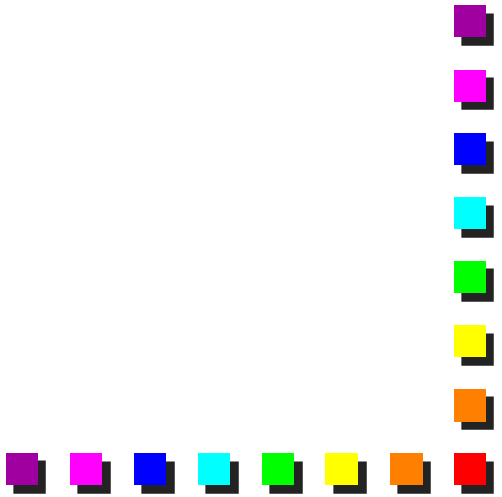
Software-based Packet Filtering

Fulvio Risso

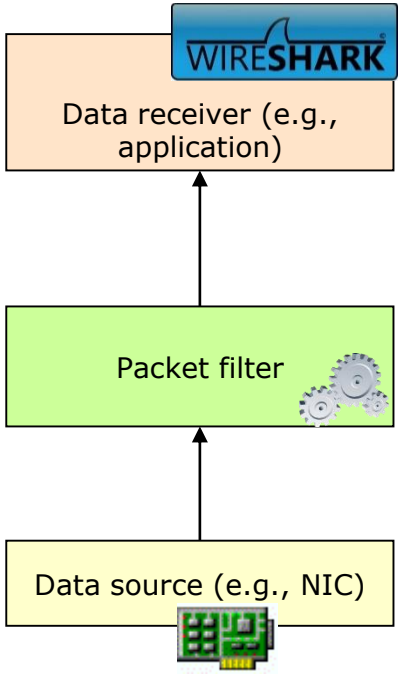
Politecnico di Torino



Part 1:
Packet filtering concepts

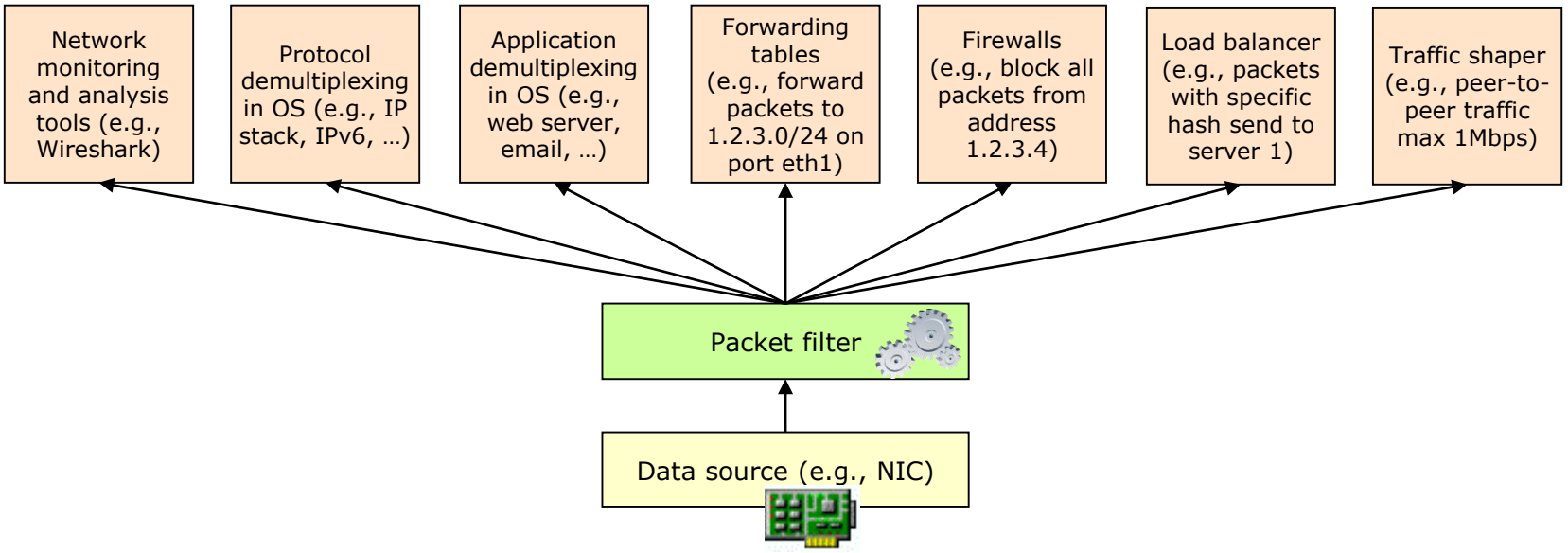
Introduction to packet filters

- A **packet filter** is a system that applies a Boolean function to each incoming packet
  - Needed in all cases in which an application needs to operate ("*filter*") on a subset of the packets coming in
- A **packet classifier** is a system that, given an incoming packet and a *set* of Boolean functions, returns which rules are satisfied
  - Based on packet filtering concepts, although usually implemented in a different form

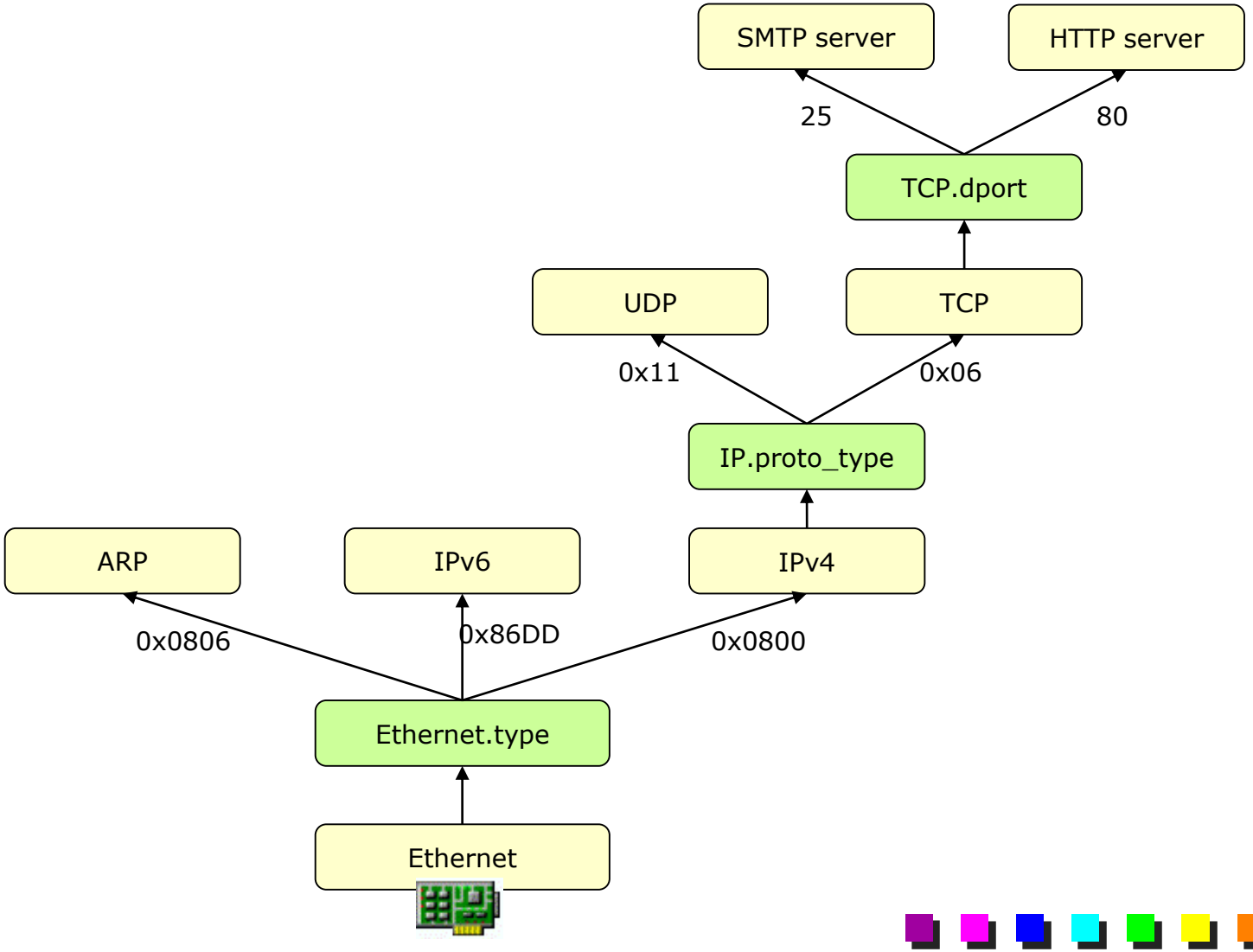


Possible applications of packet filters

- Packet filtering is a very general concept, widely used in the networking field



PF example: OS/application demultiplexing



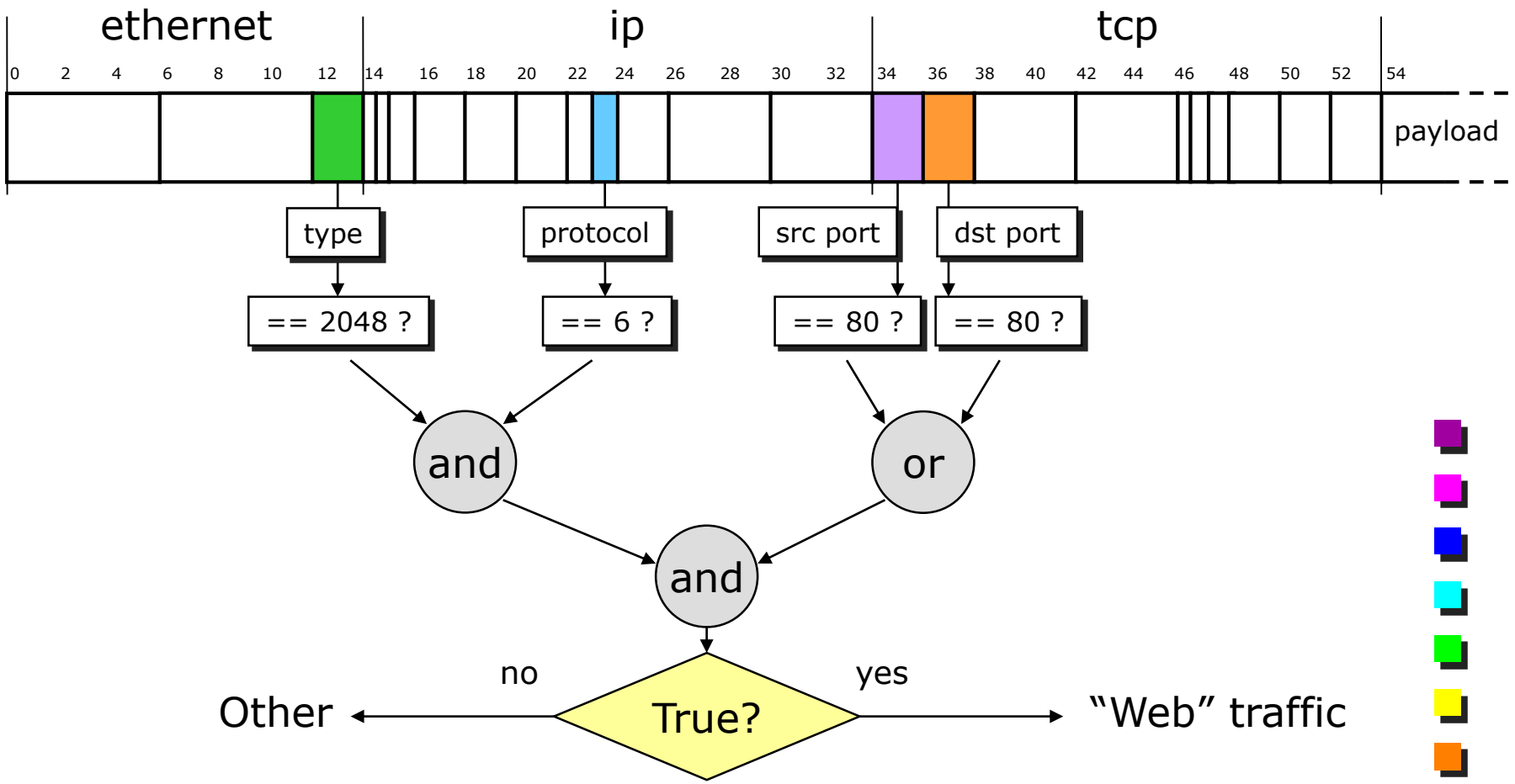
Packet filtering implementations

- The technology used to implement this function may differ based on the function we have in mind
  - Classical packet filtering, based on special purpose virtual machines, for packet capture and network monitoring
  - Optimized classification algorithms for forwarding processes
  - Static filters for protocol/application demultiplexing
  - Etc.

- The remaining of this presentation will focus on classical packet filters

Packet filtering example: “web” traffic

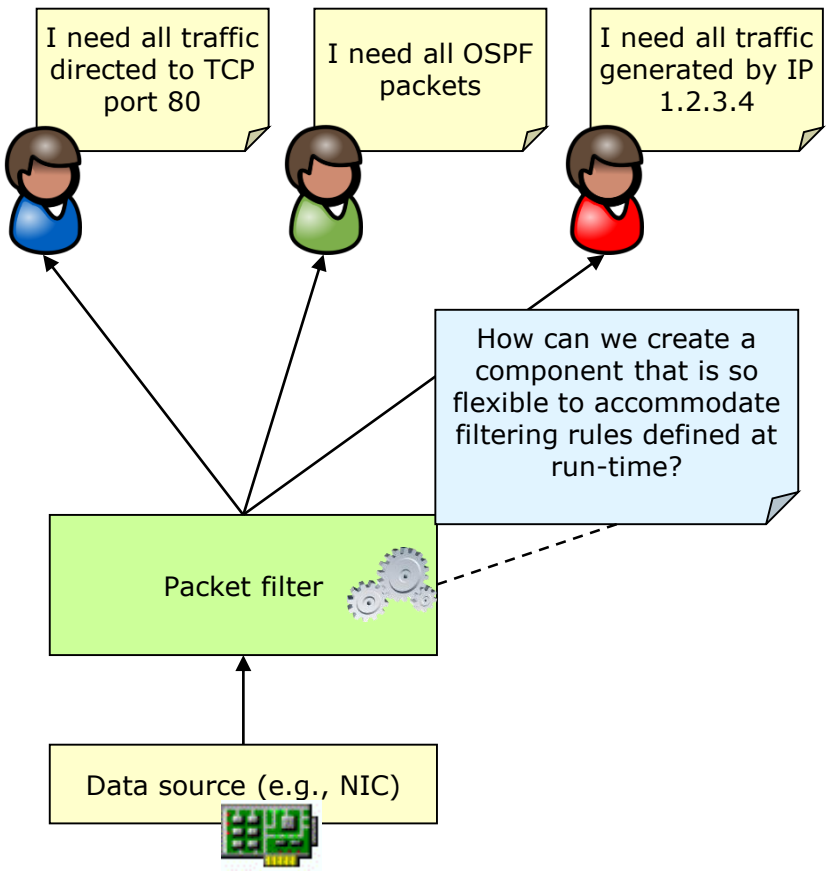
Web traffic: ip - tcp - port 80



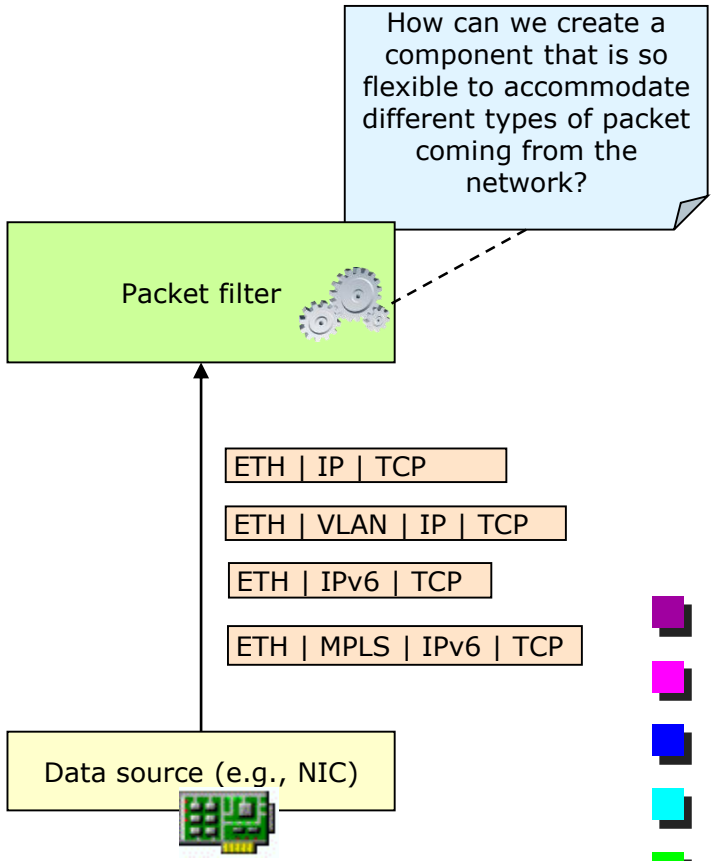
Requirements of packet filters

- Flexibility
  - Need to handle filters specified *dynamically*, at run-time
  - Need to adapt *dynamically* to network data that comes with different frame/packet format (e.g., plain Ethernet, VLAN tagged)
- Security/Safety
  - Need to be flexible enough but avoid security hazards
  - Often, packet filtering is implemented in the OS kernel
- Efficiency
  - The traffic to be analyzed may be huge, we cannot spend too much time per each packet
- Composability
  - We may need to run several filters in parallel, as we would like to avoid the sequential execution of the packet filter
- Update speed
  - Cannot wait for hours when the filter need to be updated
  - E.g., filtering over (dynamic) TCP sessions (firewall)

Packet filters and the need for flexibility



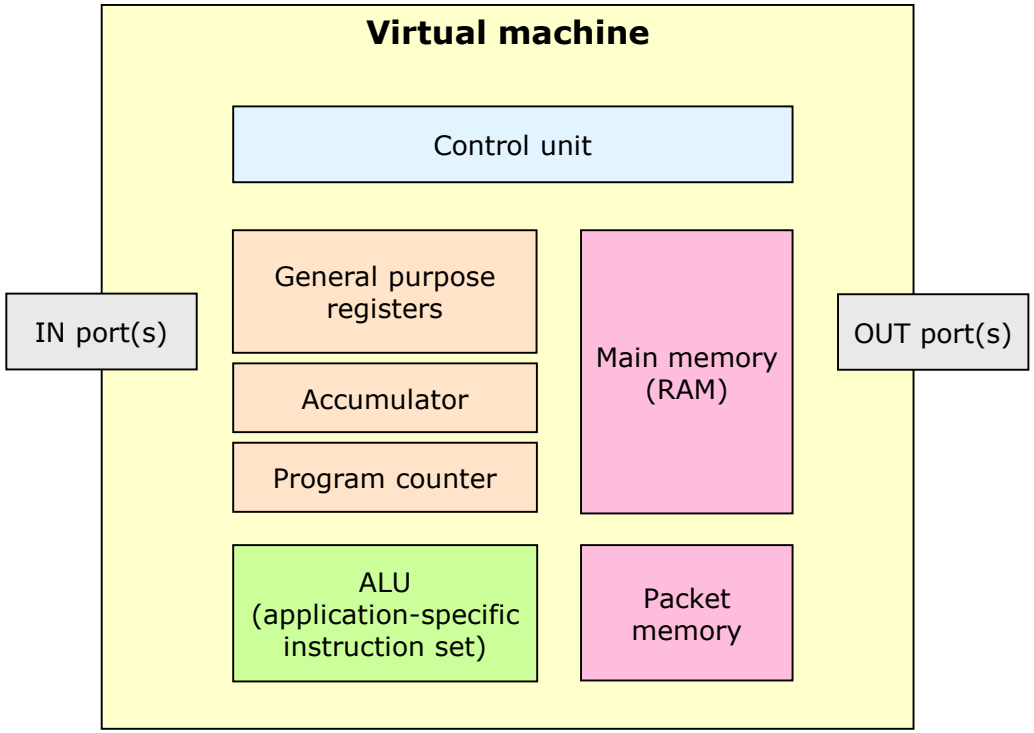
Flexibility as requirement coming from applications



Flexibility as requirement coming from traffic heterogeneity

Special purpose virtual machine

- Definition of an-hoc execution environment specially crafted for packet filtering purposes
  - E.g., specific memory for packet (not just the main RAM)




```
Sample code
(from BPF virtual machine)

Filter: "ip" (with simple
Ethernet frames)

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

Special purpose VM vs full-fledged VM

Special purpose VM

- Software architecture that emulates a specific HW component (e.g., special purpose CPU) and that is defined to solve a specific problem (e.g., packet filtering)
- Much easier to emulate
  - Just the HW, no need to support unmodified Operating Systems
- VMs for packet filtering belong to this domain
  - Actually, several types of implementation are possible

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

Virtual Machine as an interpreter

```
// Example of a register-based virtual machine
while (ProgramCounter <= FilteringInstructions)
{
  currInstruction= instruction[ProgramCounter];
  switch(currInstruction.opcode)
  {
    case LOAD_MEM32:
    {
      if (CheckForMemOffset(currInstruction.memOffset) == false)
          break;
      RegisterEAX= Memory[currInstruction.memOffset];
    };
    break;

    // … Other instructions here
    default:
    {
        // Raise exception
    }
  }

  ProgramCounter++;
}
```

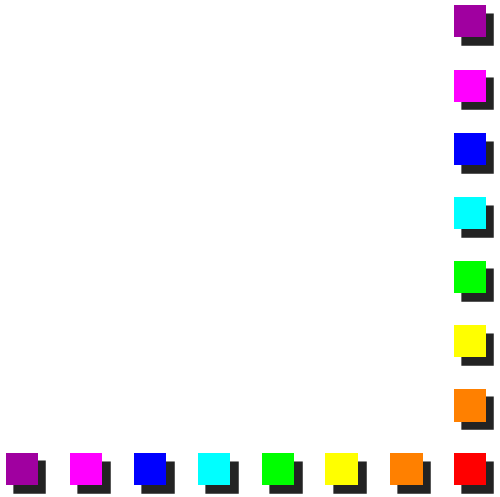
Some filtering examples

```
user@linux$ tcpdump -d ip
tcpdump: listening on \
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 3
(002) ret      #96
(003) ret      #0

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

user@linux$ tcpdump -d tcp
tcpdump: listening on \
(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
```

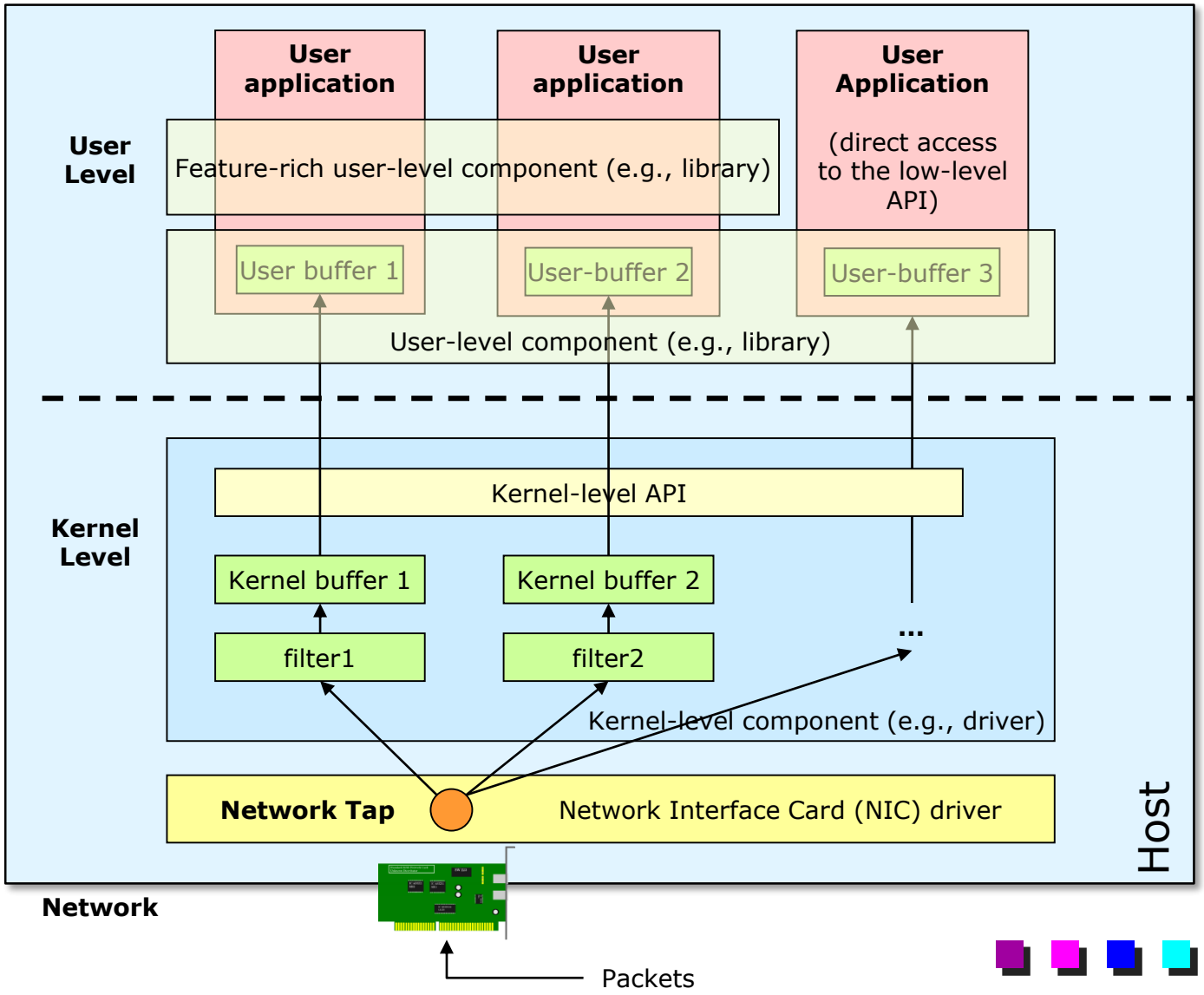
VMs and safety

- The bytecode (*opcodes*) is valid
  - Controlled by the existence of a "default" branch in the switch
- The jump/branch destinations are valid
  - Controlled with appropriate checks in the interpreter
- Finite number of instructions
  - Controlled with appropriate checks before starting the interpreter
- Reading and writing from/to a valid memory address
  - Controlled with appropriate checks in the interpreter
- Termination of the program guaranteed
  - A possibility can be by not defining some instructions (e.g., backward jumps, which forbid loops)
  - Some more clever way require ahead-of-time static inspection of the program, which is rather complex (formal verification of source code)
- Finite and predictable memory consumption

Part 2:
Software architectures for packet filtering



Typical packet capture architecture

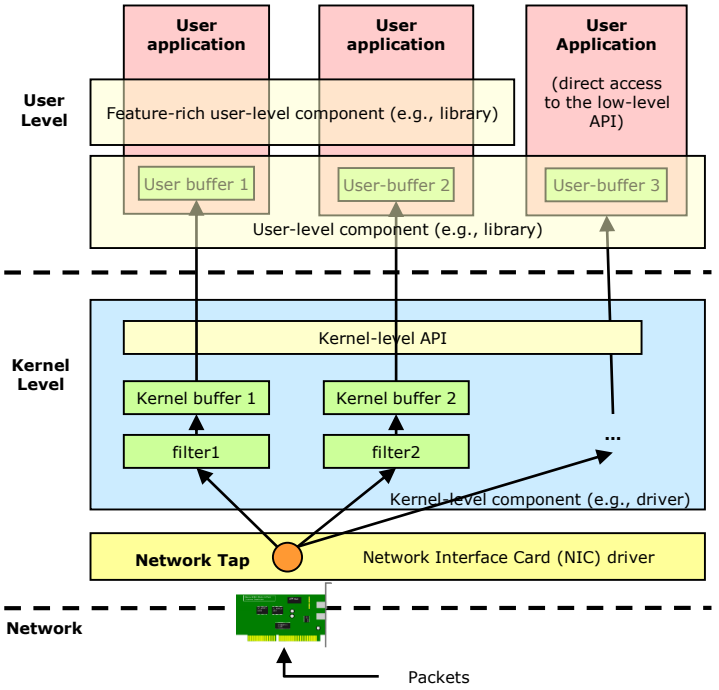


User vs. kernel processing in packet filters

- User processing is easier
  - Easy to create, install, operate software
  - More portable
  - Less risky: a program that crashes does not corrupt the entire system
- Kernel-processing is faster
  - Avoids the cost of context switch between kernel and user space
- Packet filters
  - We need a mechanism that performs the most basic operations at kernel-level, allowing to transfer to the applications only the packets that require further processing, which can be done in user-space

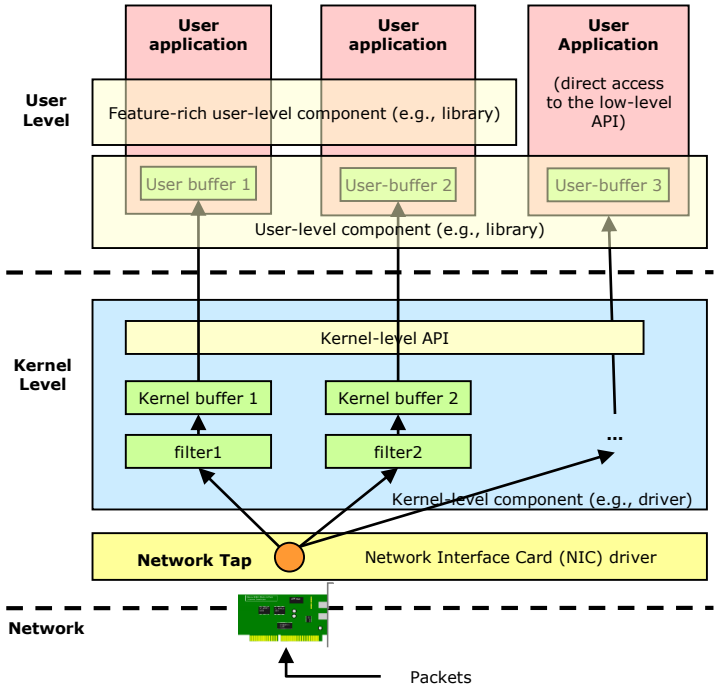
Network tap

- Component that intercepts packets from the NIC and delivers them to the packet capture components
- Different options
  - Windows: sits on top of the NIC drivers, declaring itself as a new layer-3 protocol
  - BSD: NIC drivers are patched with proper explicit calls to the capture components



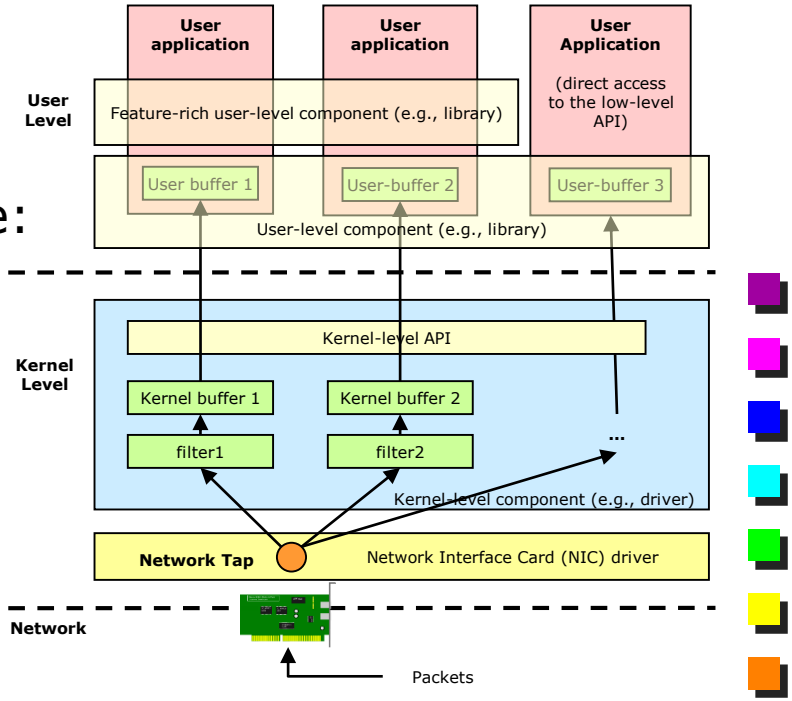
Kernel packet filter

- Component that discards unwanted packets, for efficiency reasons
  - The earlier you discard non-interesting packets, the better it is
- Only interesting packets are copied in the kernel buffer
  - So far, the packet has never been copied by the packet capture stack
  - Although both NIC and the OS may already have done some copies to that packet



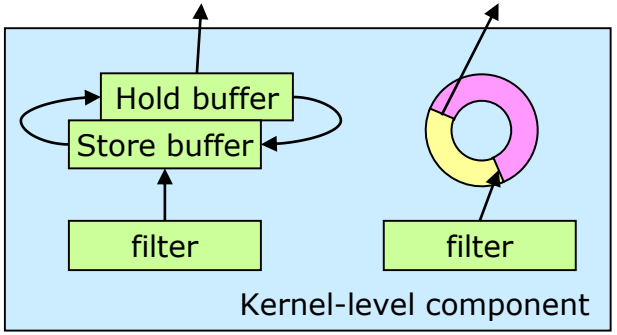
Kernel buffer

- Component that stores packets before delivering them to the application
  - Kernel buffer is one of the key components that allows batch processing (several packets copied at once in user space)
  - First copy performed by the packet capture framework
- Different architectures are possible: tradeoff between memory and CPU efficiency (see next slide)



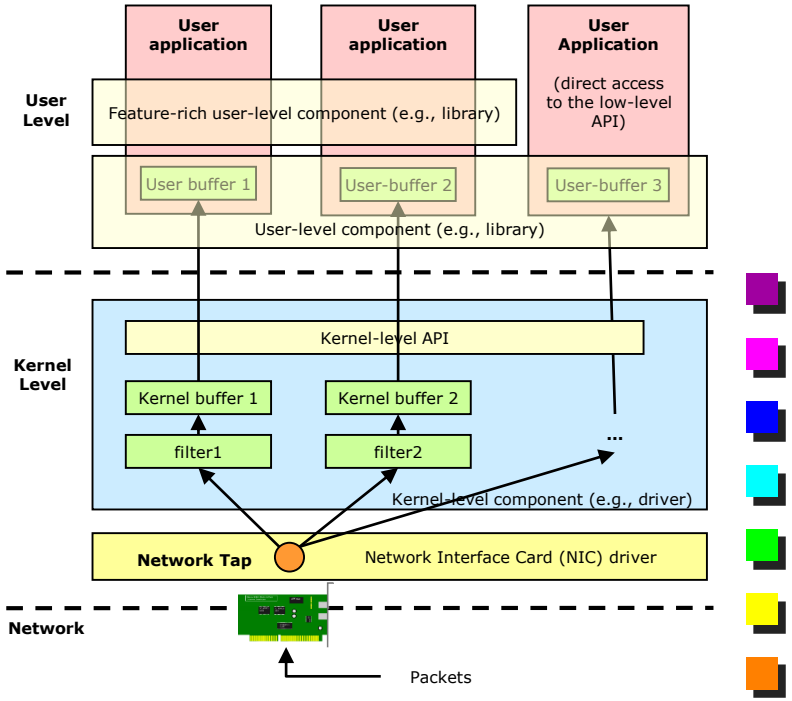
Kernel buffer (2)

- Hold/Store buffers
  - More CPU efficient, but only half the space is used for storing packets
  - The kernel-level and the user-level processes, running in parallel on different CPU cores, operate on two different memory areas, hence no cache pollution
  - No need of per-packet synchronization between the two processes
    - Sync primitives need only when buffers are swapped
- Circular buffer
  - More memory efficient
  - Requires locks for updating packet pointers in the shared buffer
  - More possibility to have cache pollution among the different CPU cores
    - Shared variables must be in both caches
    - Memory area is shared among CPUs



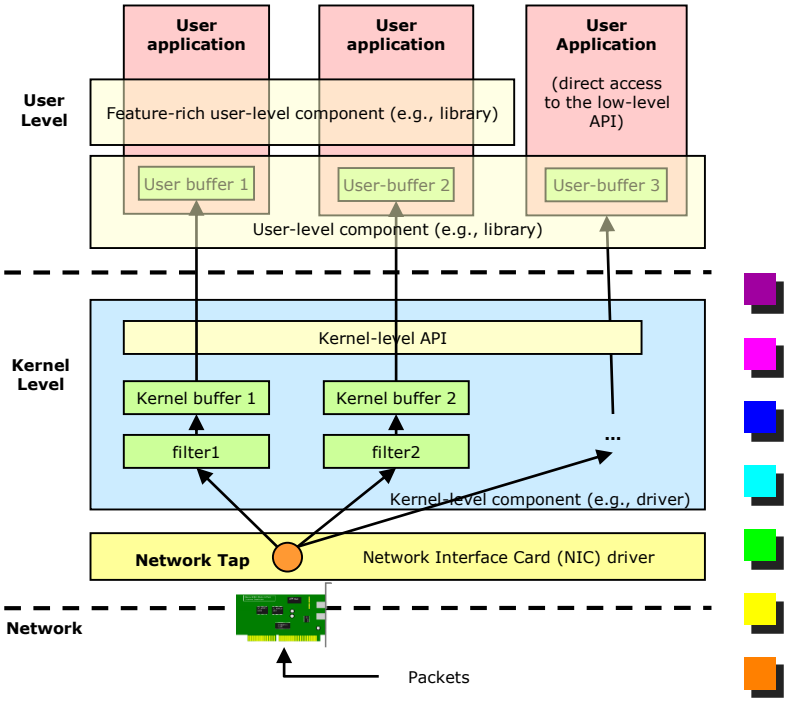
Kernel-level API

- Provides the necessary primitives to interact with the kernel-level components
  - Get access to the data stored in the buffer
  - Inject the packet filter
  - Bind he tap to the desired NIC
  - Etc.
- Often made with simple IOCTL

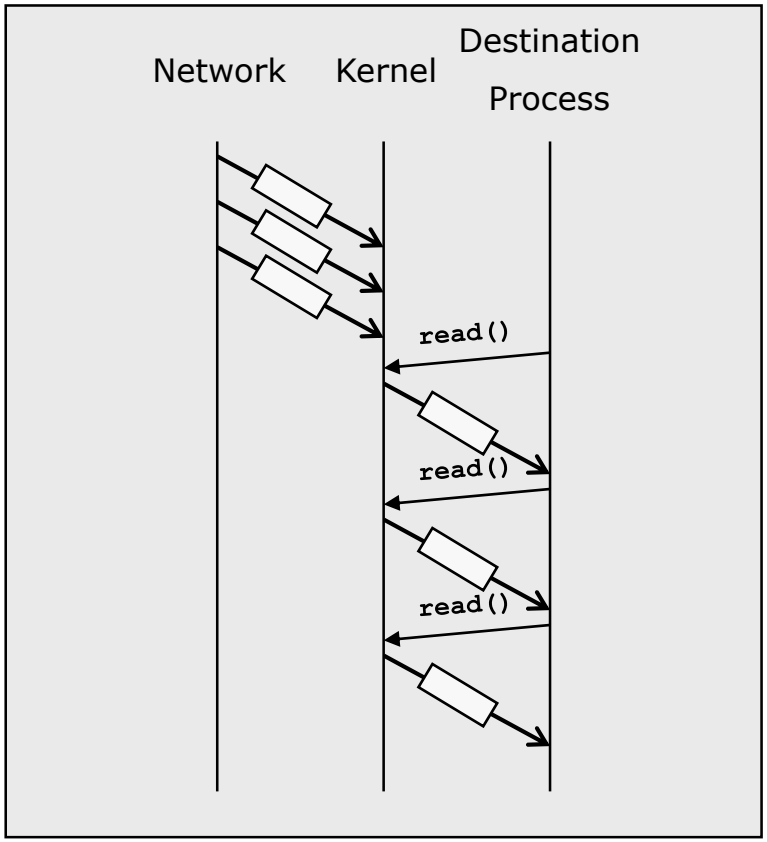


User buffer

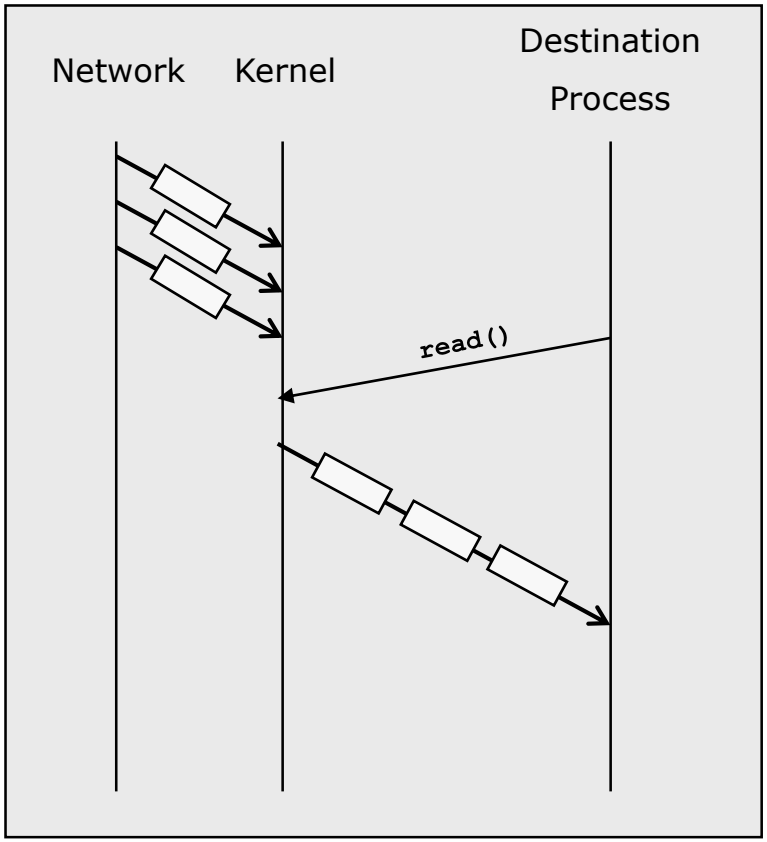
- Stores packets at the user-level
- Needed to enable batch processing, which transfers multiple packets with a single call to the kernel
  - Reduces the number of kernel/user contexts switches
  - Cache efficient because multiple packets are copied in a row
- Resides in the address space of the application



Kernel buffers and batch-processing



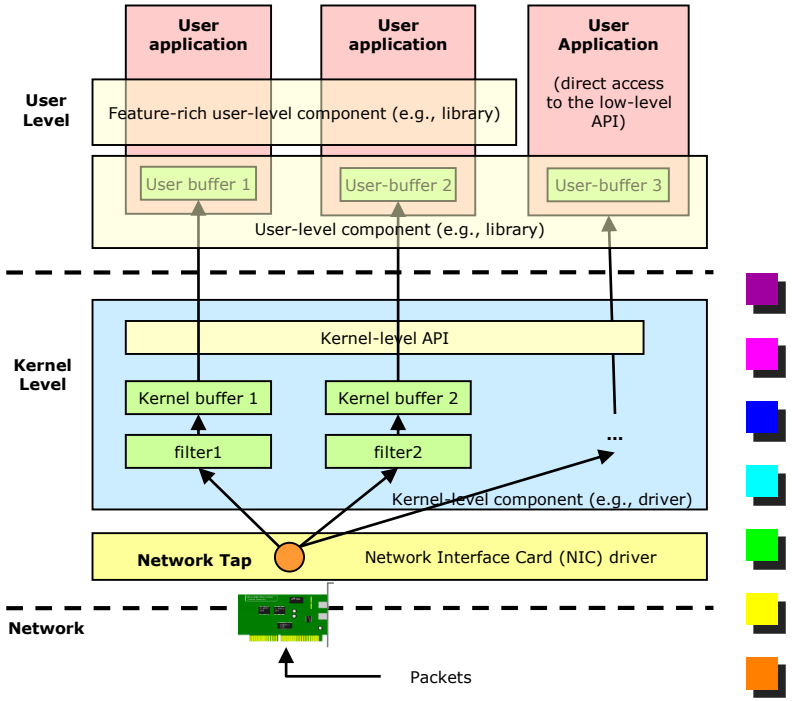
Delivery without packet-batching



Delivery with packet-batching

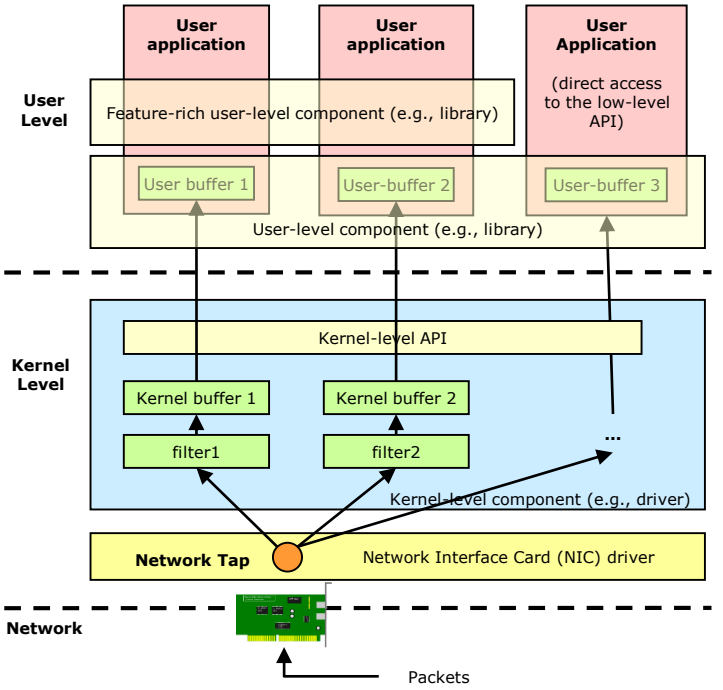
User-level API

- Exports useful functions to get access to the underlying packet capture framework, such as:
  - Read packet
  - Set packet filter
  - Set NIC in promiscuous mode
  - …
- In general, it provides access to kernel-level functions
  - Those functions are often mapped to IOCTL calls



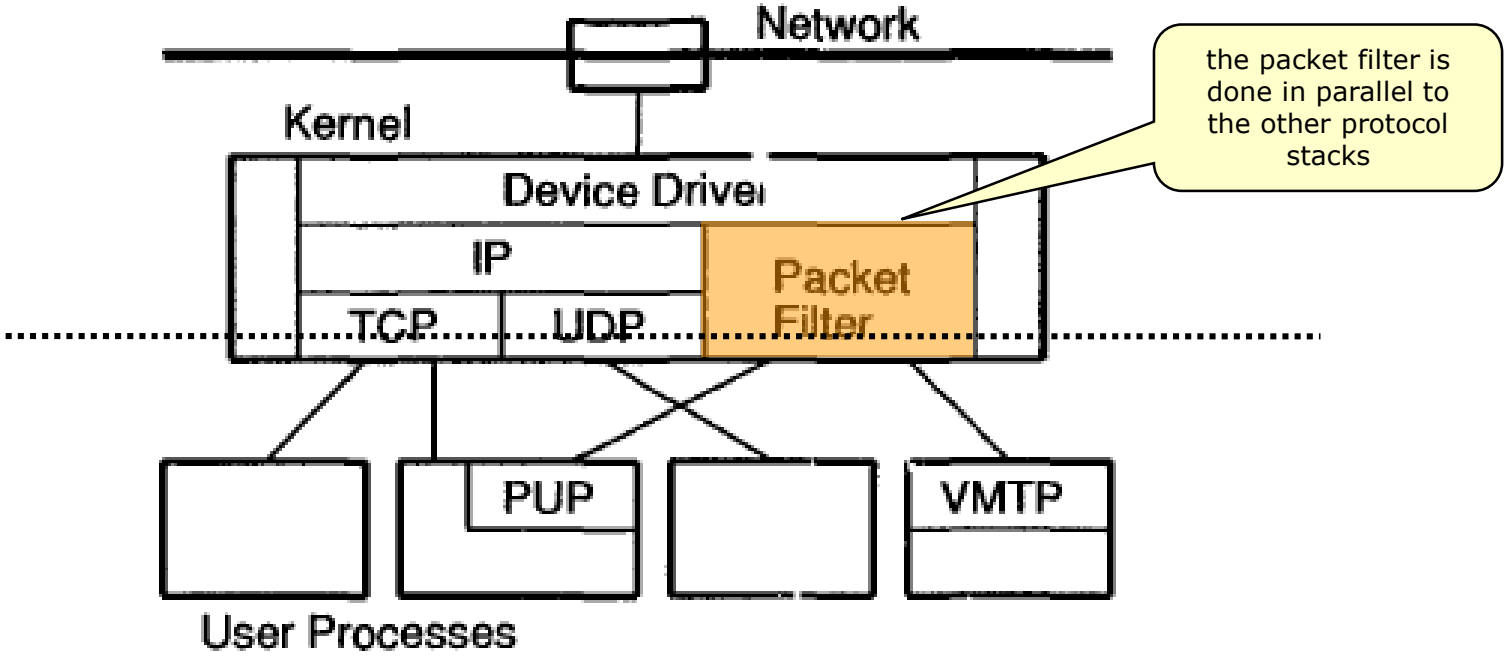
Feature-rich user-level component

- Exports (optional) additional functionalities, such as:
  - High-level compiler to create packet filtering code (e.g., from "ip.src=1.1.1.1" to the proper set of assembly instructions)
- Can provide uniform access to the underlying components across different operating systems
  - E.g., WinPcap/libpcap



The first packet filter: CSPF (CMU/Stanford Packet Filter)

- Interesting ideas
  - Implementation at kernel-level
  - Batch processing
  - Virtual Machine

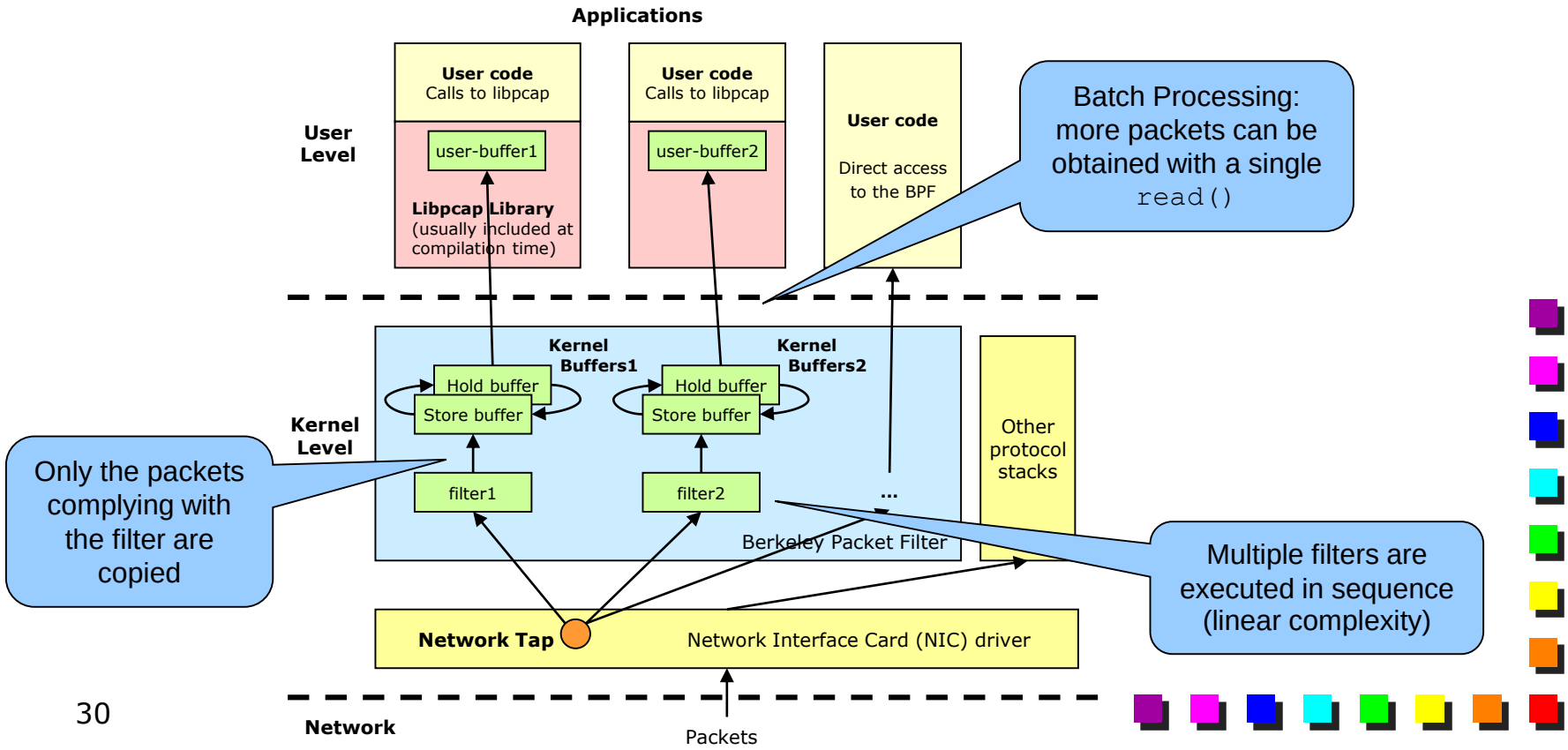


Libpcap/WinPcap

- Provides three fundamental services
  - Abstraction of the physical interface on which it works
  - Creation of a filtering expression from a high-level language
  - Abstraction of the filtering mode implemented in that particular system (in Kernel, in user space, etc.)
- Open source (BSD license), available for (almost) all operating systems
- It requires a set of kernel-level components to get access to the raw packets

Berkeley Packet Filter

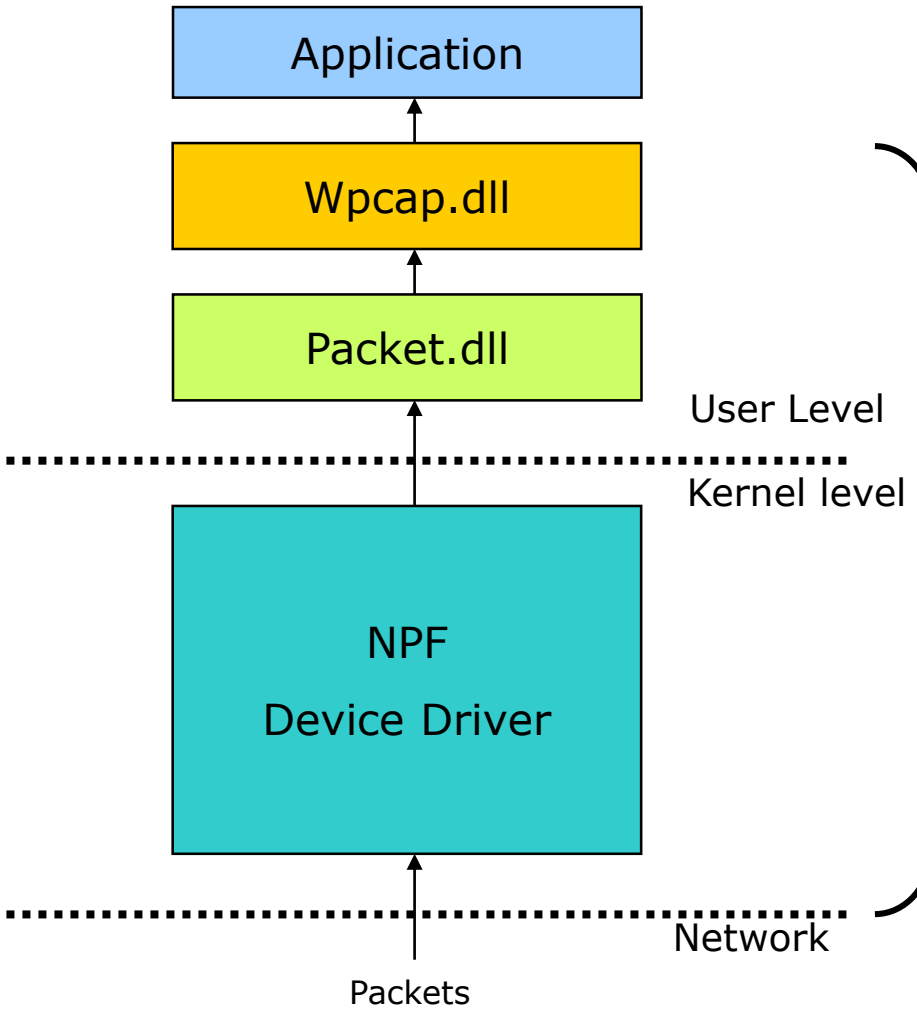
- BPF is the first serious implementation of a packet filter and it is still used today
- Small buffers
- Coupled with the libpcap library in user space



WinPcap

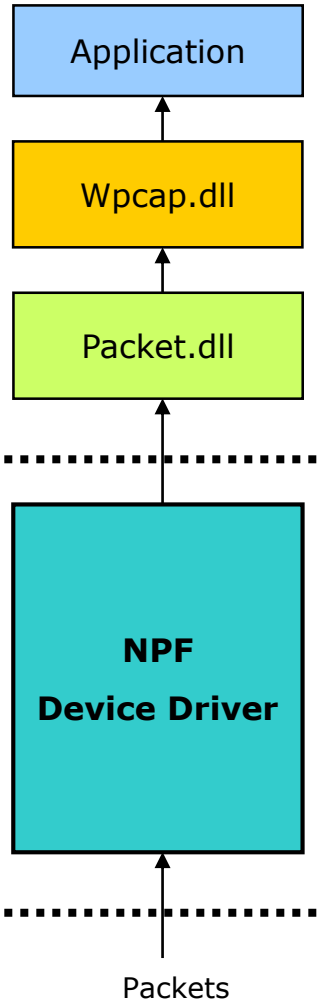
- Can be considered a porting of the entire BPF/libpcap architecture on Windows
- Complete porting of the libpcap API
  - Libpcap is integrated in one of the user-level components of WinPcap (wpcap.dll)
- Adds some functionalities not available in libpcap/BPF
  - Statistics Mode: module programmable by the user to register statistical data in the kernel without changing the context
  - Packets Injection: allows to send packets through the network interface
  - Remote Capture: is possible to activate a remote server for capturing packets (rpcapd), which delivers the captured packets to a local workstation

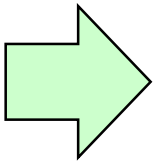
WinPcap: architecture

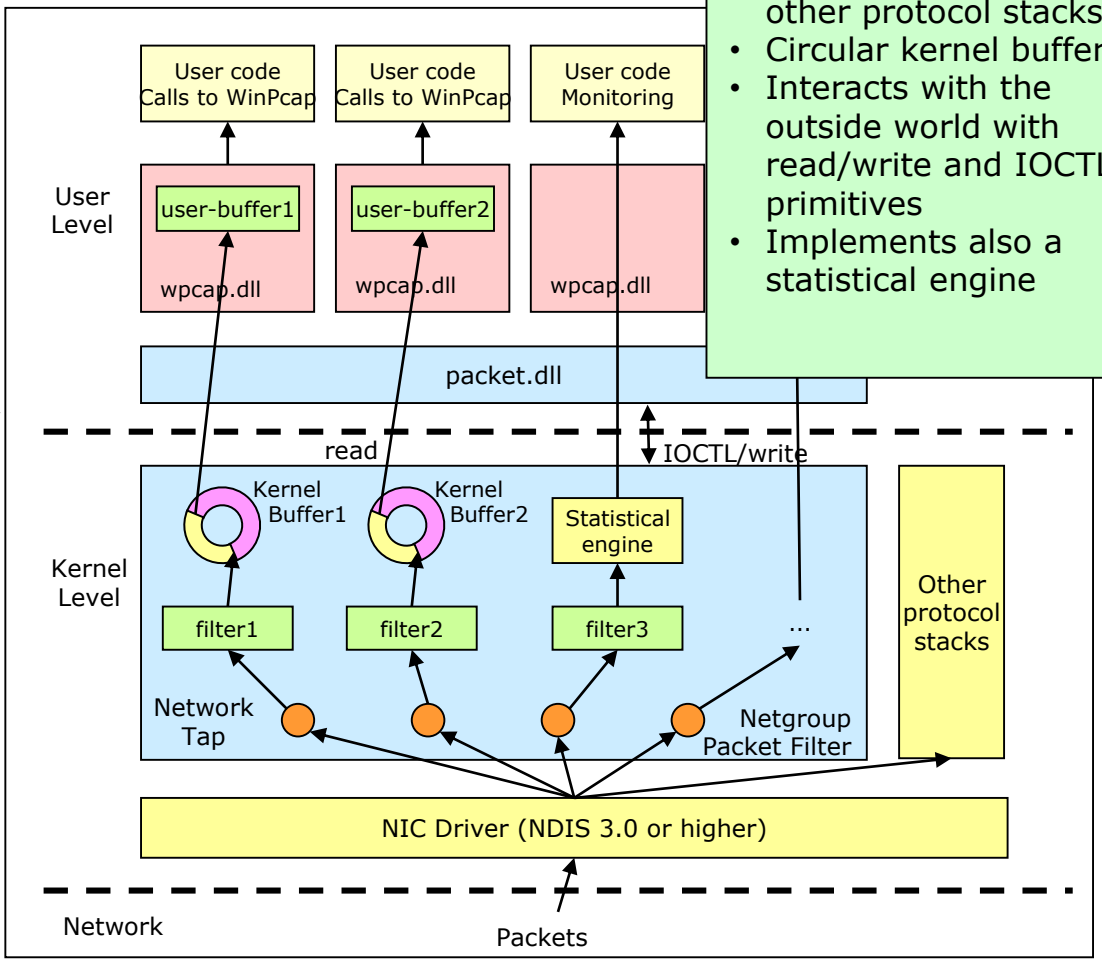


WinPcap implements exactly the logical components already presented in the previous slides, organized in the three modules shown here

NPF: Netgroup Packet Filter

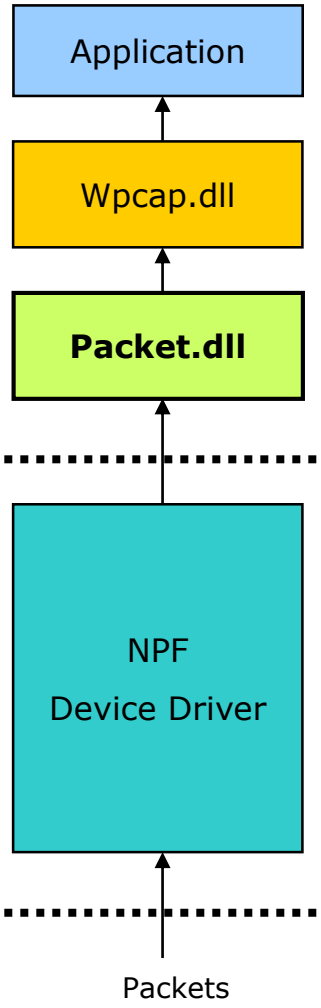


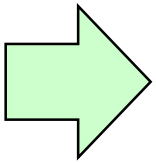


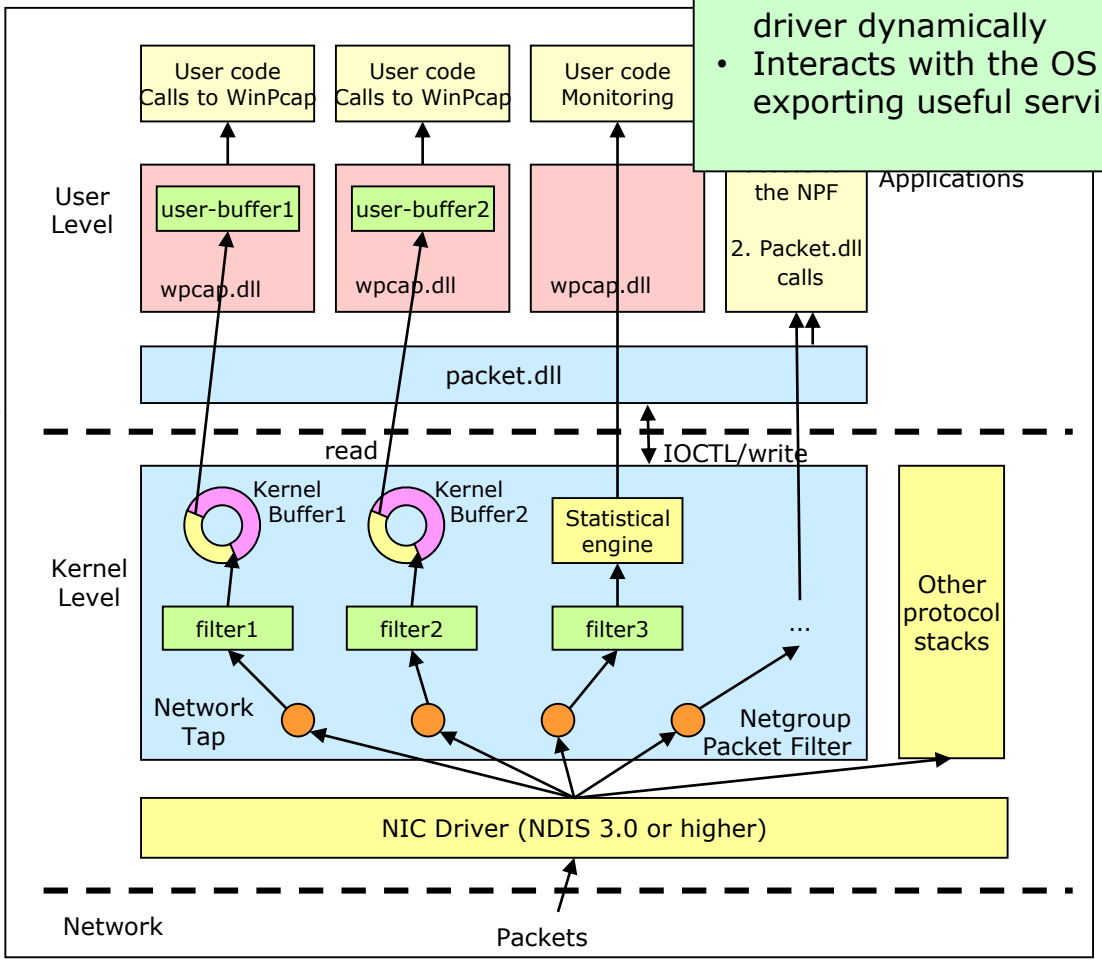


- Implement the kernel-portion of the capture stack, in parallel to other protocol stacks
- Circular kernel buffer
- Interacts with the outside world with read/write and IOCTL primitives
- Implements also a statistical engine

Packet.dll

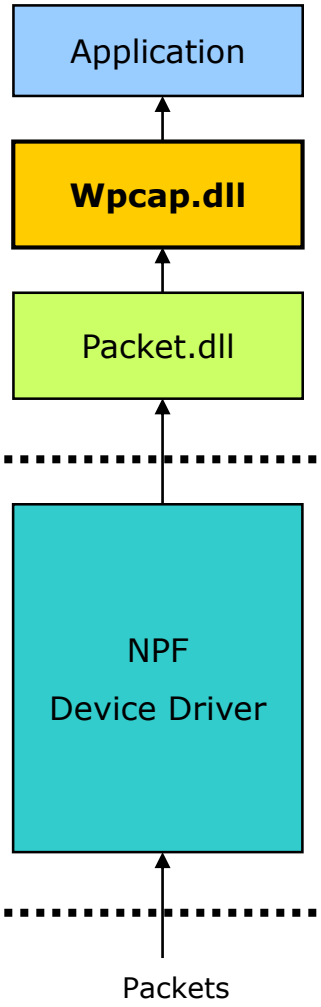


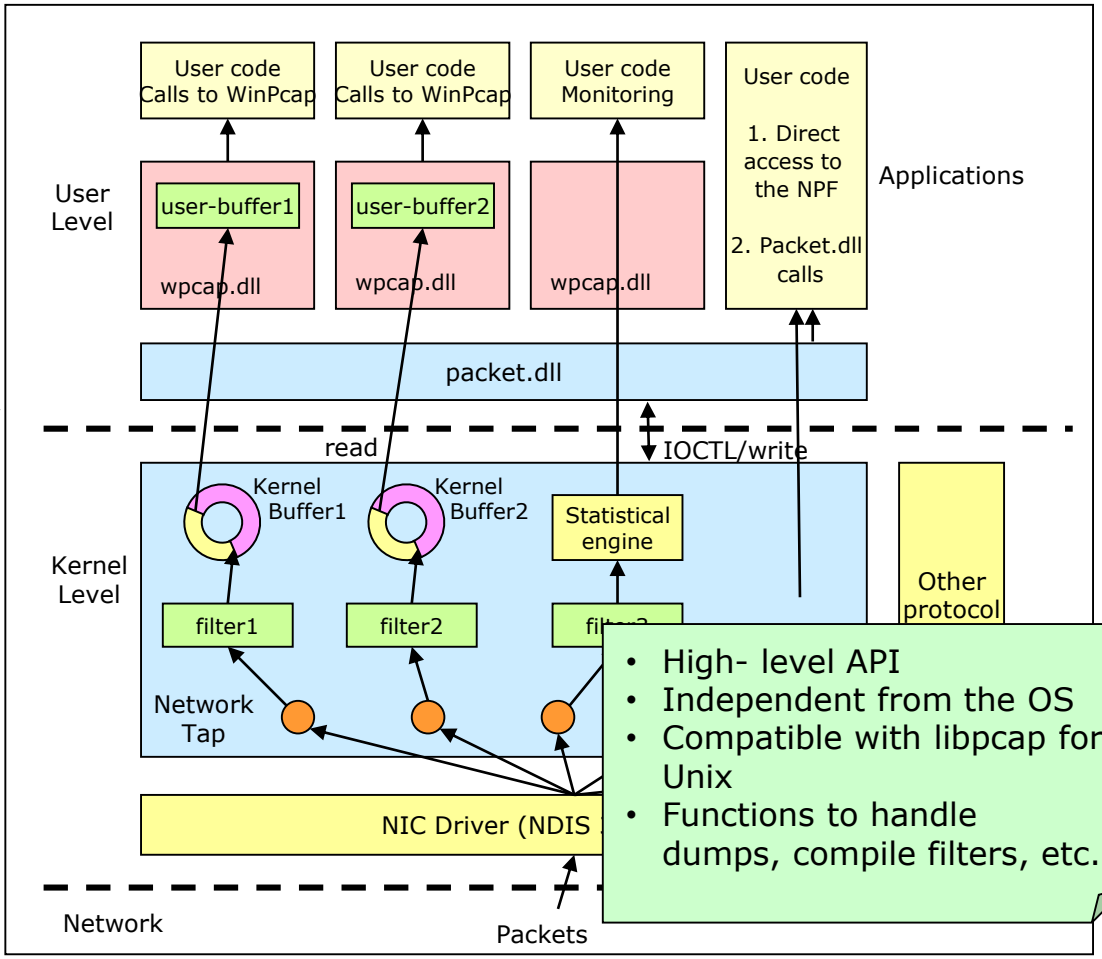




- Enables the independence from the OS
- Installs and handles the driver dynamically
- Interacts with the OS exporting useful services

Wpcap.dll


Application
Wpcap.dll
Packet.dll
NPF
Device Driver
Packets


User code
Calls to WinPcap
User code
Calls to WinPcap
User code
Monitoring
User code
1. Direct access to the NPF
2. Packet.dll calls
Applications
User Level
user-buffer1
user-buffer2
wpcap.dll
wpcap.dll
wpcap.dll
packet.dll
read
IOCTL/write
Kernel Level
Kernel Buffer1
Kernel Buffer2
Statistical engine
Other protocol
filter1
filter2
Network Tap
NIC Driver (NDIS
Network
Packets
• High- level API
• Independent from the OS
• Compatible with libpcap for Unix
• Functions to handle dumps, compile filters, etc.

Mayor improvements of WinPcap

- JIT compiler
  - Later integrated in BPF and Linux as well
  - x10 performance improvements with respect to the interpreted code
  - A very primitive technology anyway
    - It is in reality an instruction translator, more than a real JIT
- Optimized processing
  - Not only the packet filter, but the whole filtering stack
- Shared buffer instead of hold/store buffers

WinPcap JIT: example

```
while (ProgramCounter <= FilteringInstructions)
{
  currInstruction= instruction[ProgramCounter];
  switch(currInstruction.opcode)
  {
    case LOAD_MEM32:
    {
      // Check that Offset exists
      Copy("mov EAX, ", currInstruction.memOffset);
      Copy("cmp EAX, MaxMemOffset");
      Copy("jle EXCEPTION");

      // Save the value in the "EBX" register
      Copy("mov EBX," currInstruction.memOffset);
    };
    break;

    // … Other instructions here
    default: // Raise exception
  }

  ProgramCounter++;
}
```

JIT Translator vs JIT compiler and optimizer

```
// Sample inspired to 'tcpdump -d tcp'
(000) ldh      [offset_ethertype]
(001) jeq      #0x86dd          jt 2    jf 4
(002) ldb      [length_ether + offset_ipv6_protocol_type]
(003) jeq      #0x6             jt 7    jf 8
(004) jeq      #0x800           jt 5    jf 8
(005) ldb      [length_ether + offset_ipv4_protocol_type]
(006) jeq      #0x6             jt 7    jf 8
(007) ret      #96
(008) ret      #0
```

In general, JIT translators are not able to globally optimize the code.

This is just an example of the difference between the two technologies.

Pseudo-code generated by a JIT translator

```
// Add instruction to check that offset_ethertype is valid
(000) ldh      [offset_ethertype]
(001) jeq      #0x86dd          jt 2    jf 4
// Add instruction to check that length_ether + offset_ipv6_protocol_type is valid
(002) ldb      [length_ether + offset_ipv6_protocol_type]
(003) jeq      #0x6             jt 7    jf 8
(004) jeq      #0x800           jt 5    jf 8
// Add instruction to check that length_ether + offset_ipv4_protocol_type is valid
(005) ldb      [length_ether + offset_ipv4_protocol_type]
(006) jeq      #0x6             jt 7    jf 8
(007) ret      #96
(008) ret      #0
```

Pseudo-code generated by a JIT compiler and optimizer

```
// Add instruction to check that max(offset_ethertype, length_ether +
// offset_ipv6_protocol_type, length_ether + offset_ipv4_protocol_type) is valid
(000) ldh      [offset_ethertype]
(001) jeq      #0x86dd          jt 2    jf 4
(002) ldb      [length_ether + offset_ipv6_protocol_type]
(003) jeq      #0x6             jt 7    jf 8
(004) jeq      #0x800           jt 5    jf 8
(005) ldb      [length_ether + offset_ipv4_protocol_type]
(006) jeq      #0x6             jt 7    jf 8
(007) ret      #96
(008) ret      #0
```

Safety with JIT

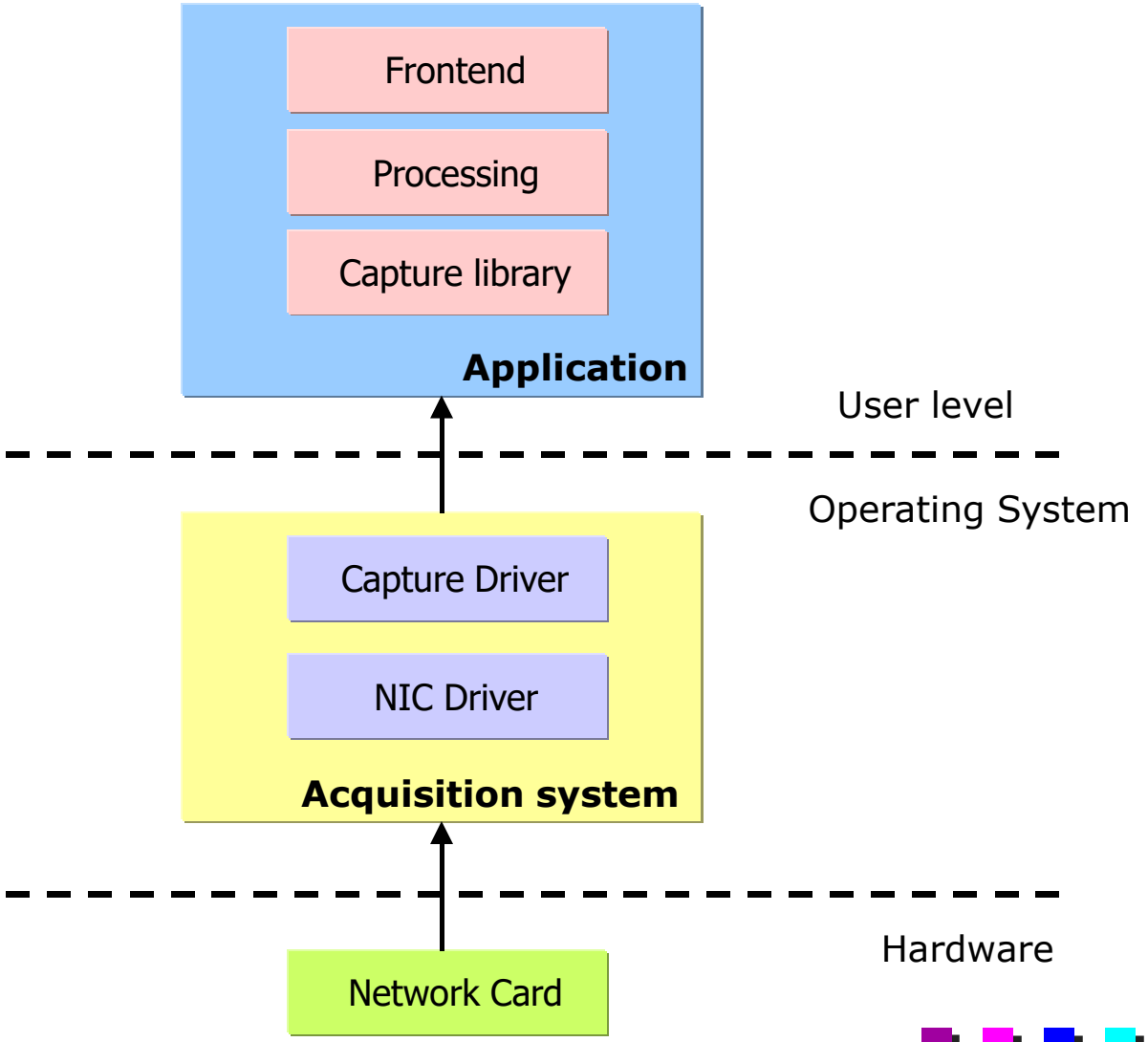
- The bytecodes (opcodes) are valid
  - Controlled “ahead of time” form the existence of a “default” branch in the switch
  - Does not cover possible translation errors of the JIT
- The destination of jump/branch are valid
  - Controlled “ahead of time” with appropriate checks in the translator
  - Controlled allowing only jumps with an explicit offset
- The number of instructions is finite
  - Controlled with appropriate checks before starting the translator
- Read and write start from valid memory zones
  - Controlled with appropriate checks in the native code
- Termination of the program guaranteed
  - A parameter can be the absence of loops
  - Some types of instructions (e.g., loops) may not be allowed
    - E.g., indirect jumps such as jmp[ECX]
- Finite and predictable memory consumption
  - It is guaranteed if there is guarantee of termination of the program

Part 3:
Toward high-speed software packet filtering

The way towards better performance

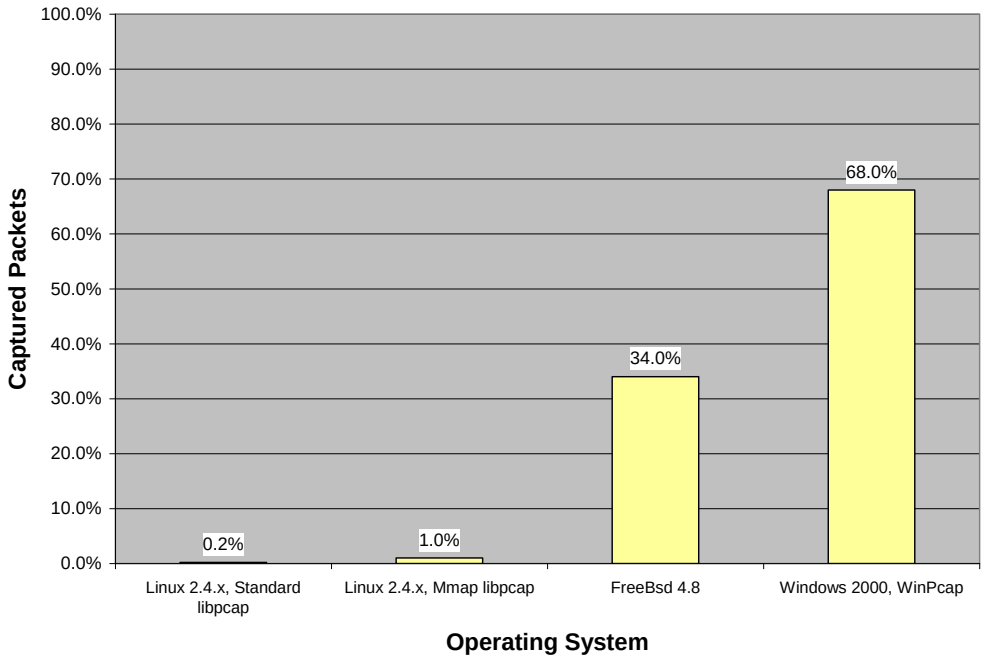
- Motivations
  - Software is very flexible
  - Necessity of speed analysis>= 1Gbps
- Possibility of improvement
  - Increase the performance of the capture
    - Increases the capacity of delivering data to the software
  - Create more intelligent analysis components
    - Only the most interesting data are delivered to the software
  - Architectural optimization
    - Try to exploit the characteristics of the application to increase performance

Reference model for Packet Capture



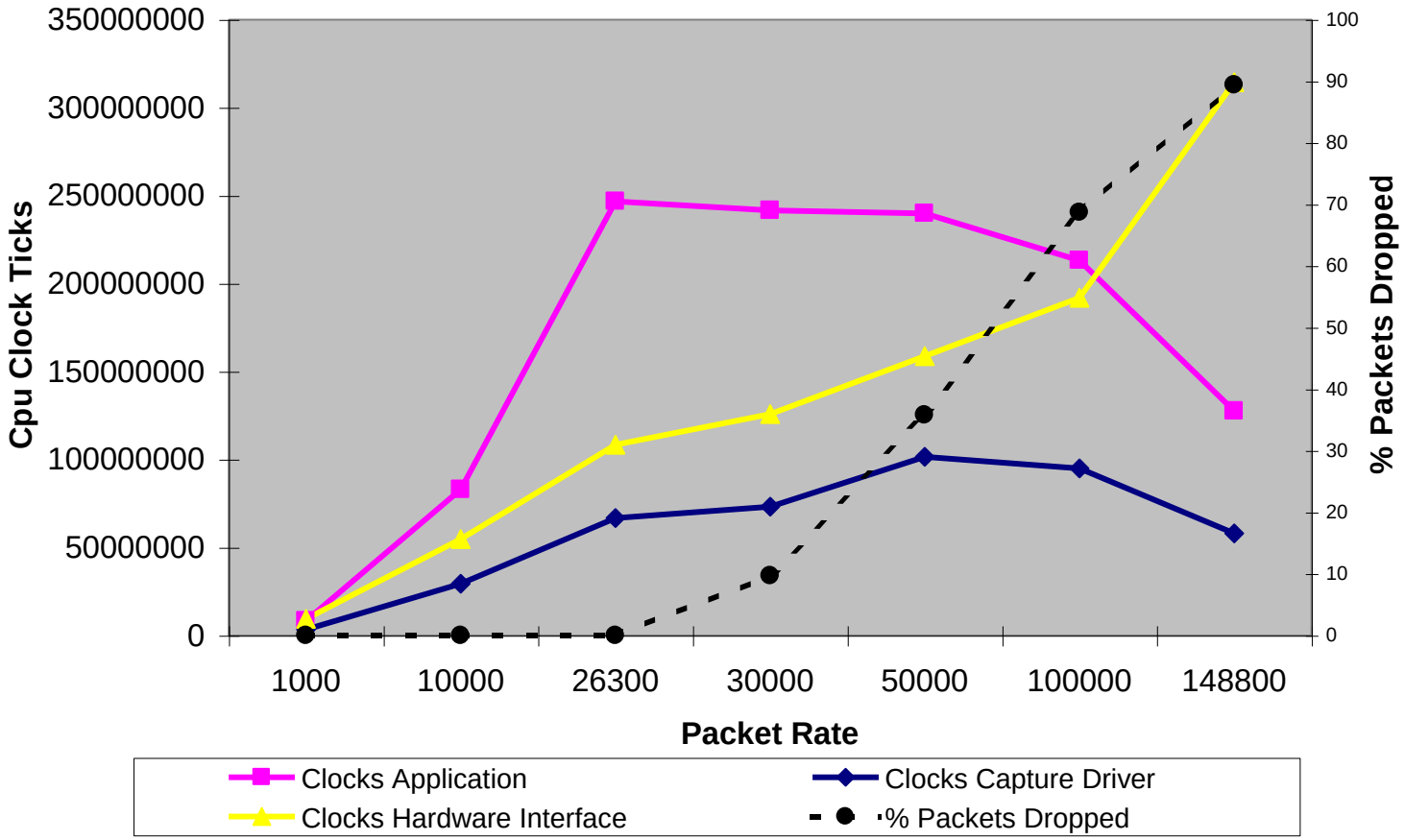
Performance of OS and capture drivers

- Huge differences for capture performance depending on
  - Operating system
  - Capture driver
- Overall architecture looks the same, but performance are very different



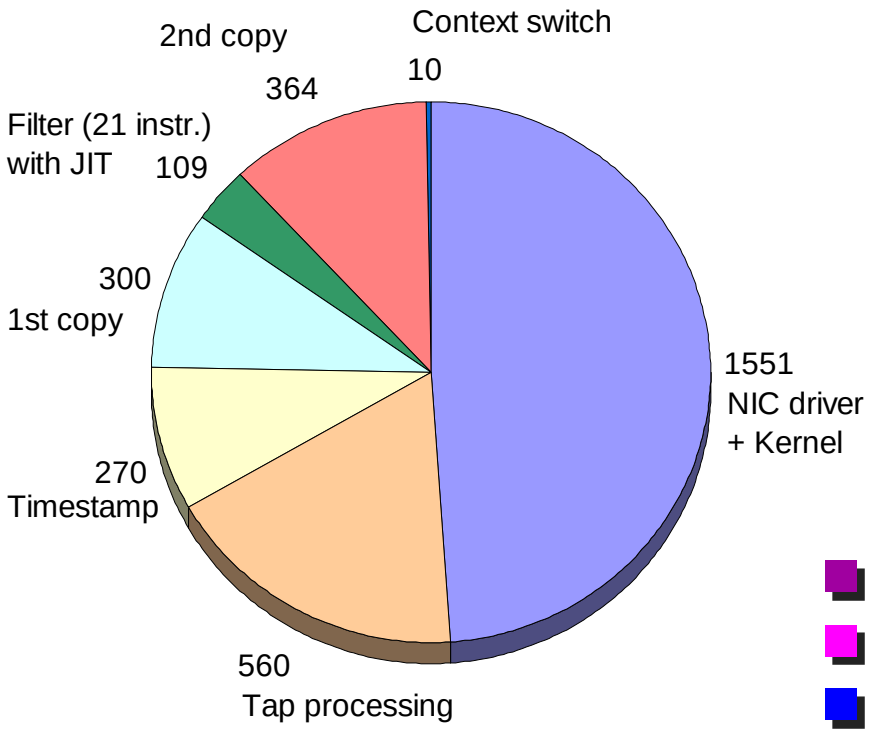
Source: Luca Deri, ntop.org

Kernel vs. user processing: the Livelock problem



Some profiling data: WinPcap

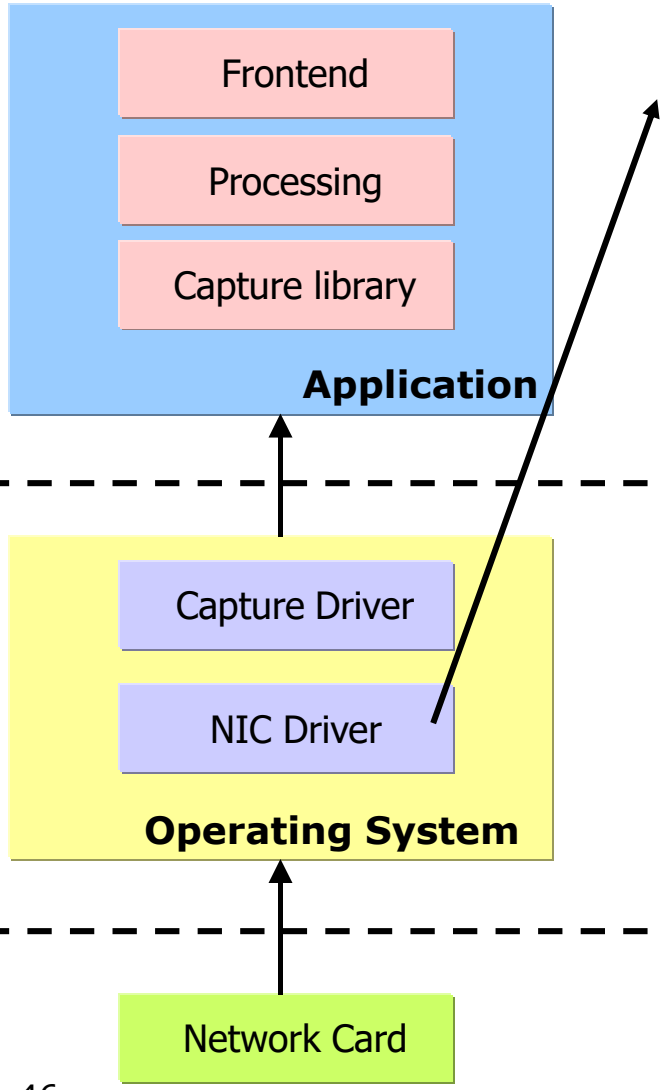
- Some data
  - The filtering costs are proportionally low
  - The copy doesn't seem to be the prevailing cost
  - The cost of read (packet batching) is insignificant
- The greatest costs are:
  - Costs of the OS and the NIC
  - Timestamp (hw?)
  - The copies can become a problem with big packets (shared buffers)



Costs measured in Winpcap 3.0 (per packet; 64B)

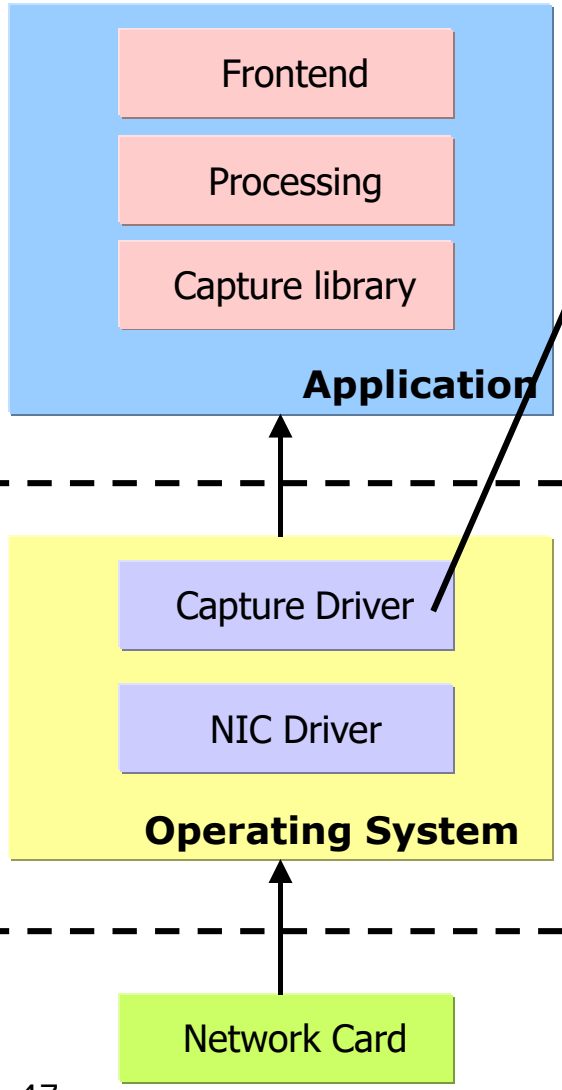
3164 clock ticks

Improving the costs related to the OS and NIC



- Problems
  - Interrupt (for each packet)
    - Hardware Interrupt Service Routine
    - Copy packets from plain RAM to Kernel structures
      - Cache miss
      - Un-optimized structures (e.g. small mbuf)
      - Allocate kernel structures
  - Access to the hardware (e.g. setting values in the NIC registers)
- Solutions:
  - Interrupt Mitigation
    - Hardware-based
  - Interrupt Batching
    - Software-based
  - Device Polling
    - E.g. FreeBSD (Rizzo)
  - Hybrid models Interrupt-Polling (e.g. Linux NAPI)
  - Pre-program the hardware (avoiding access to hw registers)
  - Pre-allocated memory

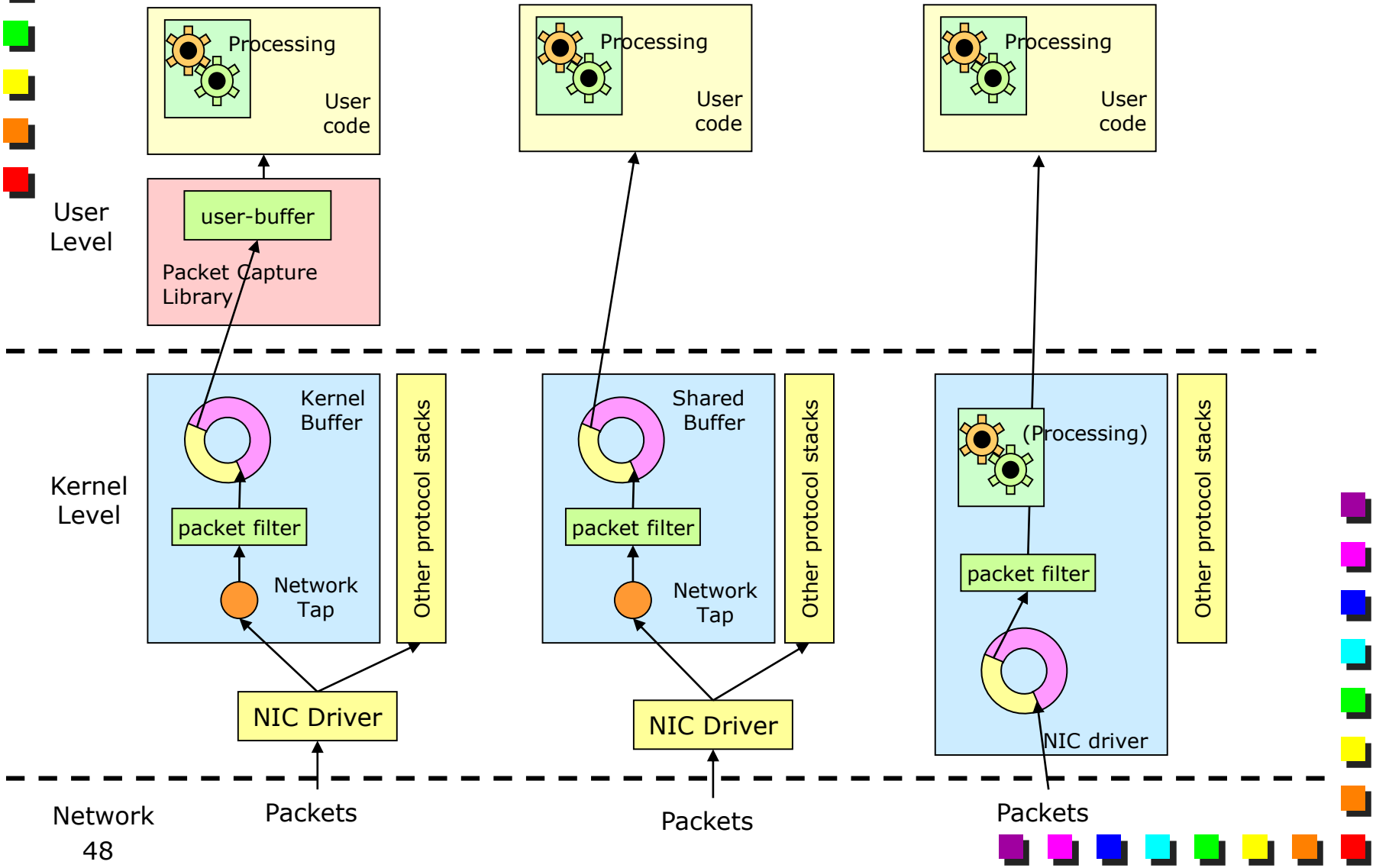
Improving the costs related to the capture driver



- Goals
  - Timestamp the packets
  - Deliver packets to the application
- Bottlenecks:
  - Context Switch (~10^4 clock cycles in Windows)
  - Packet copies
  - Cache miss
- Solutions:
  - Packet filtering, shapshot capture (not always possible)
  - Bulk copies
  - Large buffers (may be useful if shared with the application)
  - Shared memory between kernel and user space (Deri, PF_RING, 2004)*

*Luca Deri, "*Improving Passive Packet Capture: Beyond Device Polling*", Proceedings of SANE 2004, October 2004.

A possible further improvement

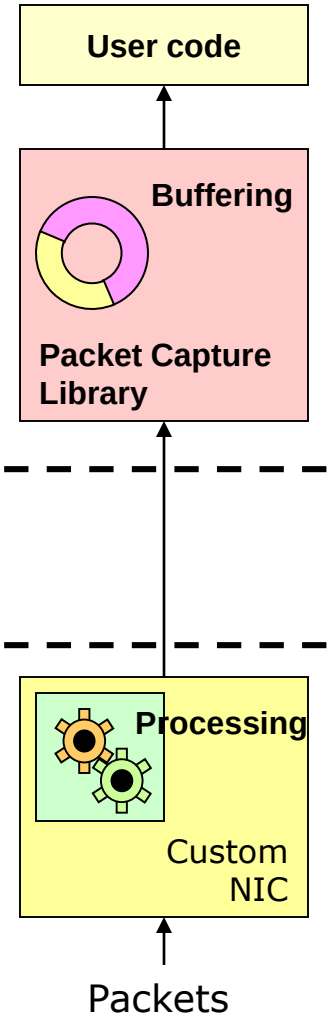


Packet filtering stack separated from the network stack

- Possible implementations
  - Traditional NIC with dedicated driver (Deri, NCAP*)
  - Intelligent NIC
- Characteristics
  - The OS is not made to support large network traffic (e.g., mbuf in BSD or skbuf in Linux)
    - It has been engineered to execute user applications, with limited memory consumption
  - Software stack (starting from the NIC driver) dedicated to the capture
    - Data is not delivered to the other TCP/IP components of the network stack
  - Modification intrusive in the operating system
  - Very good performance
    - Limited by the PCI bandwidth
  - Problems with the precision of the timestamp (if implemented in software)

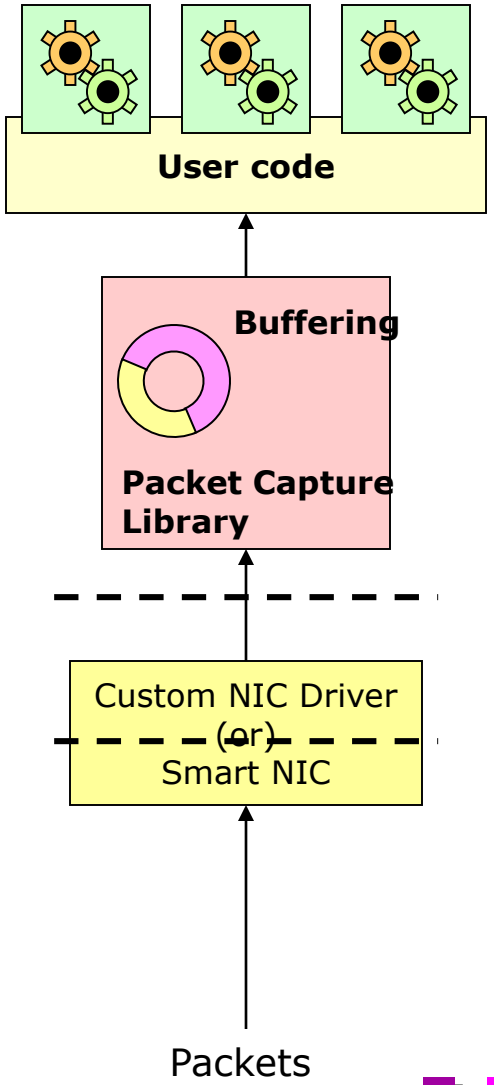
*Luca Deri, "nCap: Wire-speed Packet Capture and Transmission", Proceedings of E2EMON, May 2005.

Further improvements



Create smarter NICs

- Hardware processing
- Avoid PCI bus bottleneck (not applicable for "*capture all*" applications)
- Timestamp precision
- Need advanced mechanism for customizable processing



Increase parallelism in user space

- PCI bottleneck
- Easy to customize processing (general purpose CPUs)

Increase parallelism in kernel space

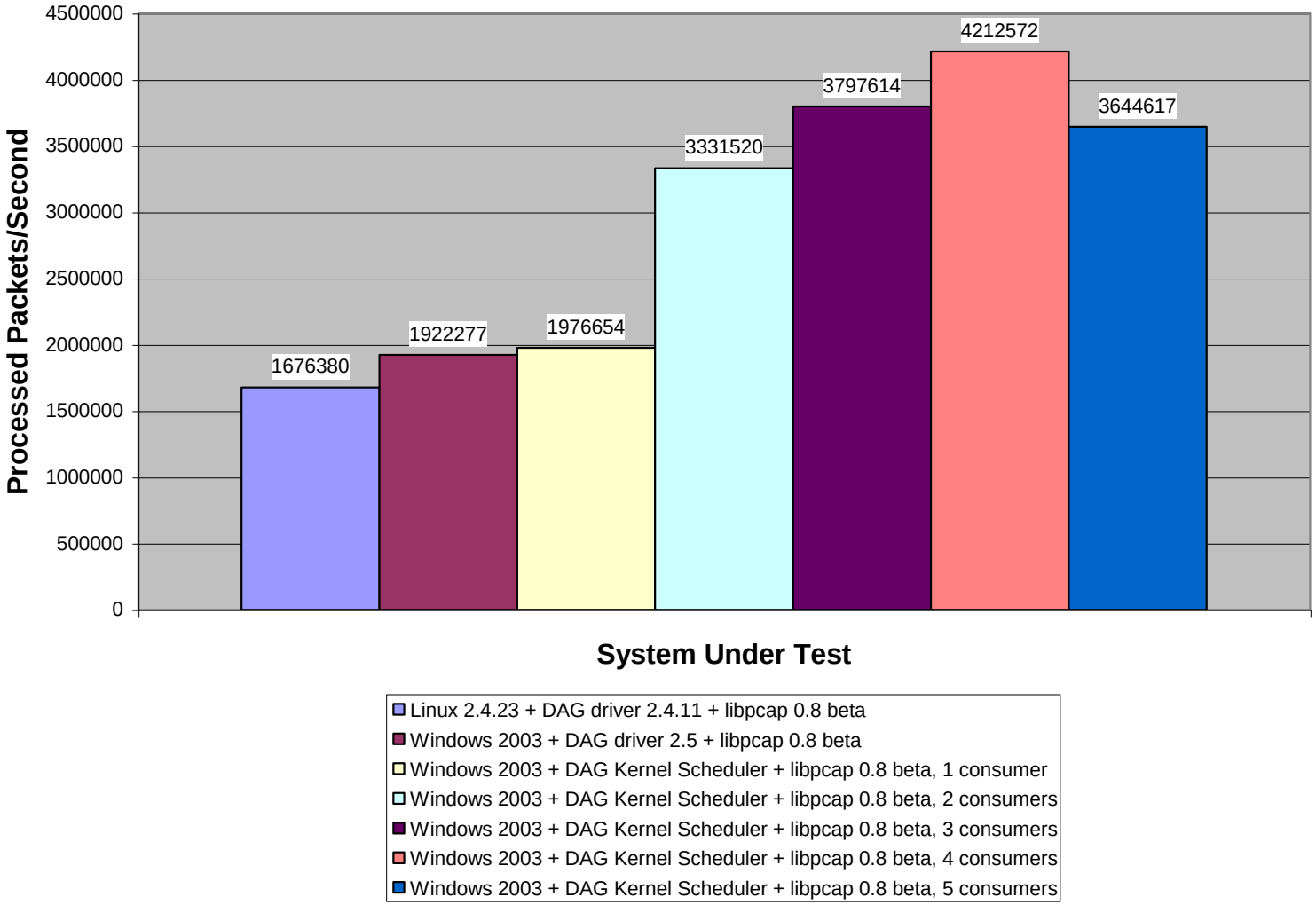
- Timestamp precision

Process parallelization in user-space

- Technique also proposed by FFPF
  - The integrations with intelligent buffer mechanisms
  - Easy to implement (it is only software)
  - Efficient on current CPU architectures
  - There may be synchronization problems
    - Applications that require the result of a previous step
  - Bus limitations:
    - PCI 1.0 (32bit, 33MHz) ➔ 1 Gbps
    - PCI 2.2 (64bit, 66MHz) ➔ 4.2 Gbps
    - PCI-X (64bit, 133MHz) ➔ 8.5 Gbps
    - PCI-X 2.0 (64bit, 266MHz) ➔ 17 Gbps
    - PCI-Express (16x) ➔ 32 Gbps
  - Growing interest in this technique

Loris Degioanni, Gianluca Varenni, "Introducing scalability in network measurement: toward 10 Gbps with commodity hardware". Internet Measurement Conference 2004, pg. 233-238

Example of parallelizzazione in user-space



The way towards better performance: summary

- Optimizes as much as it can
- Moves the processing in the kernel
  - Limits the displacement of data
- Decouples the packet filtering stack from that of the network
- Moves the processing to intelligent files
  - Limits the displacement of data
- Improves the parallelism
  - And in general, tries to exploit the characteristics of the application to go faster

Conclusions

- Academic interest mostly directed towards the packet filtering component
- In reality, the analysis of the whole system is much more important
- Current status
  - Netmap (from Luigi Rizzo) may be the fastest open-source component for direct NIC access
  - Other components (e.g., DNA, from Luca Deri) are not completely free

Bibliography

- Steven McCanne, Van Jacobson, “The BSD packet filter: a new architecture for user-level packet capture,” in *Proceedings of the USENIX Winter 1993 Conference* (USENIX'93). USENIX Association, Berkeley, CA, USA, 1993.
- Fulvio Risso, Loris Degioanni, “An Architecture for High Performance Network Analysis,” in Proceedings of the 6th IEEE Symposium on Computers and Communications (ISCC 2001), Hammamet, Tunisia, July 2001.