Packet Filters

Proposed solutions and current trends

Vasileios P. Kemerlis

Network Security Lab
Computer Science Department
Columbia University
New York, NY

04/14/2010

Outline

1. Introduction
   - Overview
   - Why bother?
2. Packet Filters
   - CMU/Stanford Packet Filter (CSPF)
   - The BSD Packet Filter (BPF)
   - The Mach Packet Filter (MPF)
   - Dynamic Packet Filters (DPF)
   - The BSD Packet Filter+ (BPF+)
   - xPacket Filter (xPF)

Outline

1. Introduction
   - Overview
   - Why bother?
2. Packet Filters
   - CMU/Stanford Packet Filter (CSPF)
   - The BSD Packet Filter (BPF)
   - The Mach Packet Filter (MPF)
   - Dynamic Packet Filters (DPF)
   - The BSD Packet Filter+ (BPF+)
   - xPacket Filter (xPF)

Packet Filter

What is it anyway?

- Kernel-level mechanism (typically, but not always)
- Allows direct access to the packets (frames?) received from the network interface controller (NIC) – “tap” NICs
- Integral part of every modern operating system (OS)

Outline

1. Introduction
   - Overview
   - Why bother?
2. Packet Filters
   - CMU/Stanford Packet Filter (CSPF)
   - The BSD Packet Filter (BPF)
   - The Mach Packet Filter (MPF)
   - Dynamic Packet Filters (DPF)
   - The BSD Packet Filter+ (BPF+)
   - xPacket Filter (xPF)

Packet Filter
Why bother?

- Almost every user-space network protocol implementation utilizes such facilities
- Utilized by modern network monitoring tools (*tcpdump, wireshark*)
- Provides a critical *handle* to intrusion detection systems (*Snort, Bro*)

Outline

1. Introduction
   - Overview
   - Why bother?
2. Packet Filters
   - CMU/Stanford Packet Filter (CSPF)
   - The BSD Packet Filter (BPF)
   - The Mach Packet Filter (MPF)
   - Dynamic Packet Filters (DPF)
   - The BSD Packet Filter+ (BPF+)
   - xPacket Filter (xPF)

CSPF

Status in the early 80's

- Historically, the first user-level "packet filter" appeared on Xerox Alto [1]
- Special-purpose process (`demux`) for deciding where each packet should go
- Multiple context switches and three system calls per received packet

[1] Butler W. Lampson and Robert F. Sproull. An open operating system for a single-user machine. In Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP), pages 98–105, Pacific Grove, CA, USA, December 1979.

CSPF

User-level packet demultiplexing

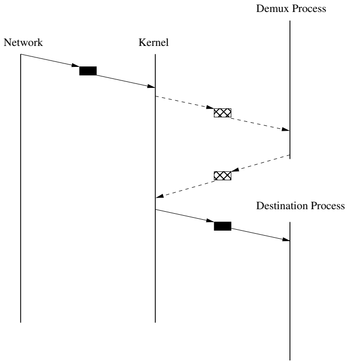

Figure: User-level packet demultiplexing

CSPF

Motivation

- User-space packet demultiplexing is expensive
- TCP/IP has yet to become the de-facto standard; experimental network protocols are flourishing
- User-level protocol implementations are necessary to allow experimentation without kernel hacking (tedious, error-prone, overwhelming) – no fancy kernel-level debugging facilities!

CSPF

Kernel-level packet demultiplexing

- Kernel facility that offers packet demultiplexing services to user-level network implementations
- Avoids the “dashed” part illustrated in Figure 1
- Flexible, protocol independent, mechanism for “selecting” packets

CSPF

Design

- Uses a special-purpose language for a stack pseudo-machine (VM in nowadays)
- Applications use the language to describe arbitrary predicates for the packets they are interested in (filters are “programs” of that language)
- Instructions are made from 16-bit words that encode typical arithmetic/logical and stack-based operations
- Each filter is “executed” with a packet as input
- If the top of the stack is non-zero at the end, a copy of the packet is delivered to the process installed the filter

```
struct enfilter f = {
10, 12, /* priority and length */
PUSHWORD+1, PUSHLIT | EQ, 2, /* packet type == PUP */
PUSHWORD+3, PUSH00FF | AND, /* mask low byte */
PUSHZERO | GT, /* PupType > 0 */
PUSHWORD+3, PUSH00FF | AND, /* mask low byte */
PUSHLIT | LE, 100, /* PupType <= 100 */
AND, /* 0 < PupType <= 100 */
AND /* && packet type == PUP */
};
```

Figure: Example of a filter program for the Pup protocol

CSPF

User-level packet demultiplexing

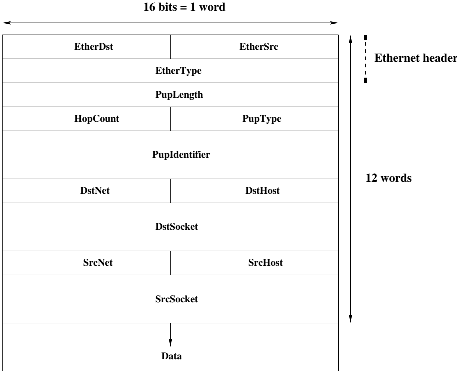

Figure: The Pup protocol header (inside an Ethernet frame)

CSPF

Implementation & usage

- CSPF was implemented in 4.3BSD UNIX (DEC VAX 11/790, PDP-11)
- Usage procedure:
  1. a special-purpose *character device* is called from the user code via the usual system calls: `open`(2), `close`(2), `read`(2), `write`(2)
  2. assemble some filters, similar to the one showed in Figure 2, and use the `ioctl`(2) system call to bind them to the character device opened in the previous step
- Evaluation of CSPF [2] indicated that kernel-level packet demultiplexing can gratefully assist user-level protocol implementations (minimize processing latency)

[2] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. In Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP), pages 39–51, Austin, TX, USA, November 1987.

Outline

1. Introduction
   - Overview
   - Why bother?
2. Packet Filters
   - CMU/Stanford Packet Filter (CSPF)
   - The BSD Packet Filter (BPF)
   - The Mach Packet Filter (MPF)
   - Dynamic Packet Filters (DPF)
   - The BSD Packet Filter+ (BPF+)
   - xPacket Filter (xPF)

BPF

State of affairs in the early-90's

- 4.3BSD UNIX brought a new TCP/IP implementation
- Quickly became the authoritative reference, inherited by many other free/commercial Unixes
- User-level protocol implementation declined
- Packet filtering facilities were mostly utilized for monitoring purposes

BPF

Motivation

- CSPF was designed around the ISA of old DEC machines
- Worked well on a 64K PDP-11, but performed sub-optimally on RISC-based architectures
- Why?

BPF
Motivation

- CSPF was designed around the ISA of old DEC machines
- Worked well on a 64K PDP-11, but performed sub-optimally on RISC-based architectures
- Why?
- The stack-based VM requires multiple memory references for the execution of a single filter
- Memory references result in hundreds of wasted CPU cycles (divergence between CPU clock speed and memory speed)

BPF

Design & architecture

- BPF uses a new register-based VM and a redefined language
- Maintains the flexibility and generality of CSPF
- Performs better on modern, RISC, machines
- Two main components:
    1. the network tap
    2. packet filter

BPF

The network tap

- Part of BPF responsible for packet collection
- “Taps” NICs; for every NIC with filters installed, it calls BPF (Figure 4)
- If the packet is accepted, a copy of it (actually a part of it) is copied in a per-filter buffer
- Can batch multiple packets and deliver them with one system call (minimizes context switches)

BPF
Network tap overview

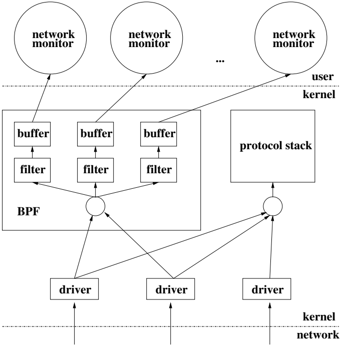

Figure: BPF architecture

BPF

The packet filter

- Most applications tend to reject more packets than they accept
- A filter should reject a packet after few instructions and avoid redundant computations
- CSPF filters are modeled as trees (Figure 5)

BPF

CSPF filter model

- Simulated operand stack
- Unnecessary or redundant computations
- Cannot handle variable length packet headers
- Requires multiple instructions to deal with 32-bit fields

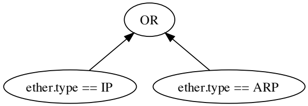

Figure: CSPF tree example

BPF

VM design constrains

- Protocol-independent design (handle future protocols)
- Generality (rich ISA for handling unforeseen cases)
- Simplified instruction decoding (performance)
- One-to-one matching (ideally) between VM registers and physical machine registers

```
ldh [12]
jeq #0x800 jt 2 jf 6
ld [26]
jeq #0xd0448b59 jt 12 jf 4
ld [30]
jeq #0xd0448b59 jt 12 jf 13
jeq #0x806 jt 8 jf 7
jeq #0x8035 jt 8 jf 13
ld [28]
jeq #0xd0448b59 jt 12 jf 10
ld [38]
jeq #0xd0448b59 jt 12 jf 13
ret #65535
ret #0
```

Figure: Example of a BPF program for “`host optimus`”

- *tcpdump* monitoring utility (v4.0.0) on Mac OS X 10.6
- `tcpdump -d -i en0 host optimus`

BPF

BPF filter model (CFG)

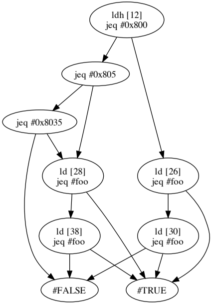

Figure: CFG representation of filter “`host foo`”

BPF

Implementation & usage

- BPF was implemented in 4.3BSD Tahoe/Reno UNIX, 4.4BSD UNIX, HP-UX BSD variants, SunOS 3.5...
- Currently is supported by every modern free BSD flavor (*e.g.,* FreeBSD, NetBSD, OpenBSD) as well as by Linux
- Using BPF from application processes shared a great similarity with CSPF
- Evaluation of BPF [3] showed that it offers 20x times faster filtering than CSPF and 150x times faster packet filtering than Sun's Network Interface Tap (NIT) – now known as Data Link Provider Interface (DLPI)

[3] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In Proceedings of the USENIX Winter Conference, pages 259-269, San Diego, CA, USA, January 1993.

Outline

1. Introduction
   - Overview
   - Why bother?
2. Packet Filters
   - CMU/Stanford Packet Filter (CSPF)
   - The BSD Packet Filter (BPF)
   - The Mach Packet Filter (MPF)
   - Dynamic Packet Filters (DPF)
   - The BSD Packet Filter+ (BPF+)
   - xPacket Filter (xPF)

MPF

Motivation

- In early 90's research in microkernel OSes made efficient packet demultiplexing a hot topic, again
- In a microkernel OS, traditional kernel-space facilities (*e.g.*, protocol processing) are pushed to user-level processes
- CSPF seems an adequate solution...
- A single point of primary dispatch for all network traffic results in an increased communication overhead (Figure 8)

MPF
Protocol processing model

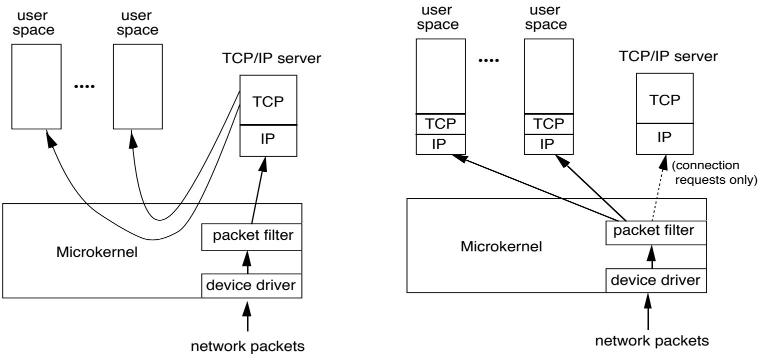

Figure: Protocol processing approaches in microkernel OSes

MPF

Details

- Kernel-level facility that efficiently dispatches incoming packets to multiple endpoints (*e.g.*, address spaces)
- Support for multiple active filters (scalable)
- Flexible and generic (5 additional instructions in BPF)
- Why not use BPF then?

MPF
Details

- Kernel-level facility that efficiently dispatches incoming packets to multiple endpoints (*e.g.,* address spaces)
- Support for multiple active filters (scalable)
- Flexible and generic (5 additional instructions in BPF)
- Why not use BPF then?
  1. *scalability issues*. The dispatching overhead increases with the number of different endpoints
  2. *cannot handle multi-packet messages*. BPF cannot identify packet fragments (it cannot “remember” what it has seen)

MPF

Efficient dispatching

- MPF exploits structural and logical similarity among different, but not identical filters
- Identifies filters that have common "prefixes"
- *Collapses* common filters into one
- Uses associative matching for dispatching to the final communication endpoint (Figure 9)

MPF
Associative model

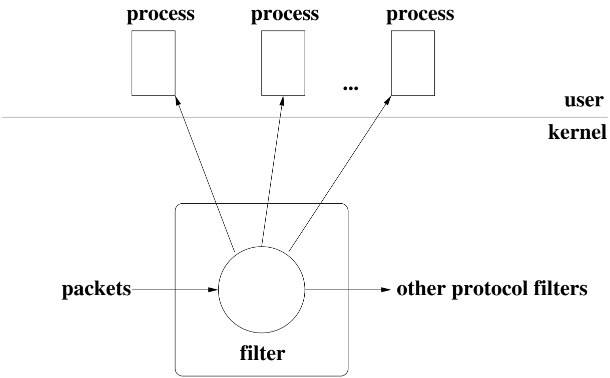

Figure: MPF associative model

```
/* Part (A) */
begin ; MPF identifier
ldh P[#OFF_ETHER_TYPE] ; A = ethernet type field
jeq #ETHER_TYPE_IP, L1, fail; if no IP fail
L1: ld P[#OFF_DST_IP] ; A = dst IP address
jeq #dst_addr, L2, fail ; if not from dst_addr fail
L2: ldb P[#OFF_PROTO] ; A = protocol
jeq #IPPROTO_TCP, L3, fail ; if not TCP, fail
L3: ldh P[#OFF_FRAG] ; A = fragmentation flags
jset #!DF_BIT, fail, L4 ; if DF_bit = 1, fail
L4:
/* Part (B) */
ld P[#OFF_SRC_IP] ; A = src IP address
st M[0] ; M[0] = A
ldxb 4 * (P[OFF_IHL] & 0xf) ; X = TCP header offset
ldh P[x + #OFF_SRC_PORT] ; A = src TCP port
st M[1] ; M[1] = A
ldh P[x + #OFF_DST_PORT] ; A = dst TCP port
st M[2] ; M[2] = A
/* Part (C) */
ret_match_imm #3, #ALL ; compare keys with M[0..2]
key #src_addr ; if matched, accept the
key #src_port ; whole packet. If not,
key #dst_port ; reject it
fail:
ret #0
```

Figure: Example of an MPF program for a TCP/IP session

MPF

Dispatching multi-packet messages

- Typical case when IP fragmentation is used
- A large TCP/UDP packet is divided into multiple IP fragments
- Only one has the TCP/UDP header
- MPF response:
    1. *filter state*. Per-filter “state” buffers
    2. additional *instructions* for handling fragments. *Postpone* the dispatch decision for a while

MPF
Critique

- 8x faster than CSPF and 4x faster than BPF [4]
- But...

[4] Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In Proceedings of the Winter USENIX Technical Conference (USENIX WTC), pages 153–165, San Francisco, CA, USA, January 1994.

MPF

Critique

- 8x faster than CSPF and 4x faster than BPF [4]
- But...
- MPF was designed for Mach 3.0 (microkernel OS). No port exists for other OSes, yet
- It demands from the filters to have specific structure in order to optimize them (collapse into one). Reduced flexibility in expressions
- Associative search instructions make extensive use of BPF's scratch memory. Depending of how memory accesses are emulated, MPF might lead in memory *spills* – recall BPF's original purpose

[4] Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In Proceedings of the Winter USENIX Technical Conference (USENIX WTC), pages 153–165, San Francisco, CA, USA, January 1994.

Outline

1. Introduction
   - Overview
   - Why bother?
2. Packet Filters
   - CMU/Stanford Packet Filter (CSPF)
   - The BSD Packet Filter (BPF)
   - The Mach Packet Filter (MPF)
   - Dynamic Packet Filters (DPF)
   - The BSD Packet Filter+ (BPF+)
   - xPacket Filter (xPF)

DPF
Motivation

- Similar to MPF:
  - Minimize end-to-end latency of user-level protocol stacks
  - Applications can explore new networking mechanisms without kernel modifications
  - Usually trade flexibility for performance
- Fast and flexible message demultiplexing is important
- Proposed solutions sacrifice one for the other
  - BPF: flexible and general, but not scalable
  - MPF: less flexible, more scalable

DPF
Design & architecture

- Kernel-level facility for rapid packet demultiplexing
- New, carefully-designed, declarative language
- Aggressive dynamic code generation
- Performance is equivalent, or can exceed, hand-coded demultiplexers

DPF

Packet filter language

- Declarative language; general, flexible, protocol agnostic
- Filters are described as sequences of boolean comparisons (*atoms*) linked by conjunctions
- Set of active filters are stored into a *prefix tree* data structure (Figure 11)

DPF

Trie structure (prefix tree)

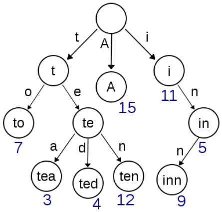

Figure: A trie for keys "A", "to", ..., "inn" (courtesy of Wikipedia)

```
(
# check ethernet header

(12:16 == 0x8) && # IP datagram?
# skip ether header (14 bytes)
(SHIFT(6 + 6 + 2)) &&

# check IP header

(9:8 == 6) && # check protocol : TCP is 6
# check IP src addr (192.12.69.1)
(12:32 == 0xc00c4501) &&


# skip IP header (assume fixed sized; 20 bytes)
(SHIFT(20)) &&

# check TCP header

# check source port (2 bytes)
(0:16 == 1234) &&

# check destination port (2 bytes)
(2:16 == 4321) &&
)
```

Figure: Example of a DPF program for a TCP/IP session

DPF
Filter handling

- New filters are stored in the trie along with path with the longest prefix match – similar to MPF, this leads in prefix “collapse”
- Duplicate checks are eliminated
- Filters that cannot merge with the trie, or they form a new one, are connected with it using an *or* branch
- “Forest” of prefix trees

DPF

Dynamic code generation

- Eliminates interpretation overhead by compiling into native code
- Aggressive optimization
  - Runtime information is encoded in the instruction scheme (*e.g.*, constants that are known only after a connection is established)
  - Fast disjunctions. Avoids hash-based lookups for disjunctive filters that have been merged, but the necessary checks are relatively few
  - Atom coalescing (Figure 13)
  - Alignment estimation
  - Bounds checking

```
# TCP header before coalescing
(0:16 == 1234) && # check source port
(2:16 == 4321) # check destination port

# TCP header after coalescing
(0:32 == 283182290) # 283182290 ==
# ((4321 << 16) | 1234)
```

Figure: Coalescing example

DPF
Critique

- 25–50x times faster than MPF [5]
- But...

[5] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pages 53–59, Standford, CA, USA, 1996.

DPF
Critique

- 25–50x times faster than MPF [5]
- But...
- DPF was designed for Aegis (exokernel OS). No port exists for other OSes, yet
- Relies on VCODE dynamic code generation system (portability?)
- No side-effects; what about variable-length headers? multi-packet messages?

[5] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pages 53–59, Standford, CA, USA, 1996.

Outline

BPF+

Motivation

- BPF has limitations (reason to have MPF, DPF, ...)
- Decision tree reduction is NP-complete
- But...

BPF+

Motivation

- BPF has limitations (reason to have MPF, DPF, ...)
- Decision tree reduction is NP-complete
- But...
- Filters have a *regular* structure that can be exploited from optimization frameworks
- MPF, DPF use local optimizations and they do not eliminate common subexpressions
  - Restrict the expressibility of the filters by imposing a specific structure (MPF)
  - Rely on the programmer to express the filter in an optimized and compact way (DPF)

BPF+
Motivation

- BPF has limitations (reason to have MPF, DPF, ...)
- Decision tree reduction is NP-complete
- But...
- Filters have a *regular* structure that can be exploited from optimization frameworks
- MPF, DPF use local optimizations and they do not eliminate common subexpressions
  - Restrict the expressibility of the filters by imposing a specific structure (MPF)
  - Rely on the programmer to express the filter in an optimized and compact way (DPF)
- Bottom line: we need *global* filter optimization

BPF+

Features

- Exploits data-flow algorithms for generalized optimization among filters (Figure 14)
- Eliminates *redundant* predicates
- Allows for matching header fields against one another
- Enables arithmetic operations on header words before matching
- Can generate native code using just-in-time (JIT) compilation
- Relies upon a refined VM (more GPR, branch instructions can use register values)

BPF+

Generalized optimization

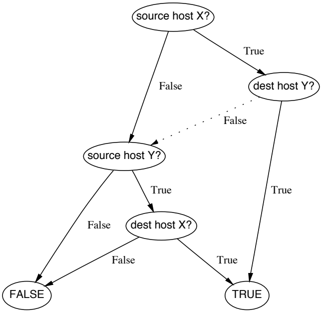

Figure: Typical (DPF) CFG for “`(src host X and dst host Y) or (src host Y and dst host X)`”

BPF+

Architecture overview

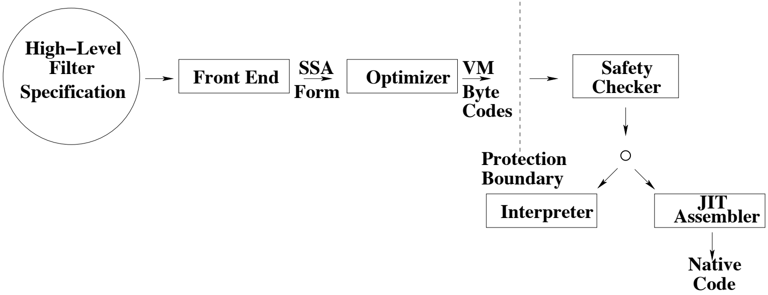

Figure: BPF+ architecture

BPF+

Design & architecture

- Filter specifications written in a high-level predicate language (*libpcap*)
  - `((src network MIT and dst network UCB) or (src netork UCB and dst network MIT)) and (TCP port HTTP)`
- Typical compiler structure (*front end*, *back end*)
- Straightforward code generator (on the fly translation to the intermediate SSA form)
- The CFG is guaranteed to by acyclic (forward branches only)
- Optimizer eliminated redundancies and performs register allocation

BPF+

Critique

- Misses juxtaposition (BPF, MPF, DPF, ...) [6]
- No per-filter state (MPF)
- No side-effects on user-level state variables or packets
- No backward branches (cannot implement loops, counting)
- Return value is still true/false. What about #predicates matched?

[6] Andrew Begel, Steven McCanne, and Susan L. Graham. BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture. ACM SIGCOMM Computer Communication Review, 29(4):123–134, 1999.

Outline

1. Introduction
   - Overview
   - Why bother?
2. Packet Filters
   - CMU/Stanford Packet Filter (CSPF)
   - The BSD Packet Filter (BPF)
   - The Mach Packet Filter (MPF)
   - Dynamic Packet Filters (DPF)
   - The BSD Packet Filter+ (BPF+)
   - xPacket Filter (xPF)

xPF

Motivation & enhancements

- Need for more elaborate computational capabilities
- Engine for *executing* monitoring applications in kernel-space rather than a demultiplexing mechanism
- Persistent memory (per-filter)
- Support for backward branches

xPF

Implementation & usage

- xPF was implemented in OpenBSD [7]
- No comparative evaluation
- No safety guarantees because of the backward branches

[7] Sotiris Ioannidis, Kostas G. Anagnostakis, John Ioannidis, and Angelos D. Keromytis. xPF: Packet Filtering for Low-cost Network Monitoring. In Proceedings of the IEEE Workshop on High-Performance Switching and Routing (HPSR), pages 121–126, Kobe, Hyogo, Japan, 2002.