DPF: fast flexible packet filtering

Dawson Engler
Frans Kaashoek
MIT Lab for Computer Science

What & Why

- Packet filter:
  - kernel extension written in restricted language
  - input: incoming packet, output: {accept,reject}
- Why?
  - allows networking to be done at user-level
  - empirically useful: most OSes have one

The Problem

- Goal: speed + flexibility
- Packet filters:
  - tradeoff: extensible but slow
  - result: speed freaks don’t use them
- Hard-wired C/asm:
  - tradeoff: fast but inflexible
  - result: cannot interoperate with new protocols

Solution: Dynamic Packet Filters

- Extensibility:
  - DPF filters downloaded into kernel
- Speed:
  - Main trick: filters are dynamically compiled
  - Compilation enables further optimizations
- Result:
  - 10 - 50 x faster than best PF in literature
  - Speed rivals (or surpasses) hand-crafted asm

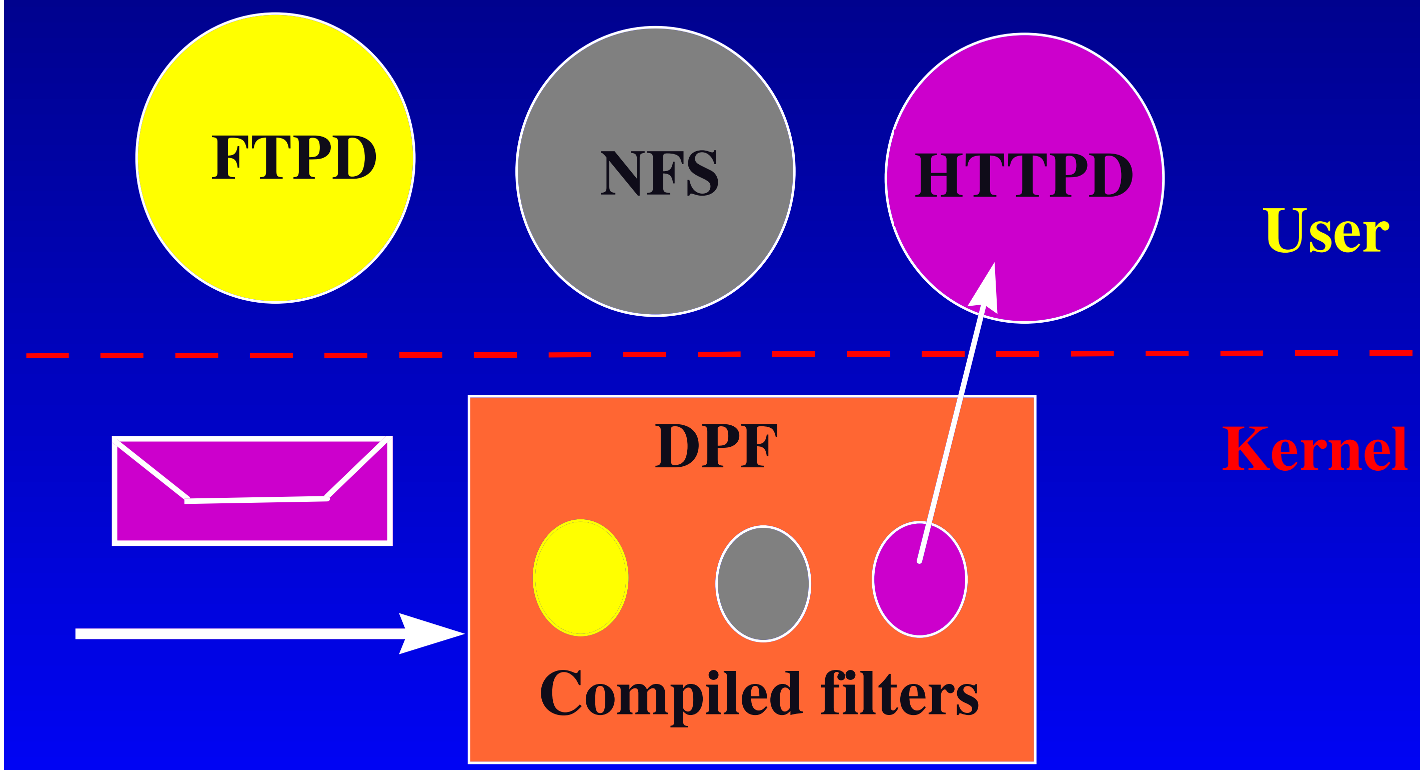
Talk Overview

- Problem & Solution
- Talk overview
- DPF overview
- Filter optimizations
- Results
- Experience + Summary

DPF: Operational View

- Filters:
  - written by clients
  - downloaded into DPF (in kernel)
  - checked for validity
  - merged into trie of filters, and then compiled
- Usage:
  - jam message through compiled trie
  - “first longest” match wins

DPF: Operational Overview



DPF Language Overview

- Syntax: C-style boolean expression
- Extensions:
  - load message values using “:”
  - move pointer into message using “shift”
- Example: demux using TCP ports
  - (0:16 ==1234) && (2:16 = 4321) && shift(20)
- Rest of talk: making filters fast

Dynamic code generation

- Executable code:
  - sequence of bits corresponding to machine inst
- Dynamic code generation (DCG):
  - generation of executable code at runtime
- Why?
  - eliminate interpretation overhead
  - runtime = more information = faster code

Using DCG for speed

- PF + DCG is a good match:
  - filter compiled once, used many times
- Not difficult:
  - complete code generator = 444 lines
- Portable:
  - used vcode public-domain DCG system
- General:
  - applying to existing PFs is straight-forward

Making Filters Fast

- Optimizations exploiting DCG
  - runtime constants
  - fast disjunctions
- Optimizations exploiting compilation
  - atom coalescing
  - alignment optimization
  - bounds-check aggregation

Exploiting runtime constants

- Insight: many filter-specific constants
  - optimization: hard-code in instruction stream
- Example:
  - (0:16 == 1234) : encode 1234 in branch inst
- Result:
  - no lookup: fewer insts, memory refs
  - more efficient than static demultiplexer!

Fast Disjunctions

- Many filters in same protocol family:
  - compare same msg offset to different values
  - for speed, values are stored in hash table
- DPF uses DCG to compile hash table
  - exploit small tables
  - elide collision check if not needed
  - can generate specialized hash function for keys
  - again, more efficient then static demux!

Atom Coalescing

- Insight: atoms check adjacent msg fields
  - optimization: coalesce into a single check
- Example:
  - (0:16 == 0x1234) && (2:16 == 0x4321)
  - becomes (0:32 == 0x43211234)
- Result:
  - fewer branches/insts, less icache misses ...

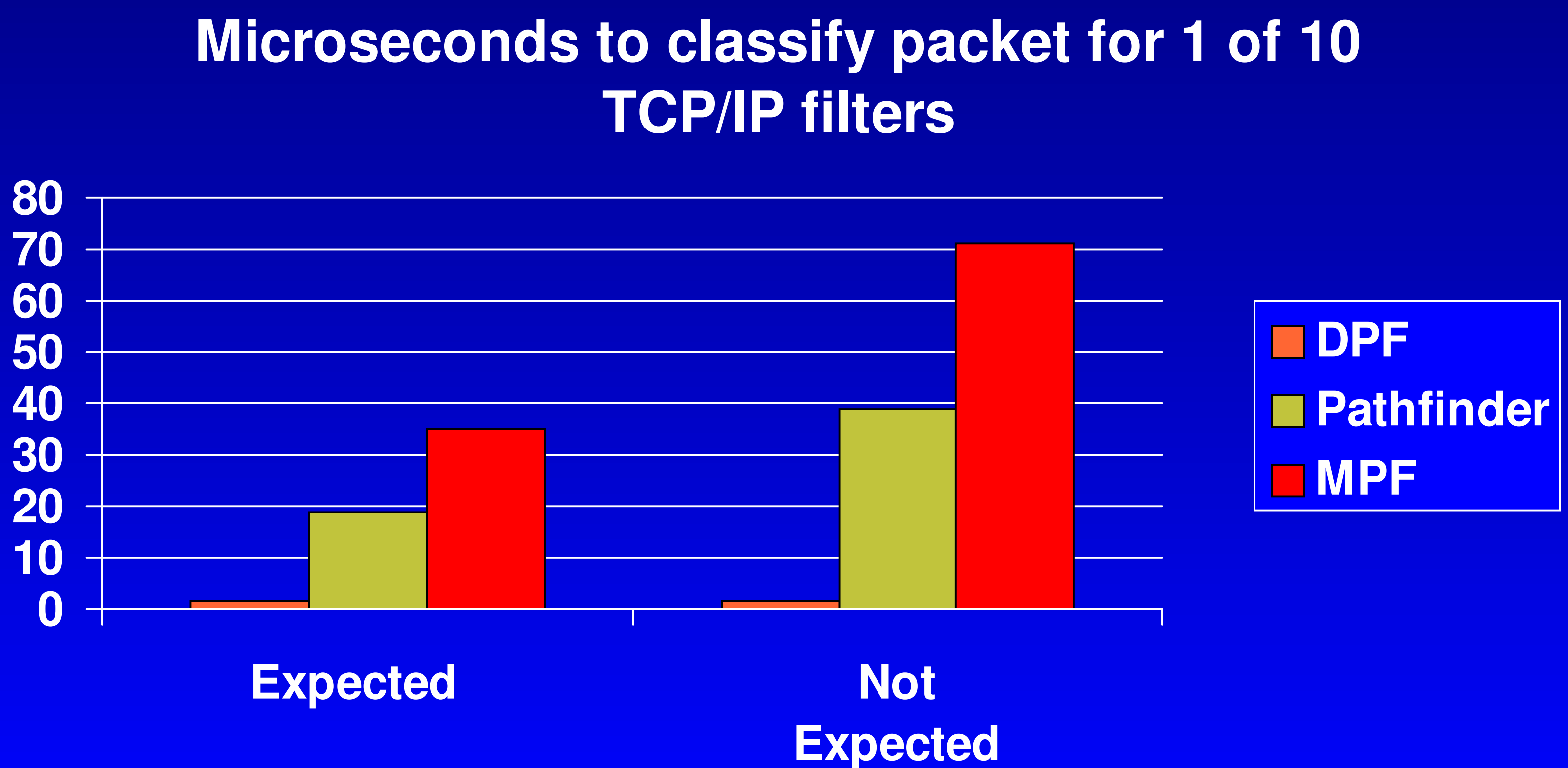
Alignment optimization

- Problem: can't trust messages or filters
  - traditional: treat every msg load as unaligned
  - slow!
- Soln: compute alignment during insertion
  - alignment starts at alignment of network buffer
  - updated after each msg ptr shift
  - msg loads generated using computed alignment
- Result: msg load costs 1 inst instead of 10

Bounds Check Aggregation

- Problem: can’t trust filter or message
  - traditional: all loads are bounds checked
  - slow!
- Soln: eliminate checks while inserting
  - statically checks loads before any shift
  - after shift, computes bound at single place
- Result: remove compare+branch on load

Classification Overhead



Insertion Overhead

- DPF: 220 usec, Pathfinder 77 usec
  - naïve implementation
- Breakeven point: 8 message classifications

User-level Networking with DPF

- Easy protocol experimentation
- Speed comparable to in-kernel networking
- 60-byte roundtrip on DECstation /Ethernet
  - User-level UDP: 309 usec
  - User-level TCP/IP: 412 usec

Experience

- Used in experimental OS for 2 years
- Current version:
  - rewrite with simpler language: 2000 lines
  - runs on sparc, mips, alpha, x86
- Available software:
  - DPF alpha release: email engler@lcs.mit.edu
  - vcode: http://www.pdos.lcs.mit.edu/~engler

Optimization of small languages

- Large languages are complex
  - hard, in general, to beat good hacker
- Small languages are simple, restricted
  - can frequently win by automating tedious tricks
  - bonus: don’t have to be a compiler god to do so
- Keys to optimizing DPF:
  - Declarative language = easy
  - Restricted domain = thorough

DPF Summary

- DPF is a fast packet filter engine
  - 10x faster than best in literature
  - rivals hand-crafted demultiplexors
- Speed from DCG + compiler optimizations
  - techniques applicable to other PFs