FFPF: Fairly Fast Packet Filters

Herbert Bos†, Willem de Bruijn†, Mihai Cristea, Trung Nguyen, Georgios Portokalidis
†Vrije Universiteit Amsterdam, The Netherlands
Universiteit Leiden, The Netherlands

USENIX OSDI 2004

Presented by Hsu Hao Chen
Outline

- Introduction
- FFPF overview
- Implementation
- IXP1200 platform
- Evaluation
- Conclusions
FFPF is a network monitoring framework designed for three things:

- **Speed**
  - Support 1Gbps above

- **Scalability**
  - Ability to handle multiple applications

- **Flexibility**
  - Support many filter expression languages (BPF、FPL-1、FPL2、C、OKE-Cyclone)
Introduction (2/2)

- **FFPF application:**
  - An intrusion detection system (IDS) checks the payload of every packet.
  - An application base on the “Coralreef” suite keeps statistics for the ten most active flows.
  - A tool is interested in monitoring flows for which the port number are not known a prior. (BT、H.323 etc..)
  - Multiple monitoring applications (snort、tcpdump etc..)
FFPF overview

Figure 1: The FFPF architecture
FFPF overview

- Flow grabber: filter information (packets, statistics)
- PBuf (packet buffer): A flow grabber receives the packets to stores.
- IBuf (index buffer): Applications use the pointers in IBuf to find packets in PBuf.
- MBuf (memory buffer): The buffer is used to produce results for applications.
FFPF overview

- **Flow groups** are used to minimize packet copying. Application in the same group share a common PBuf.
FFPF overview

- Receiving packets in a flow

flow_create()
flow_populate()
flow_instantiate()
flow_activate()
flow_pause()
flow_close()
FFPF overview

- Filter expressions
  - Next section explain
- Construction of filter graphs by users
  - **ffpf-flow**: a simple command-line tool

```
./ffpf-flow
"(device,eth0) | (device,eth1) -> (sampler,2,4) -> \n   (FPL-2,"...") | (BPF,"...") -> (bytecount,,8)"
"(device, eth0) -> (sampler,2,4) -> (BPF,"...") \n   -> (packetcount,,8)"
```

(sampler,2,4) captures one in two packets and requires our bytes of MBuf counter
**FFPF overview**

- **Ovals**: configuration
- **Circles**: filters component
- **Squares**: filters classes

*Figure 3: Auto-generated diagram of filter graph*
FFPF overview

- Processing
  - Most processing take place at the lowest possible level
    - Example: processing in the kernel or network processor
  - FFPF spans all there levels of the processing (trade-off efficiency vs. stability)
    1. Userspace
    2. Kernel
    3. Network interface (IXP1200 include network processor)
FFPF overview

Figure 4: FFPF software structure (with some sample flows)
Implementation

- **The buffers**
  - Both PBuf and I Buf are circular buffers of \( N \) fixed size slots, with \( N \) a configurable constant (\( N=64\) bits).
  - Applications are able to see packets received by all other in the same flow group.
Implementation

- Buffer management
  - Circular buffers in FFPF have two indices to indices the current read and write positions.
  - Buffer management system include SRP and FRP
  - SRP (slow reader preference):
    - In SRP, as longer as the buffer is full, all new packets are dropped.
FRP (fast reader preference):
- FRP is a departure from the traditional way of dealing with buffer overflow.
- R (read pointer) is of no concern to FFPF
  - Applications check after they processed a set of packets (check overwritten or consider them lost after all)

Filter-specific memory array (MBuf)
- It is used by both the filters in the kernel and the userspace application.
- Entire memory array as a hash table
  - Filter first checks a packet of TCP/IP
    - Hash of the <ipsrc,ipdest,srcport,dstport> tuple
Implementation

Application A

Shared memory

Application B

Flow grabber A  Flow grabber B
Implementation

- The flows (packets received way)
  - Polling
    - Loop (filter_getnext_pkt())
  - Blocking
    - wait_for_n_pkts(n): returns after n packets are received
  - Nonblocking
    - Install a filter_callback(): results in a callback (register time)
  - Sleep
    - Use filter_getnext_pkt() and sleep(10)
Implementation

- FPL (filter packet language)

<table>
<thead>
<tr>
<th>operator-type</th>
<th>operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>+, -, /, *, %, +, ++</td>
</tr>
<tr>
<td>Assignment</td>
<td>=, *=, /=, %=, +=, -=</td>
</tr>
<tr>
<td>Logical/Relational</td>
<td>==, !==, &gt;, &lt;, &gt;=, &lt;=,</td>
</tr>
<tr>
<td>Bitwise</td>
<td>&amp;,</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>statement-type</th>
<th>operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>if/then/else</td>
<td>IF (expr) THEN stmt1 ELSE stmt2 FI</td>
</tr>
<tr>
<td>for()</td>
<td>FOR (initialise; test; update) stmts; BREAK; stmts; ROF</td>
</tr>
<tr>
<td>external function</td>
<td>EXTERN(filter, input, output)</td>
</tr>
<tr>
<td>hash()</td>
<td>INT HASH(start byte, len, table size)</td>
</tr>
<tr>
<td>return a value</td>
<td>RETURN (val)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data type</th>
<th>syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register n(\d)</td>
<td>R([n])</td>
</tr>
<tr>
<td>Memory location n</td>
<td>MEM([n])</td>
</tr>
<tr>
<td>Packets access:</td>
<td>PKT.B([f(n)])</td>
</tr>
<tr>
<td>- byte f(n)</td>
<td>PKT.W([f(n)])</td>
</tr>
<tr>
<td>- word f(n)</td>
<td>PKT.B[n].U1([m])</td>
</tr>
<tr>
<td>- bit m in byte n</td>
<td>PKT.W[n].U8([m])</td>
</tr>
<tr>
<td>- byte m in word n</td>
<td>(many options, including macros)</td>
</tr>
</tbody>
</table>

Figure 5: FPL-2 language constructs (\(m\) and \(n\) arbitrary variables)
// count number of packets in every flow,
// by keeping counters in hash table
// (assume hash is unique for each flow)
IF (PKT.IP_PROTO == PROTO_TCP)
THEN
    // register = hash over TCP flow fields
    R[0] = Hash(14,12,256);
    // increment the pkt counter at this position
    MEM[ R[0] ]++;
FI

Figure 6: Example of FPL-2 code: count TCP flow activity
Implementation

- Monitoring application with dynamic ports
  - Many existing packet filters are not well suited for handling applications with dynamic ports.
    - example: peer-to-peer networks and multimedia streams that employ control protocols like RTSP, SIP, and H.323 to negotiate port numbers for data transfer protocols.
Implementation

1. // R[0] initially 0 stores no. of dynports found
2. IF (PKT.IP_PROTO==PROTO_TCP) THEN
3.   IF (PKT.TCP_DPORT==554) THEN
4.     MEM[R[0]]=EXTERN("GetDynTCPDPortFromRTSP",0,0);
5.     R[0]++;
6.   ELSE
7.     FOR (R[1]=0; R[1] < R[0]; R[1]++)
8.       IF (PKT.TCP_DPORT == M[ R[1] ] ) THEN
9.         RETURN TRUE;
10.  FI
11.  ROF
11.  FI
12.  FI
12.  RETURN FALSE;

Figure 7: Monitoring dynamic flows

- For example, given that RTSP packets are sent on port 544, GetDynTCPDPortFromRTSP scan all RTSP session packets for the occurrence of ‘transport’, ‘client_port’ and ‘server_port’. These ports stored in MBuf.
Implementation

Real Time Streaming Protocol (RTSP)

client

server

OPTIONS

DESCRIBE

SETUP

play

TEARDOWN

Contains dynamic port numbers
Implementation

- Compiler-time checks
  - Resources
  - Loop
  - Check (function called)
  - Simple authorisation (safety policy)

Figure 8: User compiles kernel module
Implementation

- Authorisation

Figure 9: User loads module in the kernel
Implementation

- FFPF packet sources
  - hook_handle_packet() takes a packet as argument.
  - Get packet sources:
    1. Netfilter
    2. Raw
    3. IXP 1200
Implementation

Iptables: INPUT
  - filter
  - NAT-DNAT
  - conntrack
  - mangle
  - NF_IP_LOCAL_IN
  - NF_IP_PRE_ROUTING
  - Iptables: INPUT

Iptables: FORWARD
  - filter
  - NF_IP_FORWARD
  - Iptables: FORWARD

Iptables: OUTPUT
  - filter
  - mangle
  - conntrack
  - NAT-MASQUERADE
  - NAT-SNAT
  - NF_IP_LOCAL_OUT
  - NF_IP_POST_ROUTING
  - Iptables: OUTPUT

Netfilter 建構在 Linux Kernel 2.4
IXP1200 platform
IXP1200 platform
I XP1200 platform

- FFPF on the I XP1200
  - The PBuf in the I XP implementation resides in SDRAM(256MB) on the network card.
  - Support zero copy

- The microengines
  - A single microengine per Gigabit port is responsible for receiving and buffering packets in PBuf.
  - All remaining microengines execute application filter expressions.
  - microengine places an index for the packet in the filter’s IBuf.
IXP1200 platform
IXP1200 platform
I XP1200 platform

Figure 2-5. The Compiler Thread Window

```c
while(1)
{
    if (packet_buf_addr == (dram_base *)) UNALLOCATED
        buf_pop(&pop_xfer,FREELIST_ID, ctx_swap); // if no buffer

    port_rxrdy_chk(rdready_inflight, rec_req);  // check for data
    critsect_enter(req_inflight);             // block other
    port_rx_request(rec_req);                 // get mpacket
    port_rx_receive(exception,rec_state, ETHER_100M); // get mpacket

    // mpacket_received#
    if (packet_buf_addr == (dram_base *)) UNALLOCATED
    {
        #if (FREELIST_ID == 0)
            #define BASE_ADDR SRAM BUFF_DESCRIPTOR_BASE
        #else
            #define HALF_BUFFER_COUNT (BUFFER_COUNT / 2)
            #define BASE_ADDR (SRAM BUFF_DESCRIPTOR_BASE + (HALF_BUFFER COUN
```
Figure 2-7. The Execution Coverage Window
To copy or not to copy

Developed three implementations:

- Zero copy
- Always copy packets to the host processor
- Only copy packets that have been marked as interesting by a host-side flow

We argue that this is not always optimal (zero copy)
Example: host application also needs to access the packet extensively, most reads have to cross the PCI bus.

FFPF conclude that in situations where both the host and the IXP copy the interesting packets once is always better than a zero-copy solution.
Evaluation

- **Testing environment**
  - Intel P3 1.2Ghz workstation with a 64/66 PCI bus running Linux 2.6.2
  - Application a: libpcap with Linux' LSF (Linux socket filter) backend
  - Application b: libpcap based on an FFPF implementation
Evaluation

Packet sniffing performance

Figure 11: System idle time for FFPF and LSF as a function of the bandwidth for different capture lengths
Figure 12: Idle time as function of the no. of concurrent applications for various capture lengths at 600Mbps
Conclusions

- FFPF provides a complete monitoring platform that caters to most applications.
- Speed, Scalability, Flexibility
- Further
  - Reader that are ‘too slow’ will be automatically placed in a separate flow group
  - Packet reception: will be able to block, edit, transmit (firewalling, NAT and route)