(current_Key) in local_dictionary):
       #first part of the HTTP GET/POST url is already present in local dictionary (prev_payload_string)
       prev_payload_string = local_dictionary[binascii.hexlify(current_Key)]
       #looking for CR+LF in current packet.
       if (crlf in payload_string):
         #last packet. containing last part of HTTP GET/POST url splitted in N packets.
         #append current payload
         prev_payload_string += payload_string
         #print HTTP GET/POST url
         printUntilCRLF(prev_payload_string)
         #clean bpf_sessions & local_dictionary
         try:
           del bpf_sessions[current_Key]
           del local_dictionary[binascii.hexlify(current_Key)]
         except:
           print ("error deleting from map or dictionary")
       else:
         #NOT last packet. containing part of HTTP GET/POST url splitted in N packets.
         #append current payload
         prev_payload_string += payload_string
         #check if not size exceeding (usually HTTP GET/POST url < 8K )
         if (len(prev_payload_string) > MAX_URL_STRING_LEN):
           print("url too long")
           try:
             del bpf_sessions[current_Key]
             del local_dictionary[binascii.hexlify(current_Key)]
           except:
             print ("error deleting from map or dict")
         #update dictionary
         local_dictionary[binascii.hexlify(current_Key)] = prev_payload_string
     else:
       #first part of the HTTP GET/POST url is NOT present in local dictionary
       #bpf_sessions contains invalid entry -> delete it
       try:
         del bpf_sessions[current_Key]
       except:
         print ("error del bpf_session")
```

Filter HTTP traffic (version II)

http-parse-v2.py

```
CLEANUP_N_PACKETS  = 50       #run cleanup every CLEANUP_N_PACKETS packets received
MAX_URL_STRING_LEN = 8192     #max url string len (usually 8K)
MAX_AGE_SECONDS    = 30       #max age entry in bpf_sessions map

#cleanup function
def cleanup():
  #get current time in seconds
  current_time = int(time.time())
  #looking for leaf having:
  #timestap  == 0        --> update with current timestamp
  #AGE > MAX_AGE_SECONDS --> delete item
  for key,leaf in bpf_sessions.items():
    try:
      current_leaf = bpf_sessions[key]
      #set timestamp if timestamp == 0
      if (current_leaf.timestamp == 0):
        bpf_sessions[key] = bpf_sessions.Leaf(current_time)
      else:
        #delete older entries
        if (current_time - current_leaf.timestamp > MAX_AGE_SECONDS):
          del bpf_sessions[key]
    except:
      print("cleanup exception.")
  return

#check if dirty entry are present in bpf_sessions
if (((packet_count) % CLEANUP_N_PACKETS) == 0):
  cleanup()
```

Project Code Links

- https://github.com/netgroup-polito/ebpf-test

Conclusions

- EBPF is very powerful for some specific kind of analysis and processing. For Example statistics on traffic type, traffic control, kernel events and 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 is the only way because I can't perform complex HTTP payload analysis inside ebpf program, mainly because of limitations on 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