DPDK
DATA PLANE DEVELOPMENT KIT

Using nDPI over DPDK to Classify and Block Unwanted Network Traffic

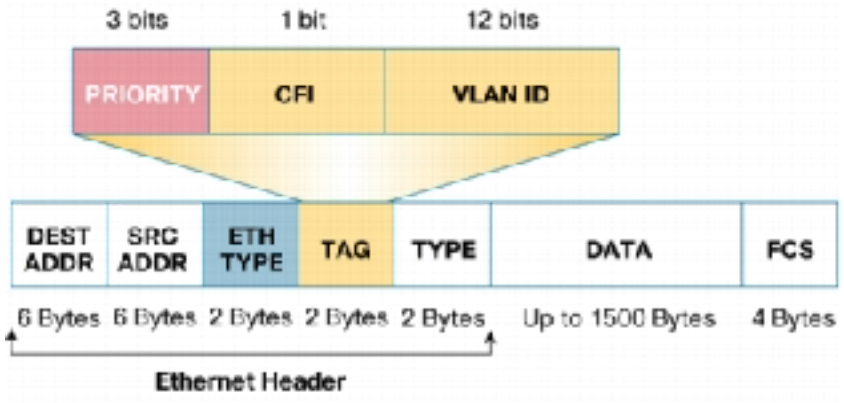
Luca Deri <deri@ntop.org>
@lucaderi

Traffic Classification: an Overview

- Traffic classification is compulsory to understand the traffic flowing on a network and enhance user experience by tuning specific network parameters.
- Main classification methods include:
  - TCP/UDP port classification.
  - QoS based classification (DSCP).
  - Statistical Classification.
  - Deep Packet Inspection.

Port- and DSCP-based Traffic Classification

- Port-based Classification
  - In the early day of the Internet, network traffic protocols were identified by protocol and port.
  - Can classify only application protocols operating on well known ports (no rpcbind or portmap).
  - Easy to cheat and thus unreliable (TCP/80 != HTTP).
- QoS Markers (DSCP)
  - Similar to port classification but based on QoS tags.
  - Usually ignored as it is easy to cheat and forge.



Statistical Traffic Classification

- Classification of IP packets (size, port, flags, IP addresses) and flows (duration, frequency…).
- Based on rules written manually, or automatically using machine learning (ML) algorithms.
- ML requires a training set of very good quality, and it is generally computationally intensive.
- Detection rate can be as good as 95% for cases which were covered by the training set, and poor accuracy for all the other cases.

Deep Packet Inspection (DPI)

- Technique that inspects the packet payload.
- Computationally intensive with respect to simple packet header analysis.
- Concerns about privacy and confidentiality of inspected data.
- Encryption is becoming pervasive, thus challenging DPI techniques.
- No false positives unless statistical methods or IP range/flow analysis are used by DPI tools.

Using DPI in Traffic Monitoring

- Packet header analysis is no longer enough as it is unreliable and thus useless.
- Security and network administrators want to know what are the real protocols flowing on a network, this regardless of the port being used.
- Selective metadata extraction (e.g. HTTP URL or User-Agent) is necessary to perform accurate monitoring and thus this task should be performed by the DPI toolkit without replicating it on monitoring applications.

Welcome to nDPI

- In 2012 we decided to develop our own GNU LGPL DPI toolkit (based on a unmaintained project named OpenDPI) in order to build an <u>open</u> DPI layer for ntop and third-party applications (Wireshark, netfilter, ML tools…).
- Protocols supported exceed 240 and include:
  - P2P (Skype, BitTorrent)
  - Messaging (Viber, Whatsapp, MSN, Facebook)
  - Multimedia (YouTube, Last.gm, iTunes)
  - Conferencing (Webex, CitrixOnLine)
  - Streaming (Zattoo, Icecast, Shoutcast, Netflix)
  - Business (VNC, RDP, Citrix, *SQL)

What is a Protocol in nDPI? [1/2]

- Each protocol is identified as <major>.<minor> protocol. Example:
  - DNS.Facebook
  - QUIC.YouTube and QUIC.YouTubeUpload

- Caveat: Skype or Facebook are protocols in the nDPI world but not for IETF.
- The first question people ask when they have to evaluate a DPI toolkit is: how many protocol do you support? This is not the right question.

What is a Protocol in nDPI? [2/2]

- Today most protocols are HTTP/SLL-based.
- nDPI includes support for string-based protocols detection:
  - DNS query name
  - HTTP Host/Server header fields
  - SSL/QUIC SNI (Server Name Indication)

- Example: NetFlix detection

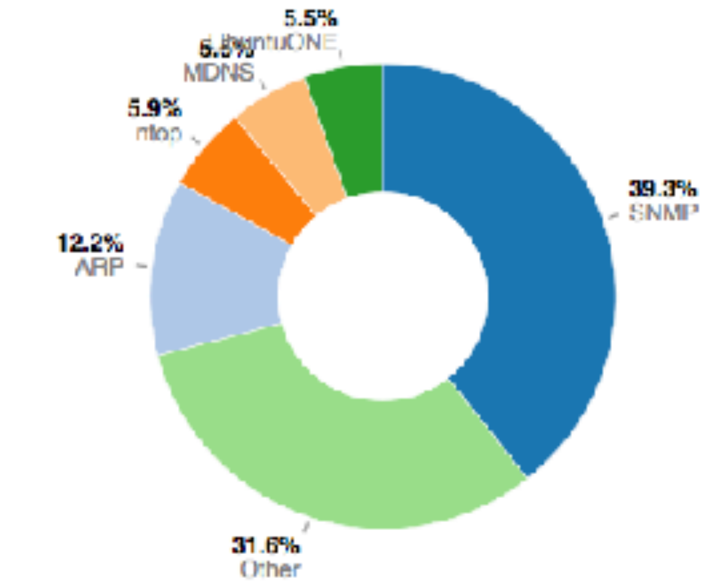
```
{ "netflix.com", NULL,   "netflix" TLD,   "NetFlix",  NDPI_PROTOCOL_NETFLIX, NDPI_PROTOCOL_CATEGORY_STREAMING, NDPI_PROTOCOL_FUN },
{ "nflxext.com", NULL,   "nflxext" TLD,   "NetFlix",  NDPI_PROTOCOL_NETFLIX, NDPI_PROTOCOL_CATEGORY_STREAMING, NDPI_PROTOCOL_FUN },
{ "nflximg.com", NULL,   "nflximg" TLD,   "NetFlix",  NDPI_PROTOCOL_NETFLIX, NDPI_PROTOCOL_CATEGORY_STREAMING, NDPI_PROTOCOL_FUN },
{ "nflximg.net", NULL,   "nflximg" TLD,   "NetFlix",  NDPI_PROTOCOL_NETFLIX, NDPI_PROTOCOL_CATEGORY_STREAMING, NDPI_PROTOCOL_FUN },
{ "nflxvideo.net", NULL, "nflxvideo" TLD, "NetFlix",  NDPI_PROTOCOL_NETFLIX, NDPI_PROTOCOL_CATEGORY_STREAMING, NDPI_PROTOCOL_FUN },
{ "nflxso.net", NULL,    "nflxso" TLD,    "NetFlix",  NDPI_PROTOCOL_NETFLIX, NDPI_PROTOCOL_CATEGORY_STREAMING, NDPI_PROTOCOL_FUN },
```

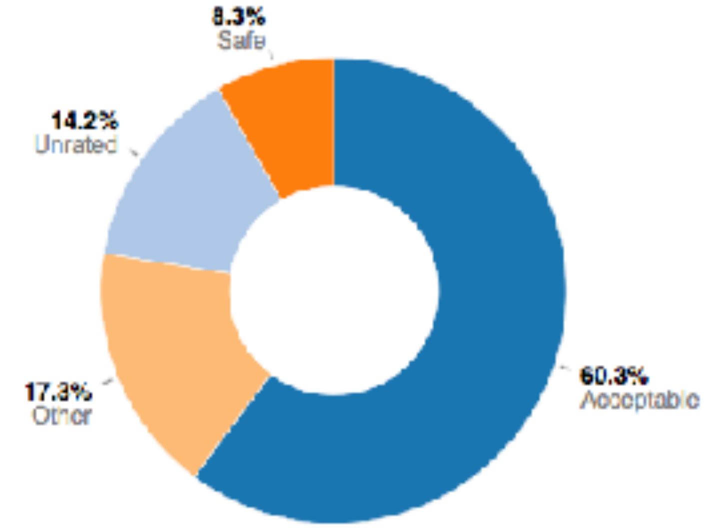
nDPI Categories [1/2]

- Protocols are too many, and they increase daily.
- Many people are not familiar with protocol names.
- Often people ask us questions like “How can I prevent my children from using social networks?”
- Solution
  - nDPI allows protocols to be clustered in user-defined categories such as VoIP, P2P, Cloud…
  - Categories can include thousand of entries and can be (re-)loaded dynamically. Example: malware, mining, advertisement, banned site, inappropriate content…

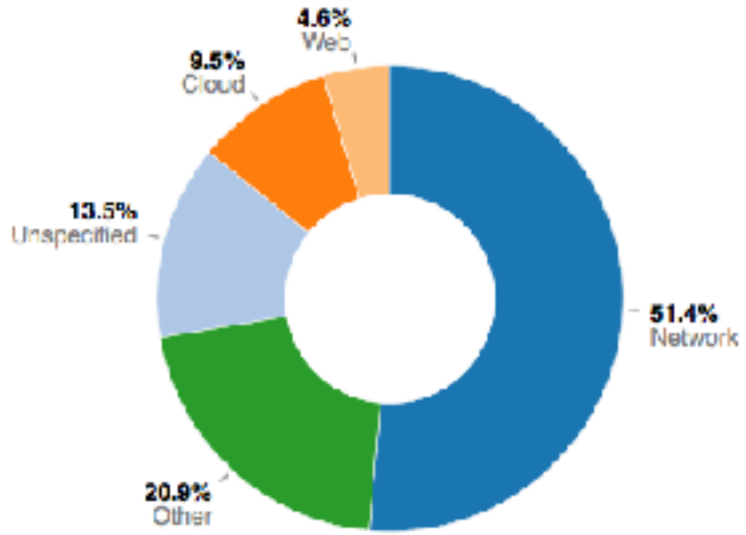
nDPI Categories [2/2]

Application Protocol Overview





Application Protocol Category Overview



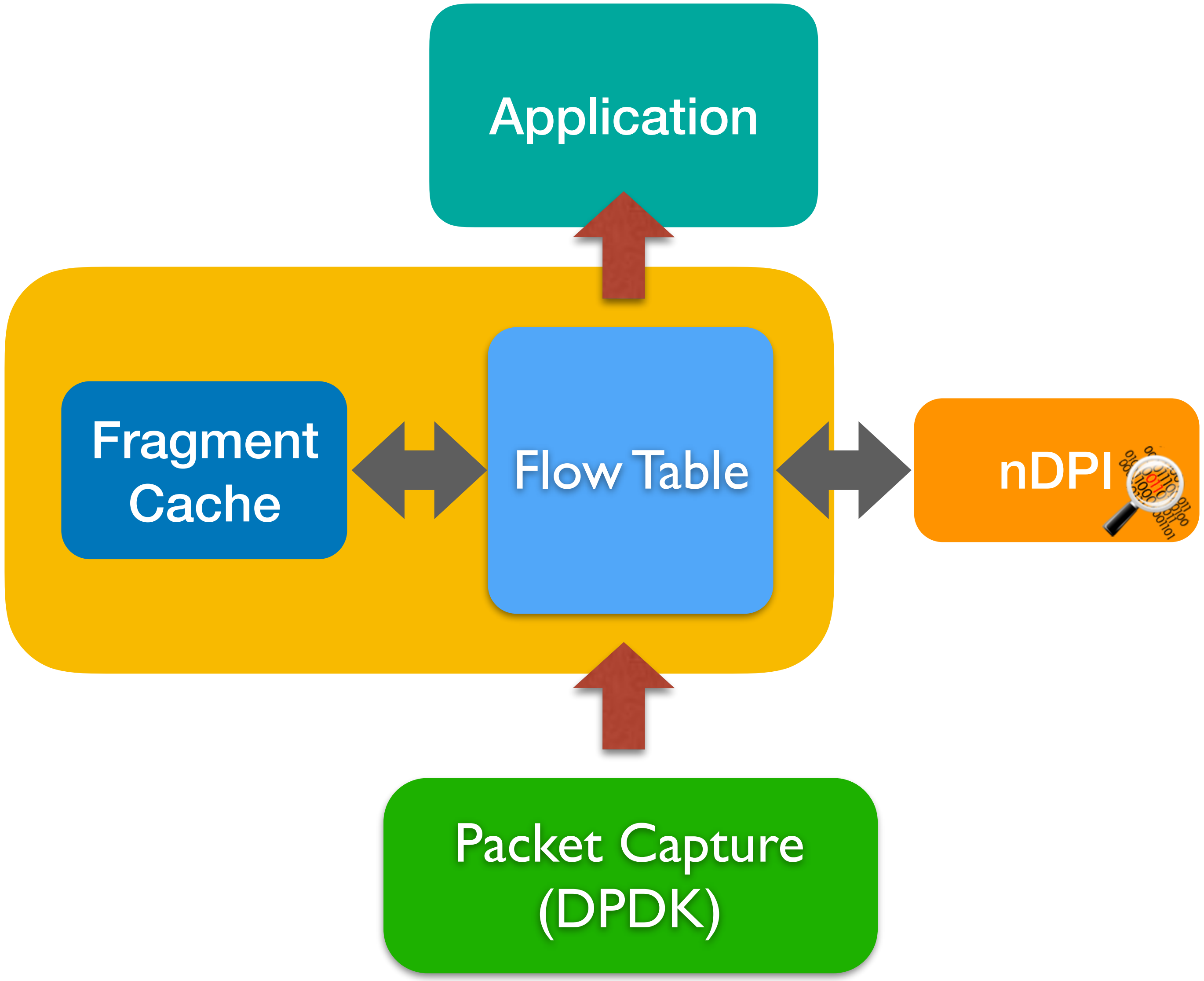
nDPI Internals

- Applications using nDPI are responsible for
  - Capturing (forwarding in inline mode) packets
  - Maintaining flow state.
- Based on flow protocol/port all dissectors that can potentially match the flow are applied sequentially starting from the one that most likely match.
- Each dissector is coded into a different .c file for the sake of modularity and extensibility.
- There is an extra .c file for IP matching (e.g. identify Spotify traffic based on Spotify AS).

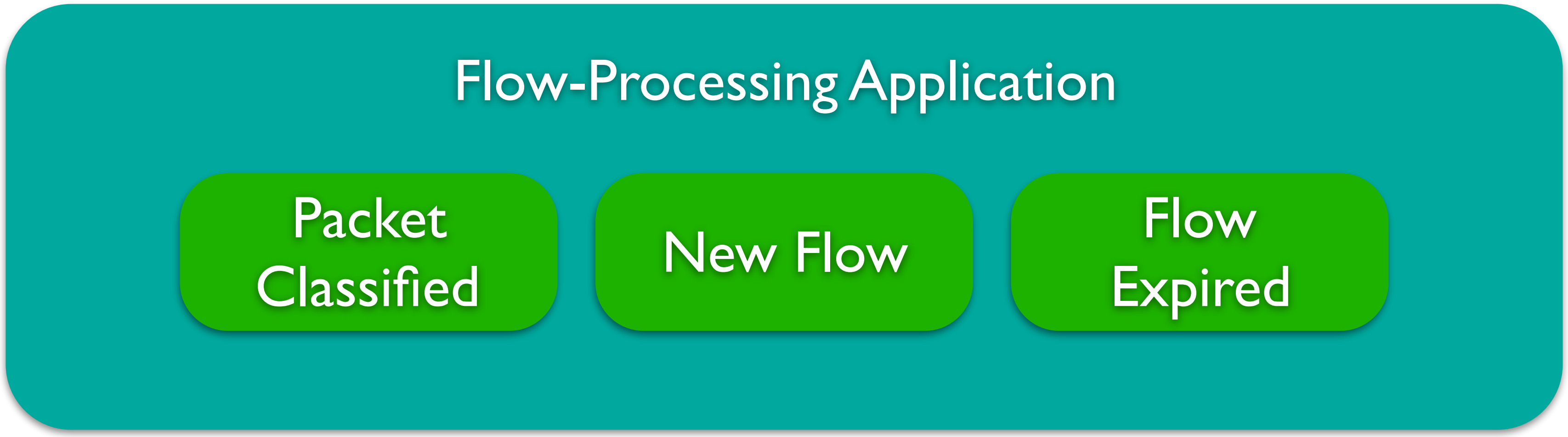
Traffic Classification Lifecycle

- Based on traffic type (e.g. UDP traffic) dissectors are applied sequentially starting with the one that will most likely match the flow (e.g. for TCP/80 the HTTP dissector is tried first).
- Each flow maintains the state for non-matching dissectors in order to skip them in future iterations.
- Analysis lasts until a match is found or after too many attempts (8 packets is the upper-bound in our experience).

nDPI-based Applications: Architecture



Flow Lifecycle [1/2]



- DPI-oriented applications have to deal with flows
- A flow is identified by 5+1 tuple (VLAN, proto, IP/port src/dst).
- It is first created when the first packet is received
- Expires based on timeout or termination (FIN/RST)
- Flow packets are nDPI-processed until the protocol is detected until a max number of iterations (unknown protocol).

Flow Lifecycle [2/2]

- Flows are usually kept in a hash table hashed with the 5-tuple.
- Nasty traffic (e.g. DNS) could cause several collisions that might drive overall the performance down.
- Performance is affected by both Mpps (DPDK) and number of concurrent flows.
- Adding DPI in existing applications (e.g. a traffic monitoring application) must pay attention to flow lifecycle as much as packet processing.

DPDK Integration [1/2]

- nDPI is packet-capture neutral (DPDK, PF_RING, netmap, pcap…)
- Inside nDPI/example there is an application named *ndpiReader* that demonstrates how to use the nDPI API when reading from pcap files and DPDK.

```
$ cd nDPI/example
$ make -f Makefile.dpdk
$ sudo ./build/ndpiReader -c 1 --vdev=net_pcap0,iface=eno1 -- -v 1
```

DPDK Integration [2/2]

```
while(dpdk_run_capture) {
  struct rte_mbuf *bufs[BURST_SIZE];
  u_int16_t num = rte_eth_rx_burst(dpdk_port_id, 0, bufs, BURST_SIZE);
  u_int i;

  if(num == 0) {
    usleep(1);
    continue;
  }

  for(i = 0; i < PREFETCH_OFFSET && i < num; i++)
    rte_prefetch0(rte_pktmbuf_mtod(bufs[i], void *));

  for(i = 0; i < num; i++) {
    char *data = rte_pktmbuf_mtod(bufs[i], char *);
    int len = rte_pktmbuf_pkt_len(bufs[i]);
    struct pcap_pkthdr h;

    h.len = h.caplen = len;
    gettimeofday(&h.ts, NULL);

    ndpi_process_packet((u_char*)&thread_id, &h, (const u_char *)data);
    rte_pktmbuf_free(bufs[i]);
  }
}
```

nDPI-over-DPDK Inline Mode

- You can take any DPDK application and add nDPI support to it

Branch: master ▾ dpdk / examples / skeleton / basicfwd.c

```
for (;;) {
    RTE_ETH_FOREACH_DEV(port) {

        /* Get burst of RX packets, from first port of pair. */
        struct rte_mbuf *bufs[BURST_SIZE];
        const uint16_t nb_rx = rte_eth_rx_burst(port, 0, bufs, BURST_SIZE);

        if (unlikely(nb_rx == 0))
            continue;

        /* nDPI processing code goes here */

        /* Send burst of TX packets, to second port of pair. */
        const uint16_t nb_tx = rte_eth_tx_burst(port ^ 1, 0, bufs, nb_rx);

        /* Free any unsent packets. */
        if (unlikely(nb_tx < nb_rx)) {
            uint16_t buf;
            for (buf = nb_tx; buf < nb_rx; buf++)
                rte_pktmbuf_free(bufs[buf]);
        }
    }
}
```

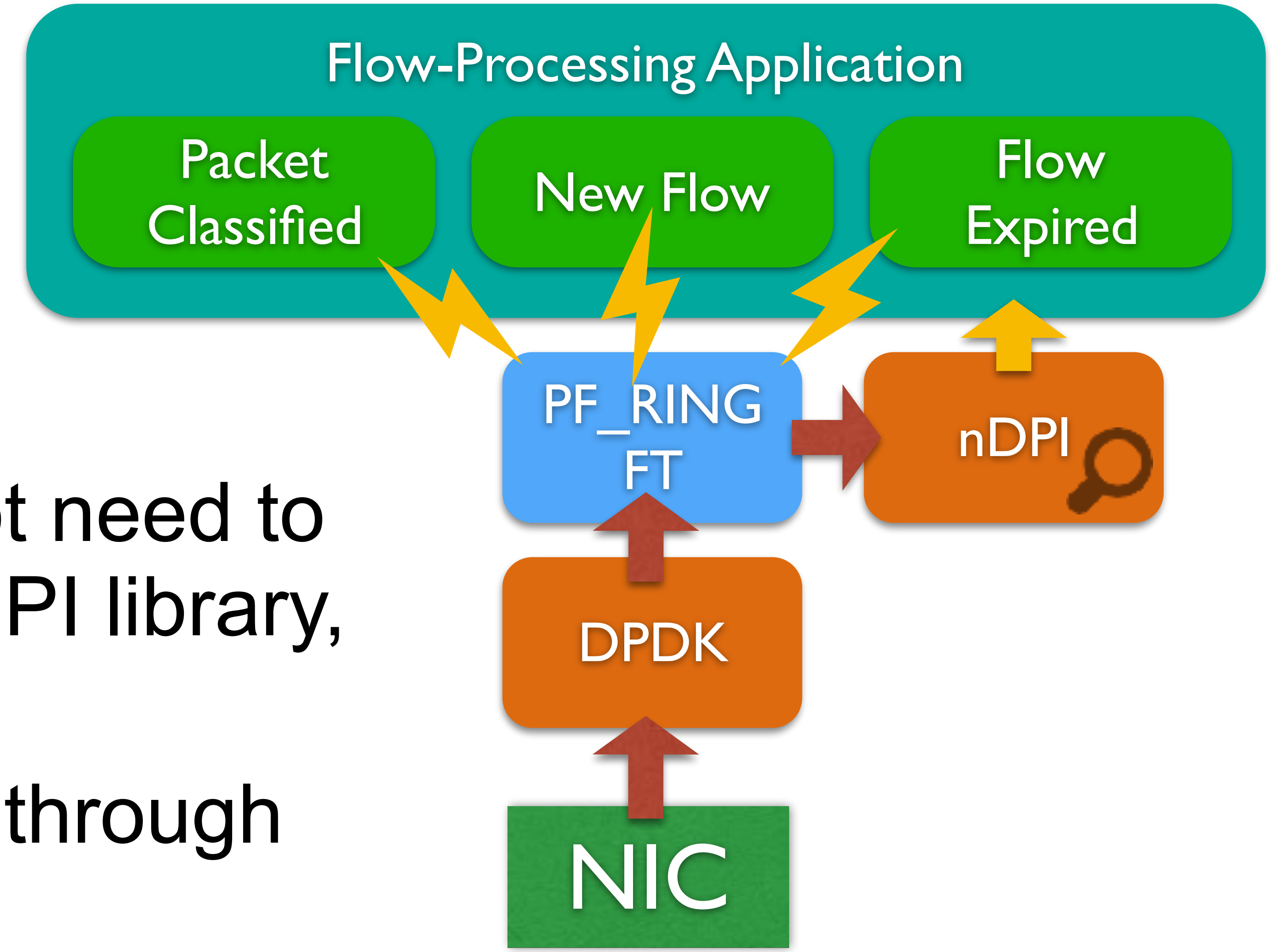
nDPI + PF_RING FT + DPDK [1/3]

- PF_RING FT is natively integrated with nDPI for providing L7 protocol information

- The application does not need to deal directly with the nDPI library, as it:

  1. enables L7 detection through the API

  2. reads the L7 protocol from the exported metadata

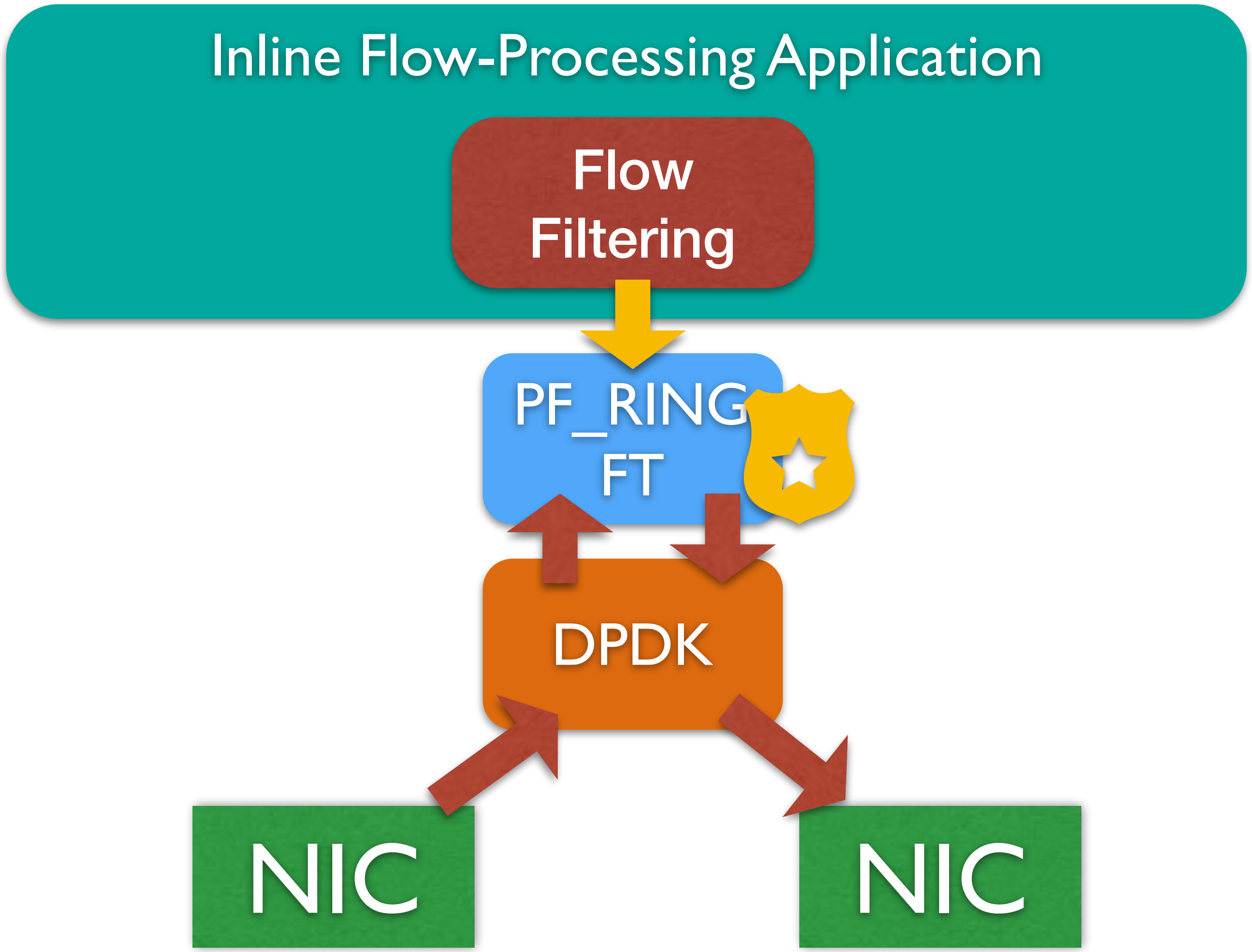


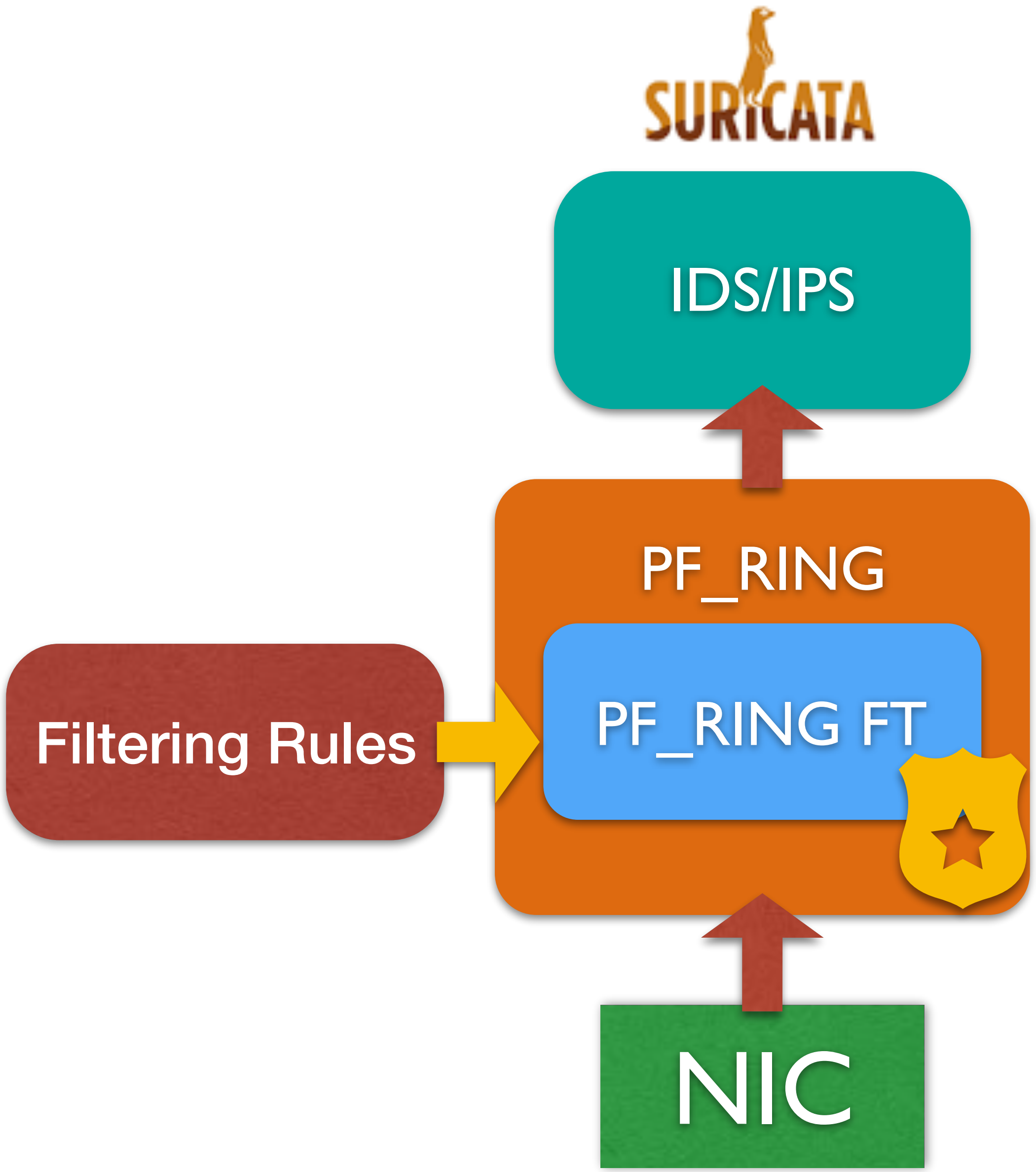
nDPI + PF_RING FT + DPDK [2/3]

```
pfring_ft_table *ft = pfring_ft_create_table(
  flags, max_flows, flow_idle_timeout, flow_lifetime_timeout);

/* Callback for 'new flow' events */
pfring_ft_set_new_flow_callback(ft, new_flow_callback, user);

/* Callback for 'packet processed/classified' events */
pfring_ft_set_flow_packet_callback(ft, packet_processed_callback, user);

/* Callback for 'flow to be exported' events */
pfring_ft_set_flow_export_callback(ft, export_flow_callback, user);

…

/* Process Captured Packets */
while (1) {
  int num = rte_eth_rx_burst(port_id, 0, bufs, BURST_SIZE);
  pfring_ft_pcap_pkthdr h;
  pfring_ft_ext_pkthdr ext_hdr = { 0 };

  for (i = 0; i < num; i++) {
   char *data = rte_pktmbuf_mtod(bufs[i], char *);
   int len = rte_pktmbuf_pkt_len(bufs[i]);

   if(pfring_ft_process(ft, (const u_char *)data, &h, &ext_hdr) != PFRING_FT_ACTION_DISCARD)
      rte_eth_tx_burst(twin_port_id, 0, &bufs[i], 1);
   }
}
```

Full Example: https://github.com/ntop/PF_RING/blob/dev/userland/examples_ft/ftflow_dpdk.c

nDPI + PF_RING FT + DPDK [3/3]





nDPI: Packet Processing Performance: Pcap

```
nDPI Memory statistics:
     nDPI Memory (once):        203.62 KB
     Flow Memory (per flow):    2.01 KB
     Actual Memory:             95.60 MB
     Peak Memory:               95.60 MB
     Setup Time:                1001 msec
     Packet Processing Time:    813 msec

Traffic statistics:
     Ethernet bytes:          1090890957    (includes ethernet CRC/IFC/trailer)
     Discarded bytes:         247801
     IP packets:              1482145       of 1483237 packets total
     IP bytes:                1055319477    (avg pkt size 711 bytes)
     Unique flows:            36703
     TCP Packets:             1338624
     UDP Packets:             143521
     VLAN Packets:            0
     MPLS Packets:            0
     PPPoE Packets:           0
     Fragmented Packets:      1092
     Max Packet size:         1480
     Packet Len < 64:         590730
     Packet Len 64-128:       67824
     Packet Len 128-256:      66380
     Packet Len 256-1024:     157623
     Packet Len 1024-1500:    599588
     Packet Len > 1500:       0
     nDPI throughput:         1.82 M pps / 9.99 Gb/sec
     Analysis begin:          04/Aug/2010 04:15:23
     Analysis end:            04/Aug/2010 18:31:30
     Traffic throughput:      28.85 pps / 165.91 Kb/sec
     Traffic duration:        51367.223 sec
     Guessed flow protos:     0
```

← Single Core (E3 1241v3)

nDPI: Packet Processing Performance: Live Capture

- 10 Gbit tests on Intel E3-1230 v5 3.4GHz DDR4 2133
- 100 Gbit tests on 2x Intel E5-2630 v2 2.6GHz DDR3 1600 (much slower than modern Xeon Scalable)
- nDPI integrated in a flow monitoring application (nProbe Cento)

Traffic	**Capture Card**	**Number of Cores**	**Per Core Performance**	**All Cores Performance**
10 Gbit / 64-byte packets	Intel 10G (X520)	1	14.8 Mpps / 10 Gbps	14.8 Mpps / 10 Gbps
100 Gbit / 1-kbyte packets	FPGA 100G	1	10.8 Mpps / 90 Gbps	10.8 Mpps / 90 Gbps
100 Gbit / 1-kbyte packets	FPGA 100G	**4**	2.8 Mpps / 24 Gbps	11.5 Mpps / **96 Gbps**
100 Gbit / 64-byte packets	FPGA 100G	4	11.2 Mpps / 7.6 Gbps	45.2 Mpps / 30.4 Gbps
100 Gbit / 64-byte packets	FPGA 100G	6 + 6 (2 CPUs)	10.8 Mpps / 7.3 Gbps	**130 Mpps** / 87.6 Gbps

nDPI and Intel HyperScan.io

- Hyperscan is a high-performance regex matching library that can be used in nDPI instead of the native Aho-Corasick (`configure --with-hyperscan`)
- String matching is used in protocol detection.

HyperScan

```
nDPI Memory statistics:
    nDPI Memory (once):         203.62 KB
    Flow Memory (per flow):     2.01 KB
    Actual Memory:              95.60 MB
    Peak Memory:                95.60 MB
    Setup Time:                 1001 msec
    Packet Processing Time:     813 msec
```

Aho-Corasick

```
nDPI Memory statistics:
    nDPI Memory (once):         203.62 KB
    Flow Memory (per flow):     2.01 KB
    Actual Memory:              95.61 MB
    Peak Memory:                95.61 MB
    Setup Time:                 11 msec
    Packet Processing Time:     835 msec
```

Note: same test of slide 23 with HyperScan and Aho-Corasick

Evaluating nDPI

- nDPI has been evaluated both in terms of accuracy and performance.
- “The best accuracy we obtained from nDPI (91 points), PACE (82 points), UPC MLA (79 points), and Libprotoident (78 points)”
- Source: T. Bujlow, V. Carela-Español, P. Barlet-Ros, Comparison of Deep Packet Inspection (DPI) Tools for Traffic Classification, Technical Report, June 2013.

Final Remarks

- We have presented nDPI an open source DPI toolkit able to detect many popular Internet protocols and scale at 10 Gbit on commodity hardware platforms.
- Its open design make it suitable for using it both in open-source and security applications where code inspection is compulsory.
- Code Availability (GNU LGPLv3) https://github.com/ntop/nDPI

Acknowledgment

- I would like to thank the Intel Software Innovator Program for supporting the development of nDPI