

libpcap: An Architecture and Optimization Methodology for Packet Capture

Sharkfest '11

Steve McCanne, CTO
Riverbed Technology

CS 164

- My story begins with a U.C. Berkeley course
 - Back in spring 1988, I took the compilers course in computer science at U.C. Berkeley
 - Taught by a guest lecturer from LBL
 - Van Jacobson LBL: Lawrence Berkeley Laboratory
 - Learned standard compiler topics
 - scanning, parsing, code generation, optimization
 - Took summer job in Van's group at end of term

LBL

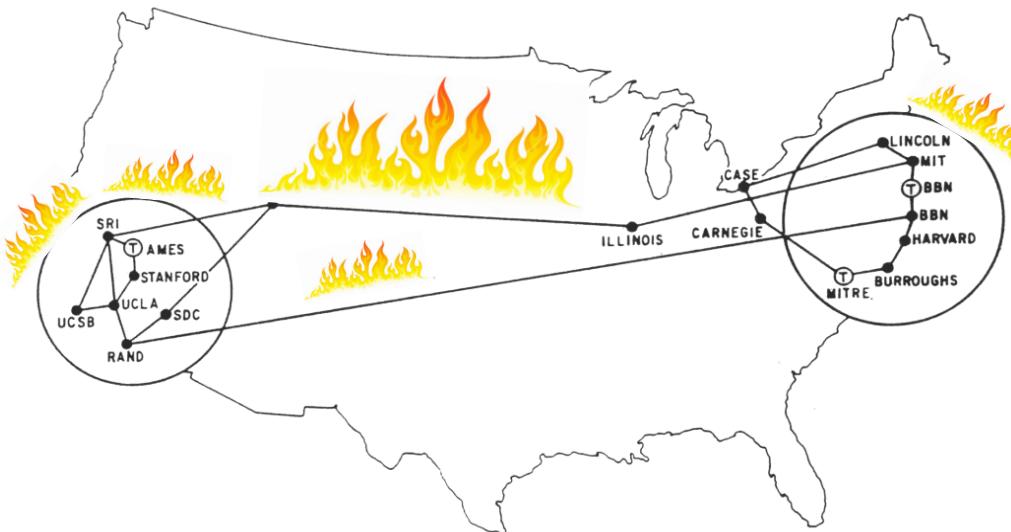
- It was a great time and place in Internet history
 - Summer job evolved into staff scientist position
- “Network Research Group”
 - Van Jacobson
 - Sally Floyd
 - Vern Paxson
 - Steve McCanne
- Lucky to be surrounded by such creative intellect

LBL Network Research Group

- flex
- TCP congestion control
- VJ header compression (CSLIP)
- BSD packet filter (BPF)
- tcpdump, pcap
- traceroute, pathchar
- BRO
- SDP/SIP
- VoIP (RTP)
- Mbone tools (vic, vat, wb)
- Scalable reliable multicast (SRM)
- ns - network simulator
- Class-based queuing (CBQ)
- Random early drop (RED)
- diffserv

Congestion Control

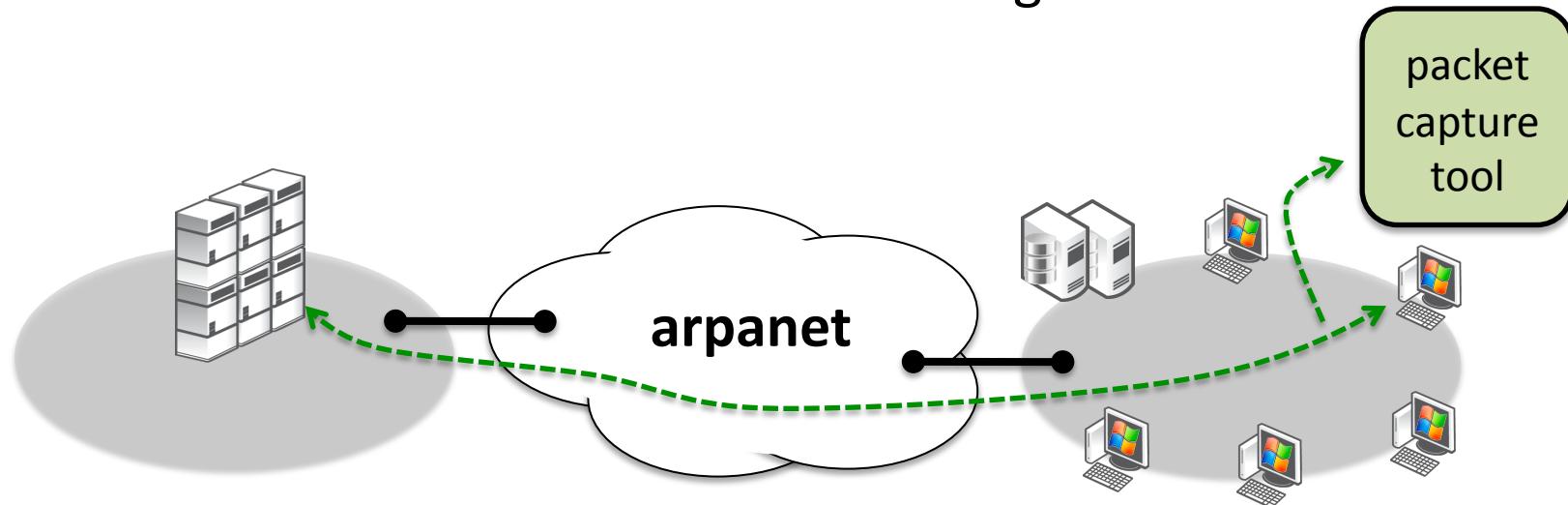
- When I first joined, Van was wrapping up his work on TCP congestion control
 - He had figured out why the Arpanet kept collapsing...



MAP 4 September 1971

Packet Capture

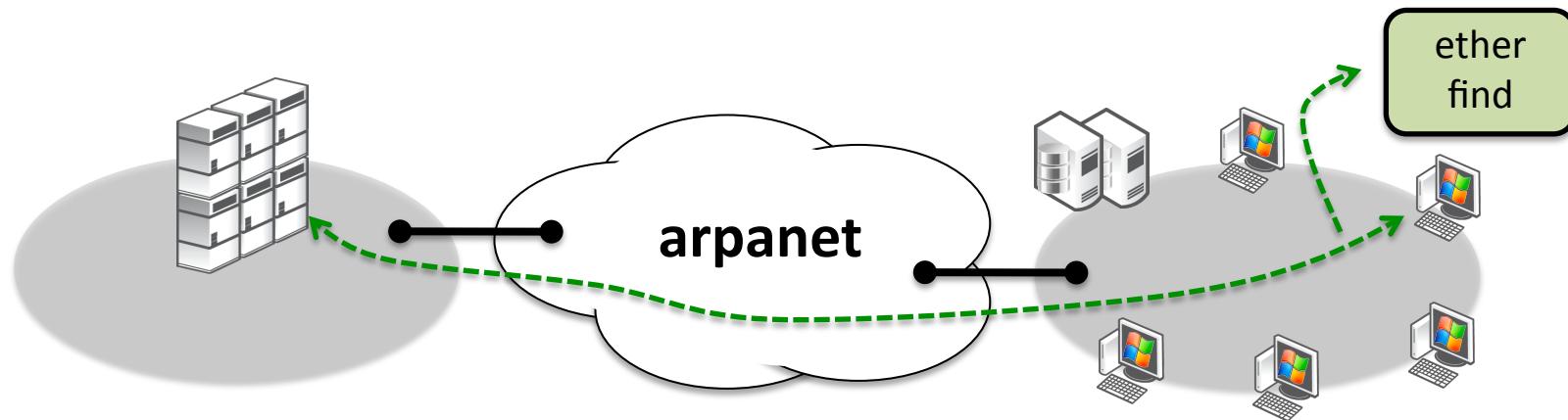
- Van needed to look at packet traces
 - to understand the problem
 - to experiment with fixes
 - to see that the solution was working



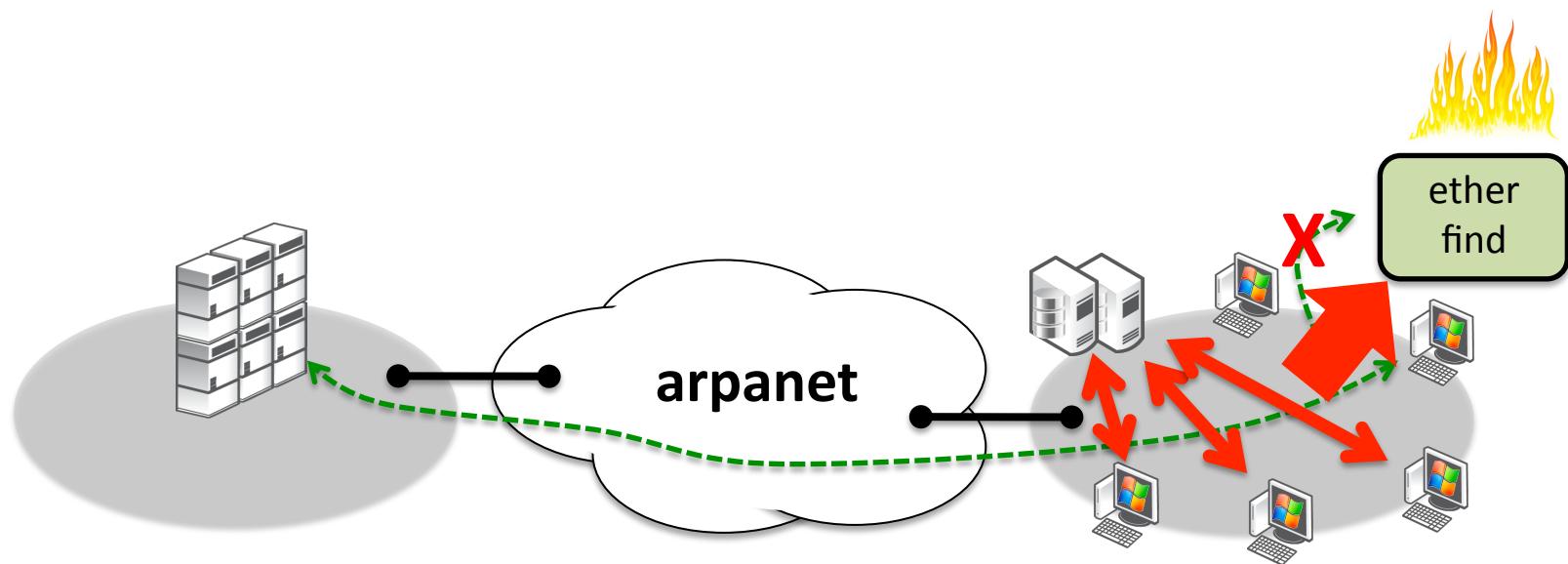
etherfind

- Frustrated with Sun’s packet capture tool
 - “etherfind” based on Unix “find” command
- Several problems
 - Clumsy filtering syntax
 - How many of you do “find . | grep ...” instead?
 - Protocol decoding was weak and cryptic
 - Horrible performance

The LAN Bottleneck



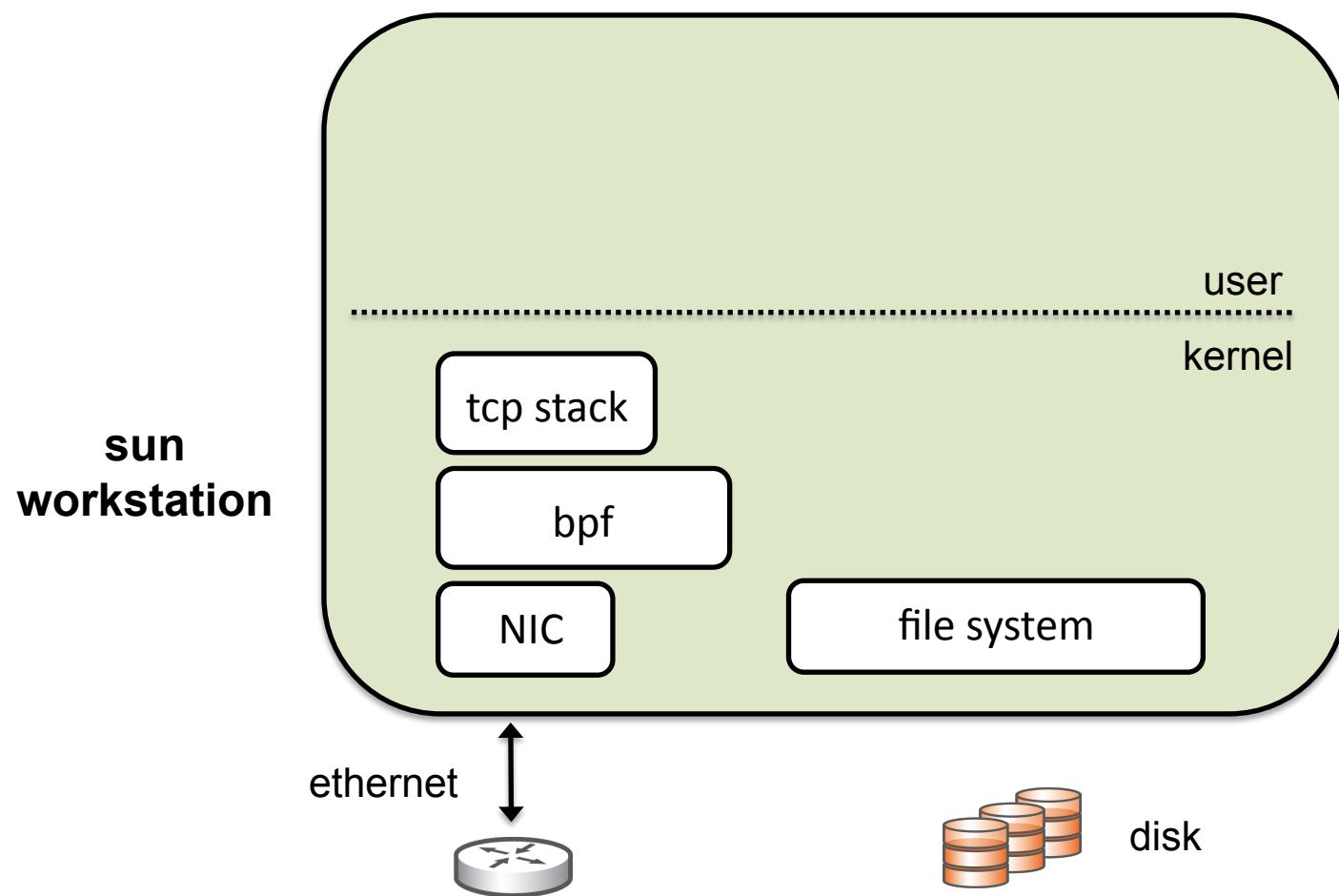
The LAN Bottleneck



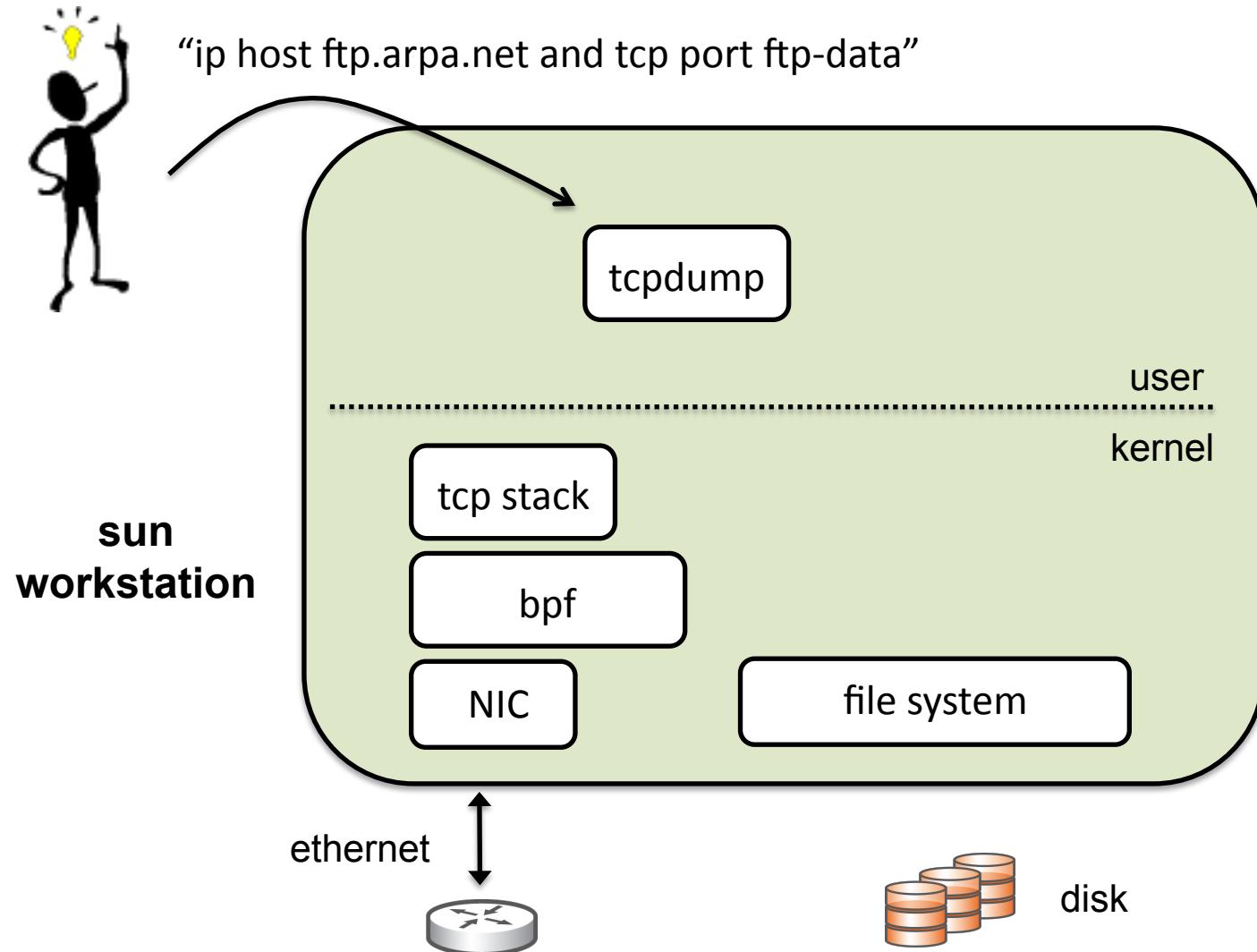
Enter tcpdump

- There must be a better way...
 - Set out to work on a new model in a tool called tcpdump
 - “Filter” packets before they come up the stack
 - Inspired by Jeff Mogul’s prior work on “enet”
 - Compile high-level filter specification into low-level code that filters packets at driver level
 - Kernel module called Berkeley Packet Filter (BPF)

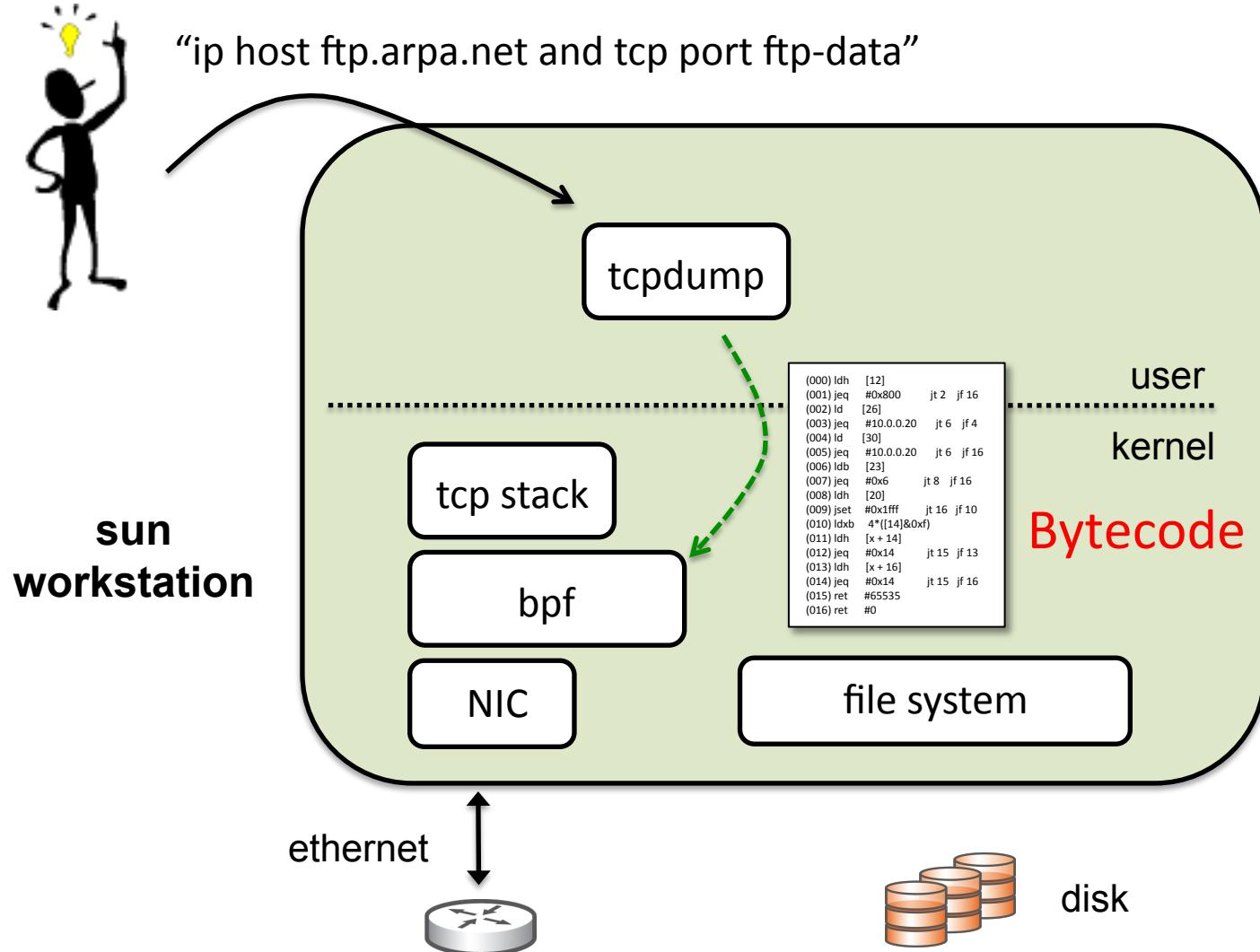
tcpdump



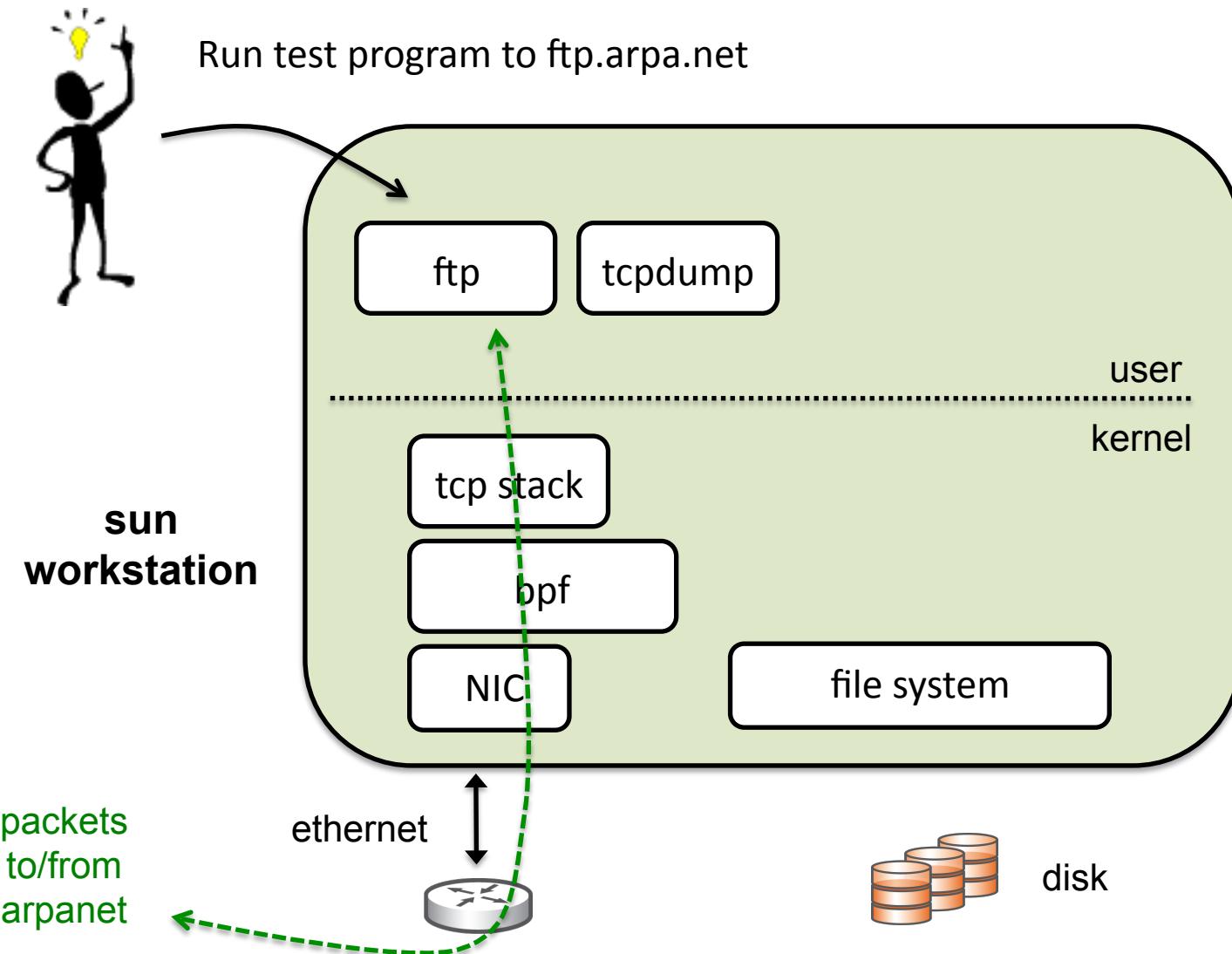
tcpdump



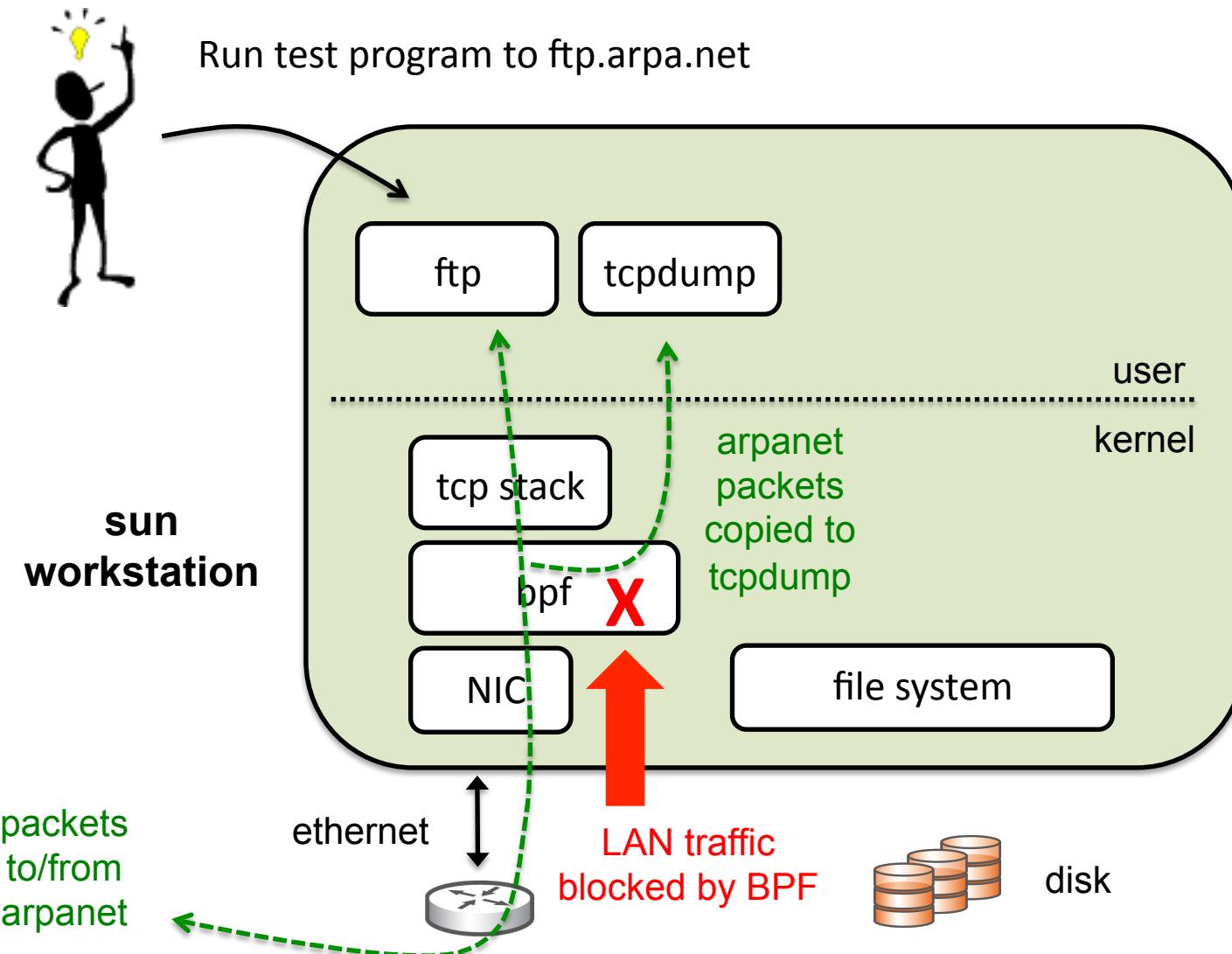
tcpdump



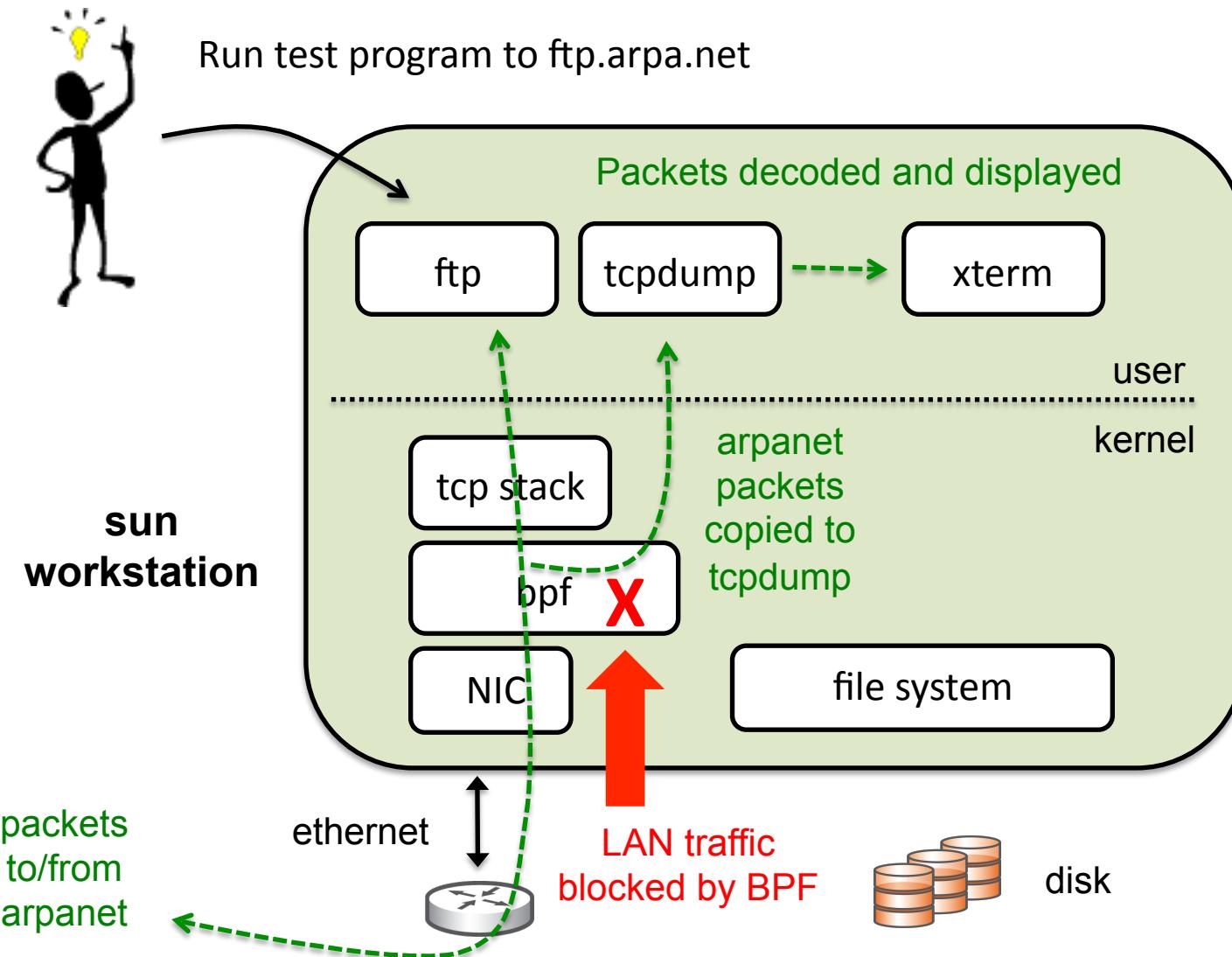
tcpdump



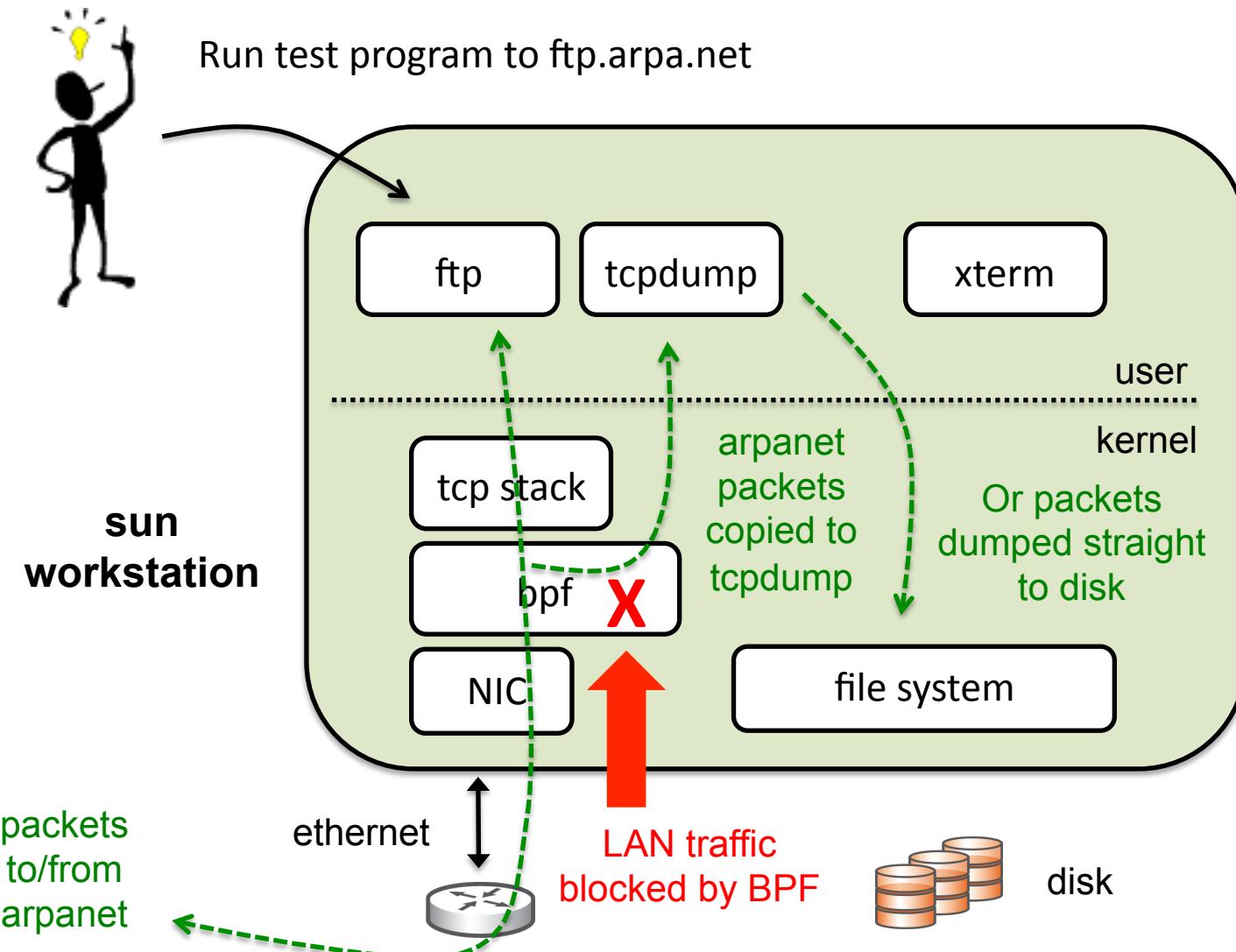
tcpdump



tcpdump



tcpdump



The BPF virtual machine

- First thing, I had to design a VM model that would run in the kernel
- Came up with a virtual machine architecture and set of machine instructions
 - Knew Apple II from my junior high days
 - Modeled after Motorola 6502
 - Accumulator (A), index register (X)
 - Packet-based memory model
 - Arithmetic and conditional logic

i.e., interpreter

FTP data packets for host 10.0.0.20

Example

```
(000) ldh [12]
(001) jeq #0x800      jt 2  jf 16
(002) ld [26]
(003) jeq #10.0.0.20  jt 6  jf 4
(004) ld [30]
(005) jeq #10.0.0.20  jt 6  jf 16
(006) ldb [23]
(007) jeq #0x6       jt 8  jf 16
(008) ldh [20]
(009) jset #0x1fff   jt 16  jf 10
(010) ldxb 4*([14]&0xf)
(011) ldh [x + 14]
(012) jeq #0x14     jt 15  jf 13
(013) ldh [x + 16]
(014) jeq #0x14     jt 15  jf 16
(015) ret #65535
(016) ret #0
```

Example

DMAC 6B + SMAC 6B = 12B
(Ethertype 2B)

FTP data packets
for host 10.0.0.20

Is ethernet type IP?

A:

X:

packet

(000) ldh	[12]	
(001) jeq	#0x800	jt 2 jf 16
(002) ld	[26]	
(003) jeq	#10.0.0.20	jt 6 jf 4
(004) ld	[30]	
(005) jeq	#10.0.0.20	jt 6 jf 16
(006) ldb	[23]	
(007) jeq	#0x6	jt 8 jf 16
(008) ldh	[20]	
(009) jset	#0x1fff	jt 16 jf 10
(010) idxb	4*([14]&0xf)	
(011) ldh	[x + 14]	
(012) jeq	#0x14	jt 15 jf 13
(013) ldh	[x + 16]	
(014) jeq	#0x14	jt 15 jf 16
(015) ret	#65535	
(016) ret	#0	



Example

FTP data packets
for host 10.0.0.20

Is ethernet type IP?

A: 0x800 X:

Idh [12]
(IP: 0x800)

packet



```
(000) ldh      [12]
(001) jeq      #0x800      jt 2  jf 16
(002) ld      [26]
(003) jeq      #10.0.0.20   jt 6  jf 4
(004) ld      [30]
(005) jeq      #10.0.0.20   jt 6  jf 16
(006) ldb      [23]
(007) jeq      #0x6        jt 8  jf 16
(008) ldh      [20]
(009) jset     #0x1fff    jt 16  jf 10
(010) ldxb    4*([14]&0xf)
(011) ldh      [x + 14]
(012) jeq      #0x14      jt 15  jf 13
(013) ldh      [x + 16]
(014) jeq      #0x14      jt 15  jf 16
(015) ret      #65535
(016) ret      #0
```

Example

L2 14B + L3 12B = 26B

FTP data packets
for host 10.0.0.20

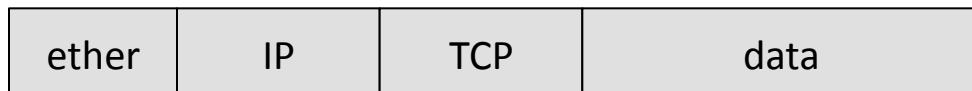
Is IP src address 10.0.0.20?

A:

X:

packet

(000) ldh	[12]		
(001) jeq	#0x800	jt 2	jf 16
(002) ld	[26]		
(003) jeq	#10.0.0.20	jt 6	jf 4
(004) ld	[30]		
(005) jeq	#10.0.0.20	jt 6	jf 16
(006) ldb	[23]		
(007) jeq	#0x6	jt 8	jf 16
(008) ldh	[20]		
(009) jset	#0xffff	jt 16	jf 10
(010) idxb	4*([14]&0xf)		
(011) ldh	[x + 14]		
(012) jeq	#0x14	jt 15	jf 13
(013) ldh	[x + 16]		
(014) jeq	#0x14	jt 15	jf 16
(015) ret	#65535		
(016) ret	#0		



Example

L2 14B + L3 12B = 26B

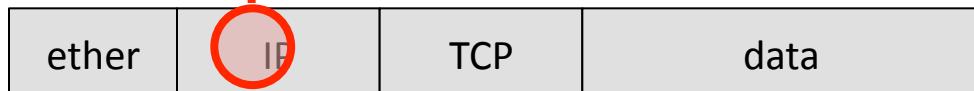
FTP data packets
for host 10.0.0.20

Is IP src address 10.0.0.20?

A: 10.0.1.11

X:

packet



(000)	ldh	[12]		
(001)	jeq	#0x800	jt 2	jf 16
(002)	ld	[26]		
(003)	jeq	#10.0.0.20	jt 6	jf 4
(004)	ld	[30]		
(005)	jeq	#10.0.0.20	jt 6	jf 16
(006)	ldb	[23]		
(007)	jeq	#0x6	jt 8	jf 16
(008)	ldh	[20]		
(009)	jset	#0xffff	jt 16	jf 10
(010)	ldxb	4*([14]&0xf)		
(011)	ldh	[x + 14]		
(012)	jeq	#0x14	jt 15	jf 13
(013)	ldh	[x + 16]		
(014)	jeq	#0x14	jt 15	jf 16
(015)	ret	#65535		
(016)	ret	#0		

Example

L2 14B + L3 12B + L3 S-IP 4B = 30B

FTP data packets
for host 10.0.0.20

Is IP dst address 10.0.0.20?

A:

X:

packet

(000) ldh	[12]		
(001) jeq	#0x800	jt 2	jf 16
(002) ld	[26]		
(003) jeq	#10.0.0.20	jt 6	jf 4
(004) ld	[30]		
(005) jeq	#10.0.0.20	jt 6	jf 16
(006) ldb	[23]		
(007) jeq	#0x6	jt 8	jf 16
(008) ldh	[20]		
(009) jset	#0xffff	jt 16	jf 10
(010) idxb	4*([14]&0xf)		
(011) ldh	[x + 14]		
(012) jeq	#0x14	jt 15	jf 13
(013) ldh	[x + 16]		
(014) jeq	#0x14	jt 15	jf 16
(015) ret	#65535		
(016) ret	#0		



Example

L2 14B + L3 12B + L3 S-IP 4B = 30B

FTP data packets
for host 10.0.0.20

Is IP dst address 10.0.0.20?

A: 10.0.0.20

X:

packet



(000)	ldh	[12]		
(001)	jeq	#0x800	jt 2	jf 16
(002)	ld	[26]		
(003)	jeq	#10.0.0.20	jt 6	jf 4
(004)	ld	[30]		
(005)	jeq	#10.0.0.20	jt 6	jf 16
(006)	ldb	[23]		
(007)	jeq	#0x6	jt 8	jf 16
(008)	ldh	[20]		
(009)	jset	#0xffff	jt 16	jf 10
(010)	ldxb	4*([14]&0xf)		
(011)	ldh	[x + 14]		
(012)	jeq	#0x14	jt 15	jf 13
(013)	ldh	[x + 16]		
(014)	jeq	#0x14	jt 15	jf 16
(015)	ret	#65535		
(016)	ret	#0		

Example

FTP data packets
for host 10.0.0.20

Is IP protocol TCP?

A: X:

packet

L2 14B + L3 8B +L3 TTL1B = 23B

(000) ldh	[12]		
(001) jeq	#0x800	jt 2	jf 16
(002) ld	[26]		
(003) jeq	#10.0.0.20	jt 6	jf 4
(004) ld	[30]		
(005) jeq	#10.0.0.20	jt 6	jf 16
(006) ldb	[23]		
(007) jeq	#0x6	jt 8	jf 16
(008) ldh	[20]		
(009) jset	#0x1fff	jt 16	jf 10
(010) idxb	4*([14]&0xf)		
(011) ldh	[x + 14]		
(012) jeq	#0x14	jt 15	jf 13
(013) ldh	[x + 16]		
(014) jeq	#0x14	jt 15	jf 16
(015) ret	#65535		
(016) ret	#0		



Example

FTP data packets
for host 10.0.0.20

Is IP protocol TCP?

A: 6 X:

packet



```
(000) ldh [12]
(001) jeq #0x800      jt 2  jf 16
(002) ld [26]
(003) jeq #10.0.0.20   jt 6  jf 4
(004) ld [30]
(005) jeq #10.0.0.20   jt 6  jf 16
(006) ldb [23]
(007) jeq #0x6         jt 8  jf 16
(008) ldh [20]
(009) jset #0x1fff     jt 16  jf 10
(010) idxb 4*([14]&0xf)
(011) ldh [x + 14]
(012) jeq #0x14        jt 15  jf 13
(013) ldh [x + 16]
(014) jeq #0x14        jt 15  jf 16
(015) ret #65535
(016) ret #0
```

Example

L2 14B + L3 6B = 20B

FTP data packets
for host 10.0.0.20

Is it first or only frag?

A: X:

packet

(000) ldh	[12]	
(001) jeq	#0x800	jt 2 jf 16
(002) ld	[26]	
(003) jeq	#10.0.0.20	jt 6 jf 4
(004) ld	[30]	
(005) jeq	#10.0.0.20	jt 6 jf 16
(006) ldb	[23]	
(007) jeq	#0x6	jt 8 jf 16
(008) ldh	[20]	
(009) jset	#0x1fff	jt 16 jf 10
(010) ldxb	4*([14]&0xf)	
(011) ldh	[x + 14]	
(012) jeq	#0x14	jt 15 jf 13
(013) ldh	[x + 16]	
(014) jeq	#0x14	jt 15 jf 16
(015) ret	#65535	
(016) ret	#0	

More frag bit set?



Example

FTP data packets
for host 10.0.0.20

Is it first or only frag?

frag bit not set

A: 0 X:

packet



```
(000) ldh [12]
(001) jeq #0x800      jt 2  jf 16
(002) ld [26]
(003) jeq #10.0.0.20   jt 6  jf 4
(004) ld [30]
(005) jeq #10.0.0.20   jt 6  jf 16
(006) ldb [23]
(007) jeq #0x6        jt 8  jf 16
(008) ldh [20]
(009) jset #0xffff    jt 16  jf 10
(010) ldxb 4*([14]&0xf)
(011) ldh [x + 14]
(012) jeq #0x14       jt 15 jf 13
(013) ldh [x + 16]
(014) jeq #0x14       jt 15 jf 16
(015) ret #65535
(016) ret #0
```

Idxb: load byte to x

(010) Idxb 4*([14]&0xf):Lower nibble * 4 at byte offset 14

=> IP header length = 5 x 4 = 20 (normally)

x = 20

FTP data packets
for host 10.0.0.20

Is TCP src port FTP?

A:

X:

(000) Idh [12]
(001) jeq #0x800 jt 2 jf 16
(002) Id [26]
(003) jeq #10.0.0.20 jt 6 jf 4
(004) Id [30]
(005) jeq #10.0.0.20 jt 6 jf 16
(006) IDb [23]
(007) jeq #0x6 jt 8 jf 16
(008) Idh [20]
(009) jset #0xffff jt 16 jf 10
(010) Idxb 4*([14]&0xf)
(011) Idh [x + 14]
(012) jeq #0x14 jt 15 jf 13
(013) Idh [x + 16]
(014) jeq #0x14 jt 15 jf 16

(0011) Idh [x + 14]:

* 從第 x+14 個byte開始讀16bits (half word)



* 目的: 得到 src port

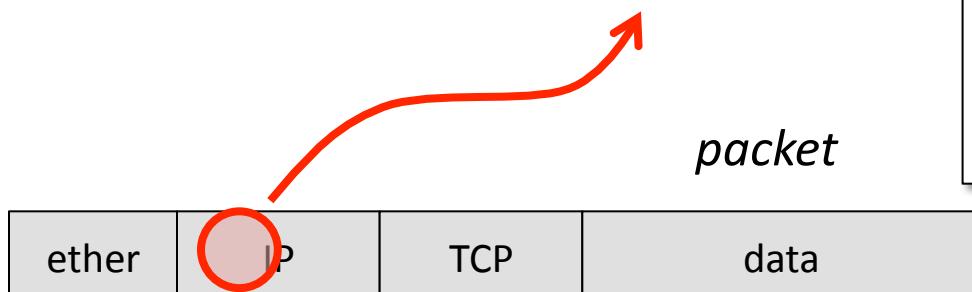
* 結果: src port 為 8377

Example

FTP data packets
for host 10.0.0.20

Is TCP src port FTP?

A: X:



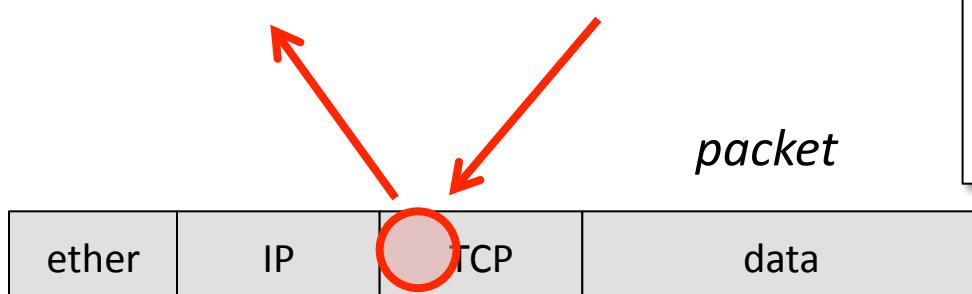
```
(000) ldh [12]
(001) jeq #0x800      jt 2  jf 16
(002) ld [26]
(003) jeq #10.0.0.20  jt 6  jf 4
(004) ld [30]
(005) jeq #10.0.0.20  jt 6  jf 16
(006) ldb [23]
(007) jeq #0x6        jt 8  jf 16
(008) ldh [20]
(009) jset #0xffff    jt 16  jf 10
(010) idxb 4*([14]&0xf)
(011) ldh [x + 14]
(012) jeq #0x14      jt 15  jf 13
(013) ldh [x + 16]
(014) jeq #0x14      jt 15  jf 16
(015) ret #65535
(016) ret #0
```

Example

FTP data packets
for host 10.0.0.20

Is TCP src port FTP?

A: 8377 X: 20



packet

```
(000) ldh [12]
(001) jeq #0x800      jt 2  jf 16
(002) ld [26]
(003) jeq #10.0.0.20  jt 6  jf 4
(004) ld [30]
(005) jeq #10.0.0.20  jt 6  jf 16
(006) ldb [23]
(007) jeq #0x6        jt 8  jf 16
(008) ldh [20]
(009) jset #0xffff    jt 16  jf 10
(010) idxb 4*([14]&0xf)
(011) ldh [x + 14]
(012) jeq #0x14      jt 15  jf 13
(013) ldh [x + 16]
(014) jeq #0x14      jt 15  jf 16
(015) ret #65535
(016) ret #0
```

Example

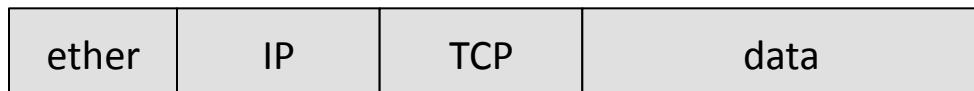
FTP data packets
for host 10.0.0.20

Is TCP dest port FTP?

A: 8377 X: 20

packet

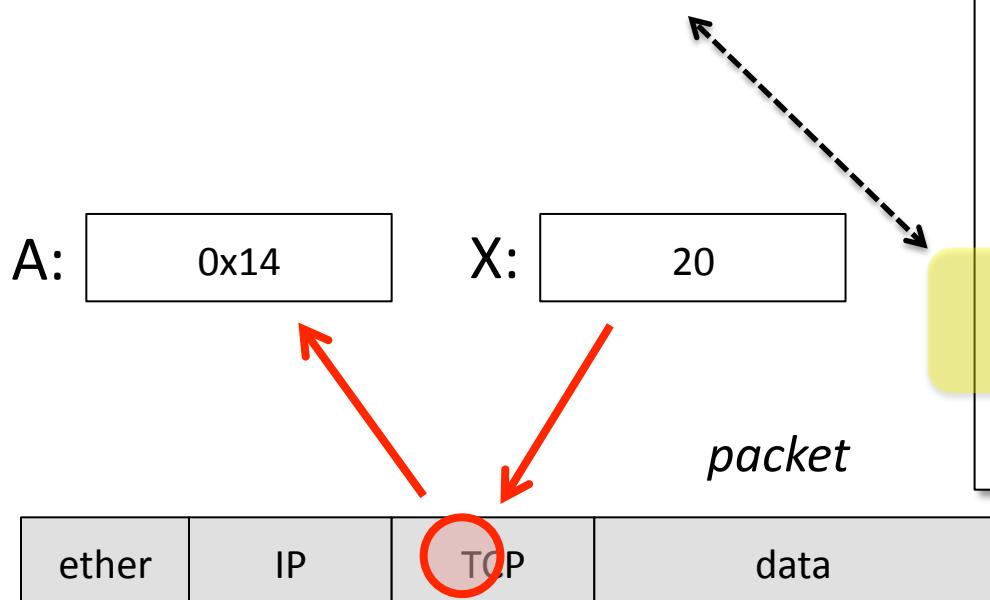
```
(000) ldh [12]
(001) jeq #0x800      jt 2  jf 16
(002) ld [26]
(003) jeq #10.0.0.20  jt 6  jf 4
(004) ld [30]
(005) jeq #10.0.0.20  jt 6  jf 16
(006) ldb [23]
(007) jeq #0x6        jt 8  jf 16
(008) ldh [20]
(009) jset #0x1fff    jt 16  jf 10
(010) ldxb 4*([14]&0xf)
(011) ldh [x + 14]
(012) jeq #0x14       jt 15  jf 13
(013) ldh [x + 16]
(014) jeq #0x14       jt 15  jf 16
(015) ret #65535
(016) ret #0
```



Example

FTP data packets
for host 10.0.0.20

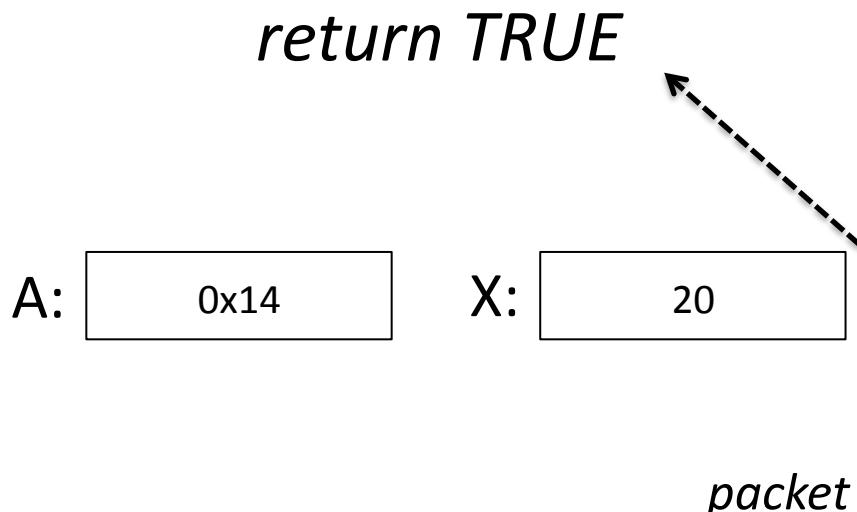
Is TCP dest port FTP?



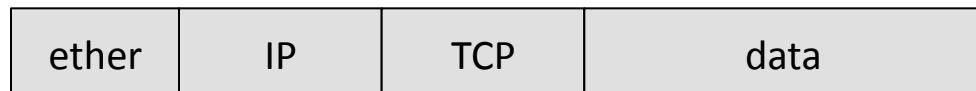
```
(000) ldh [12]
(001) jeq #0x800      jt 2  jf 16
(002) ld [26]
(003) jeq #10.0.0.20  jt 6  jf 4
(004) ld [30]
(005) jeq #10.0.0.20  jt 6  jf 16
(006) ldb [23]
(007) jeq #0x6        jt 8  jf 16
(008) ldh [20]
(009) jset #0xffff    jt 16  jf 10
(010) idxb 4*([14]&0xf)
(011) ldh [x + 14]
(012) jeq #0x14       jt 15  jf 13
(013) ldh [x + 16]
(014) jeq #0x14       jt 15  jf 16
(015) ret #65535
(016) ret #0
```

Example

FTP data packets
for host 10.0.0.20



```
(000) ldh    [12]
(001) jeq    #0x800      jt 2  jf 16
(002) ld     [26]
(003) jeq    #10.0.0.20   jt 6  jf 4
(004) ld     [30]
(005) jeq    #10.0.0.20   jt 6  jf 16
(006) ldb    [23]
(007) jeq    #0x6        jt 8  jf 16
(008) ldh    [20]
(009) jset   #0xffff     jt 16  jf 10
(010) idxb   4*([14]&0xf)
(011) ldh    [x + 14]
(012) jeq    #0x14       jt 15  jf 13
(013) ldh    [x + 16]
(014) jeq    #0x14       jt 15  jf 16
(015) ret    #65535
(016) ret    #0
```



The Challenge

- The BPF virtual machine model is a very flexible and efficient model for packet filtering
- But, you would never want to write low-level BPF programs every time you wanted to filter packets
- So, we needed a higher level model...

A Filter Language

Instead of writing this...



```
(000) ldh    [12]
(001) jeq    #0x800      jt 2  jf 16
(002) ld     [26]
(003) jeq    #10.0.0.20   jt 6  jf 4
(004) ld     [30]
(005) jeq    #10.0.0.20   jt 6  jf 16
(006) ldb    [23]
(007) jeq    #0x6       jt 8  jf 16
(008) ldh    [20]
(009) jset   #0x1fff     jt 16  jf 10
(010) idxb   4*([14]&0xf)
(011) ldh    [x + 14]
(012) jeq    #0x14       jt 15  jf 13
(013) ldh    [x + 16]
(014) jeq    #0x14       jt 15  jf 16
(015) ret    #65535
(016) ret    #0
```

A Filter Language

ip host ftp.arpa.net and tcp port ftp-data

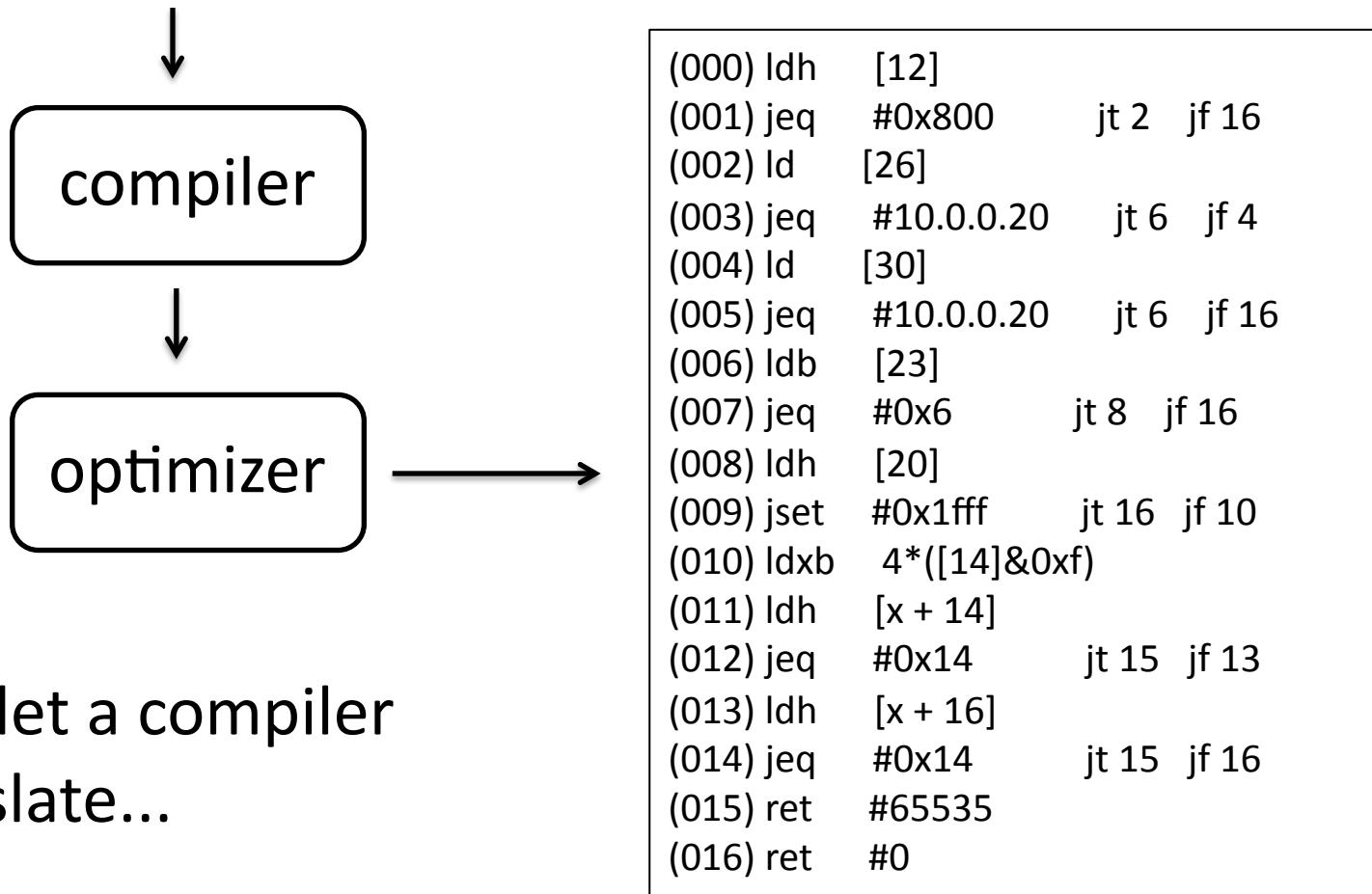
Just say this...



```
(000) ldh [12]
(001) jeq #0x800 jt 2 jf 16
(002) ld [26]
(003) jeq #10.0.0.20 jt 6 jf 4
(004) ld [30]
(005) jeq #10.0.0.20 jt 6 jf 16
(006) ldb [23]
(007) jeq #0x6 jt 8 jf 16
(008) ldh [20]
(009) jset #0x1fff jt 16 jf 10
(010) idxb 4*([14]&0xf)
(011) ldh [x + 14]
(012) jeq #0x14 jt 15 jf 13
(013) ldh [x + 16]
(014) jeq #0x14 jt 15 jf 16
(015) ret #65535
(016) ret #0
```

A Filter Language

ip host ftp.arpa.net and tcp port ftp-data



And let a compiler
translate...

The Challenge

- This is where things got a bit tricky
- Designing the language and parser so it was easy on users turned out to be hard
- Learned an important life lesson from Van
 - It's easy to make things hard
 - It's hard to make things easy
 - It's usually better to do the latter

BPF Language

- The BPF filter language starts from a basic predicate, which is true iff the specified packet field equals the indicated value

pred: field val

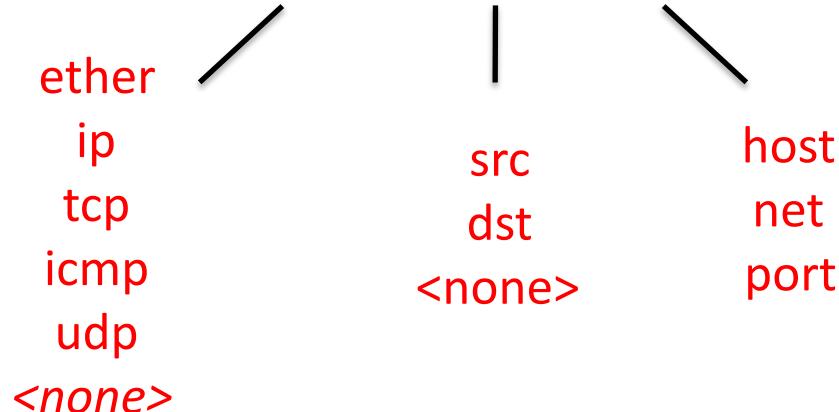
field: protocol dir selector

BPF Language

- The BPF filter language starts from a basic predicate, which is true iff the specified packet field equals the indicated value

pred: field val

field: protocol dir selector



BPF Language

- The BPF filter language starts from a basic predicate, which is true iff the specified packet field equals the indicated value

pred: field val

examples

field: protocol dir selector

ether

ip

tcp

icmp

udp

<none>

src

dst

<none>

host

net

port

ip src host 10.0.0.1

tcp dst port 80

BPF Logic

- Language includes logic to stitch together predicates into complex logic operations
 - *pred or pred*
 - *pred and pred*
 - **not** *pred*
 - ‘(‘ *pred* ‘)’

ip src host X and not port 80

My First Attempt

expr: pred

- | *expr AND pred*
- | *expr OR pred*
- | NOT *expr*
- | ('' *expr* '')

pred: field val

field: protocol dir selector

etc...

A Problem

- But Van didn't like it... too clunky...

ip src host X or ip src host Y or ip src host Z

A Problem

- But Van didn't like it... too clunky...

ip src host X or ip src host Y or ip src host Z

A Problem

- But Van didn't like it... too clunky...

ip src host X or ip src host Y or ip src host Z

- Why not just say...

ip src host X or Y or Z

- This should be easy enough to fix...

My Second Attempt

- Introduce two layers of logic
 - Lower layer would handle predicates with multiple values
 - ip host x or y
 - tcp port 80 or 1024
 - Upper layer would handle the combinations of the lower-layer expressions
 - (ip host x or y) and (tcp port 80 or 1024)

My Second Attempt

expr: term

term: pred

| *expr AND term*

| *term AND val*

| *expr OR term*

| *term OR val*

| *NOT expr*

| *NOT term*

| *('expr')*

| *('term ')*

pred: field val

field: protocol dir selector

etc...

The Second Problem

- But this didn't work at all
 - the parser needs to decide to parse as a *term*
ip src host x or y and z
 - or parse input as an *expr*
ip src host x or y and tcp port z
 - when the partial input didn't provide enough info

ip src host x or y and

unknown

parsed input

look-ahead

It's easy to make things hard

- Some easy ways out...
 - require parens or another grouping symbol
 - ip host (x or y) and tcp port z
 - ip host { x or y } and tcp port z
 - have different families of logic symbols
 - e.g., “and”, “AND”, “or”, “OR”
 - ip host x or y AND tcp port z
 - introduce terminator symbol
 - ip host x or y . and tcp port z

It's hard to make things easy

- But all those solutions made things harder on the user, even though they were easy outs
 - So, Van challenged me
 - “There must be a way. Figure it out.”
- I spent a week or two frustratingly thinking about it and finally the light bulb came on
 - Turns out this was a novel language construct

The Solution

- Have a single level of logic, not two
- Allow predicates *or* values to be tacked onto an expression
- i.e., an *expr* can be both
 - *expr AND pred*
 - *expr AND val*

Third Time's a Charm

expr: pred

- | *expr AND pred*
- | *expr AND val*
- | *expr OR pred*
- | *expr OR val*
- | *NOT expr*
- | *('expr')*

pred: field val

Not so fast...

- Ok, this grammar worked out fine, but now code generation became tricky
- Fortunately, this problem while tricky, had a solution...

My Third Attempt

expr: pred

- | *expr AND pred*
- | *expr AND val*
- | *expr OR pred*
- | *expr OR val*
- | *NOT expr*
- | *('expr')*

pred: field val

{ \$\$ = gen_cmp(\$1, \$2); }

My Third Attempt

expr: pred

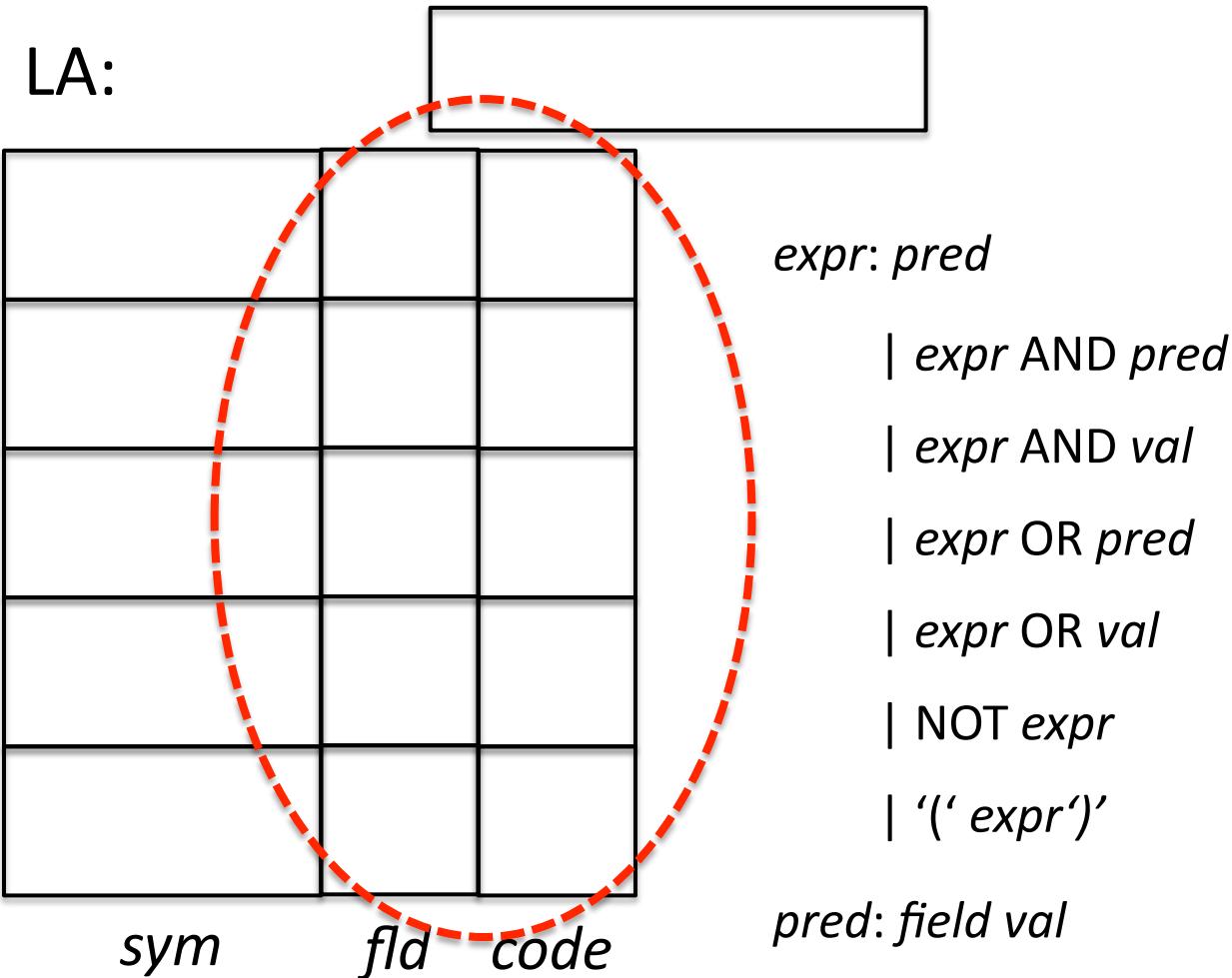
<i>expr AND pred</i>	{ \$\$ = gen_and(\$1,\$3); }
<i>expr AND val</i>	{ ??? }
<i>expr OR pred</i>	{ \$\$ = gen_or(\$1, \$3); }
<i>expr OR val</i>	{ ??? }
NOT <i>expr</i>	{ \$\$ = gen_not(\$2); }
'(' <i>expr</i>)'	{ \$\$ = \$2; }

pred: field val

{ \$\$ = gen_cmp(\$1, \$2); }



The Solution



My Third Attempt

expr: pred

- | *expr AND pred*
- | *expr AND val*
- | *expr OR pred*
- | *expr OR val*
- | NOT *expr*
- | (' *expr*)'

```
{ t = gen_cmp($1.fld, $3);
$$ .code = gen_and($1, t)
$$ .fld = $1.fld; }
```

pred: field val

```
{ $$ .code = gen_cmp($1, $2);
$$ .fld = $1; }
```

My Third Attempt

expr: pred

| *expr AND pred*

| *expr AND val*

| *expr OR pred*

| *expr OR val*

| NOT *expr*

| (' *expr*)'

{ t = gen_eq(\$1.fld, \$3);
\$\$.code = gen_and(\$1, t)
\$\$.fld = \$1.fld; }

pred: field val

{ \$\$.code = gen_cmp(\$1, \$2);
\$\$.fld = \$1; }

My Third Attempt

expr: pred

- | *expr AND pred*
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred*
- | *expr OR val*
- | NOT *expr*
- | (' *expr*)'

pred: field val

{ \$\$ = gen_cmp(\$1, \$2); }

My Third Attempt

expr: pred

<i>expr AND pred</i>	{ \$\$ = gen_and(\$1,\$3); }
<i>expr AND val</i>	{ \$\$ = gen_vand(\$1, \$3); }
<i>expr OR pred</i>	{ \$\$ = gen_or(\$1, \$3); }
<i>expr OR val</i>	{ \$\$ = gen_vor(\$1, \$3); }
NOT <i>expr</i>	{ \$\$ = gen_not(\$2); }
'(' <i>expr</i>)'	{ \$\$ = \$2; }

pred: field val

{ \$\$ = gen_cmp(\$1, \$2); }

Example

LA: IP

ip src host x or y and tcp dst port z

sym fld code

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: SRC

src host x or y and tcp dst port z

IP		

sym fld code

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: HOST			host x or y and tcp dst port z
SRC			
IP			
			<i>expr: pred</i>
			<i>expr AND pred</i> { \$\$ = gen_and(\$1,\$3); }
			<i>expr AND val</i> { \$\$ = gen_vand(\$1, \$3); }
			<i>expr OR pred</i> { \$\$ = gen_or(\$1, \$3); }
			<i>expr OR val</i> { \$\$ = gen_vor(\$1, \$3); }
			<i>pred: field val</i> { \$\$ = gen_cmp(\$1, \$2); }
			<i>field: proto dir selector</i>

sym *fld* *code*

Example

LA: X

x or y and tcp dst port z

HOST		
SRC		
IP		
<i>sym</i>	<i>fld</i>	<i>code</i>

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: X

x or y and tcp dst port z

<i>field</i>	ISH	
<i>sym</i>	<i>fld</i>	<i>code</i>

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: X

x or y and tcp dst port z

<i>field</i>	ISH	
<i>sym</i>		

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: OR

or y and tcp dst port z

$val(x)$		
<i>field</i>	ISH	
<i>sym</i>	<i>fld</i>	<i>code</i>

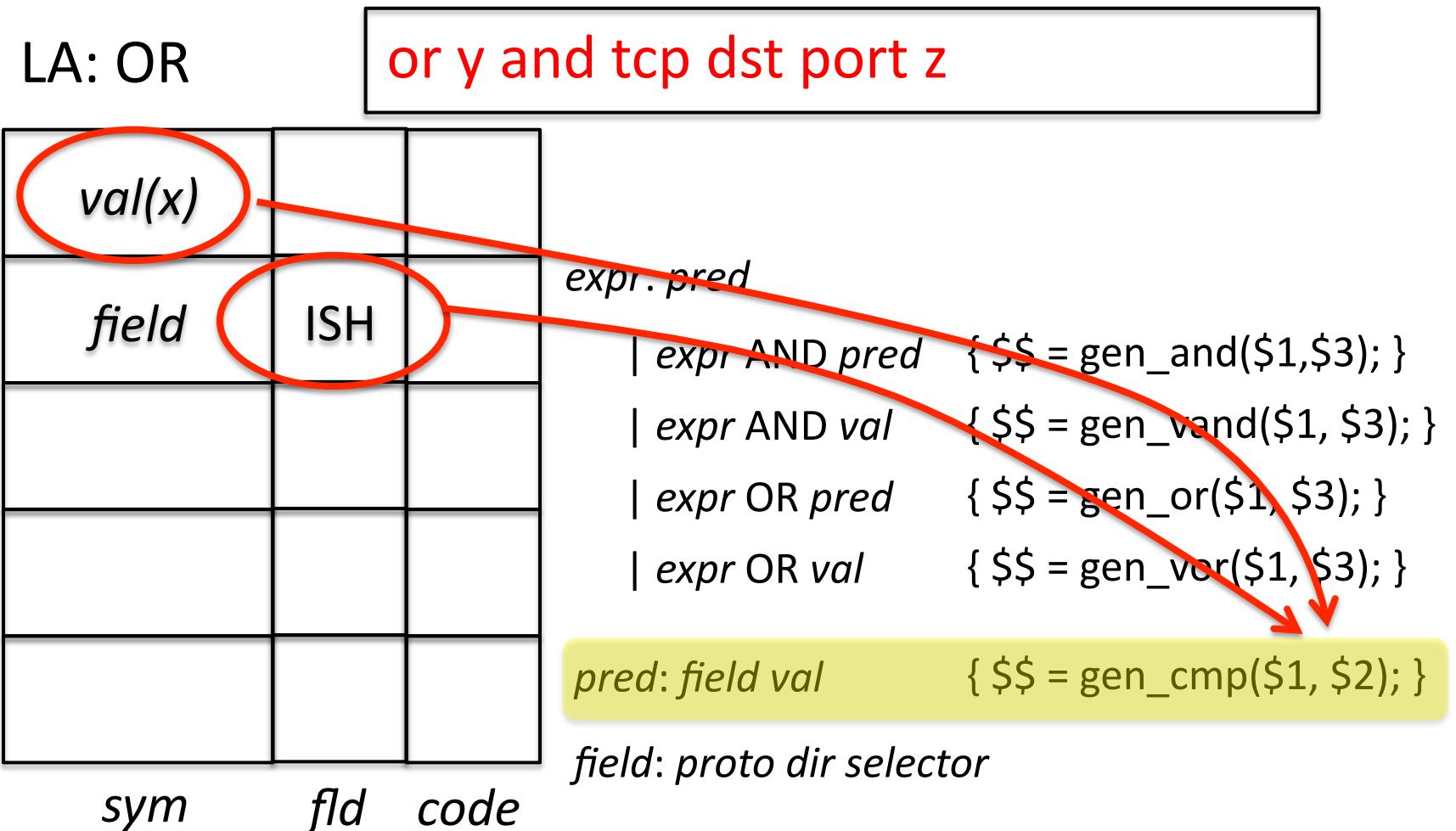
expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example



Example

LA: OR

or y and tcp dst port z

<i>pred</i>	ISH	<i>C₁</i>
<i>sym</i>		
<i>fld</i>		
<i>code</i>		

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: OR

or y and tcp dst port z

<i>pred</i>	ISH	<i>C₁</i>
<i>sym</i>	<i>fld</i>	<i>code</i>

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: OR

or y and tcp dst port z

<i>expr</i>	ISH	<i>C₁</i>
<i>sym</i>	<i>fld</i>	<i>code</i>

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: Y	y and tcp dst port z	
OR		
<i>expr</i>	ISH	<i>C₁</i>
<i>sym</i>	<i>fld</i>	<i>code</i>

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: AND			and tcp dst port z
val(y)			
OR			
expr	ISH	C_1	

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: AND			and tcp dst port z
val(y)			
OR			
expr	ISH	C_1	<i>expr: pred</i> <i>expr AND pred</i> { \$\$ = gen_and(\$1,\$3); } <i>expr AND val</i> { \$\$ = gen_vand(\$1, \$3); } <i>expr OR pred</i> { \$\$ = gen_or(\$1, \$3); } <i>expr OR val</i> { \$\$ = gen_vor(\$1, \$3); }
sym			<i>pred: field val</i> { \$\$ = gen_cmp(\$1, \$2); } <i>field: proto dir selector</i>
fld			
code			

Example

LA: AND **and tcp dst port z**

<i>val(y)</i>		
<i>OR</i>		
<i>expr</i>	<i>ISH</i>	<i>C1</i>

\$3

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: AND	and tcp dst port z	
expr	ISH	C_2
sym	fld	code

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: TCP

tcp dst port z

AND		
<i>expr</i>	ISH	<i>C₂</i>
<i>sym</i>	<i>fld</i>	<i>code</i>

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: DST			dst port z
TCP			
AND			
<i>expr</i>	ISH	<i>C₂</i>	

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: PORT	port z	
DST		
TCP		
AND		
<i>expr</i>	ISH	C_2
<i>sym</i>	<i>fld</i>	<i>code</i>

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: Z	Z	
PORT		
DST		
TCP		
AND		
<i>expr</i>	ISH	<i>C₂</i>

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: Z		
<i>field</i>	TDP	
AND		
<i>expr</i>	ISH	<i>C₂</i>

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: <eof>

$val(z)$		
$field$	TDP	
AND		
$expr$	ISH	C_2
sym	fld	$code$

expr: pred

- | $expr \text{ AND } pred$ { \$\$ = \text{gen_and}(\\$1,\\$3); }
- | $expr \text{ AND } val$ { \$\$ = \text{gen_vand}(\\$1, \\$3); }
- | $expr \text{ OR } pred$ { \$\$ = \text{gen_or}(\\$1, \\$3); }
- | $expr \text{ OR } val$ { \$\$ = \text{gen_vor}(\\$1, \\$3); }

pred: field val { \$\$ = \text{gen_cmp}(\\$1, \\$2); }

field: proto dir selector

Example

sym	fld	code	
<i>val(z)</i>			
<i>field</i>	TDP		$\begin{aligned} & \text{expr: pred} \\ & \text{expr AND pred } \{ \$\$ = \text{gen_and}(\$1, \$3); \} \\ & \text{expr AND val } \{ \$\$ = \text{gen_vand}(\$1, \$3); \} \\ & \text{expr OR pred } \{ \$\$ = \text{gen_or}(\$1, \$3); \} \\ & \text{expr OR val } \{ \$\$ = \text{gen_vor}(\$1, \$3); \} \end{aligned}$
AND			
<i>expr</i>	ISH	<i>C₂</i>	<p><i>pred: field val</i> $\{ \\$\\$ = \text{gen_cmp}(\\$1, \\$2); \}$</p> <p><i>field: proto dir selector</i></p>

Example

LA: <eof>		
<i>pred</i>	TDP	C_3
AND		
<i>expr</i>	ISH	C_2

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

LA: <eof>		
<i>pred</i>	TDP	C_3
AND		
<i>expr</i>	ISH	C_2

expr: pred

- | *expr AND pred* { \$\$ = gen_and(\$1,\$3); }
- | *expr AND val* { \$\$ = gen_vand(\$1, \$3); }
- | *expr OR pred* { \$\$ = gen_or(\$1, \$3); }
- | *expr OR val* { \$\$ = gen_vor(\$1, \$3); }

pred: field val { \$\$ = gen_cmp(\$1, \$2); }

field: proto dir selector

Example

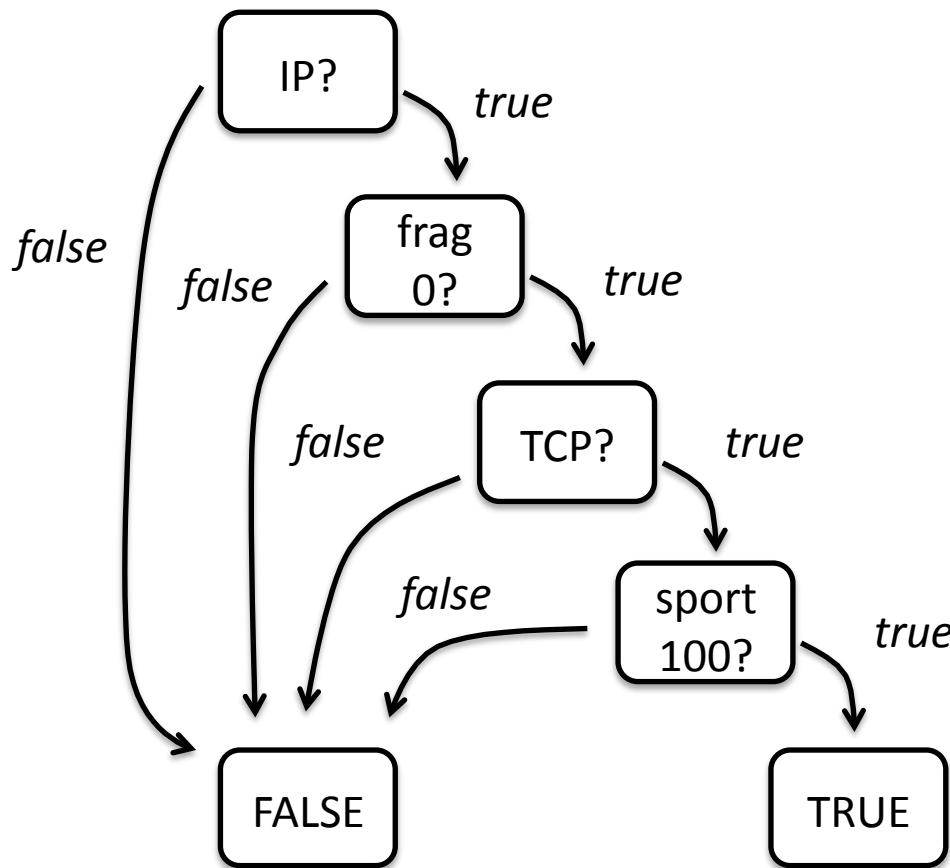
LA: <eof>

expr	ISH	C4	
			→ output BPF code
			<i>expr: pred</i>
			expr AND pred { \$\$ = gen_and(\$1,\$3); }
			expr AND val { \$\$ = gen_vand(\$1, \$3); }
			expr OR pred { \$\$ = gen_or(\$1, \$3); }
			expr OR val { \$\$ = gen_vor(\$1, \$3); }
			<i>pred: field val</i> { \$\$ = gen_cmp(\$1, \$2); }
			<i>field: proto dir selector</i>

Code Generation

- Now that we have a language and a parser to translate it, how do those gen() functions actually work?
 - gen_cmp() generates code to compare a packet field to a value
 - Ex: **tcp src port 100**
 - “tcp src port” is the field
 - 100 is the value

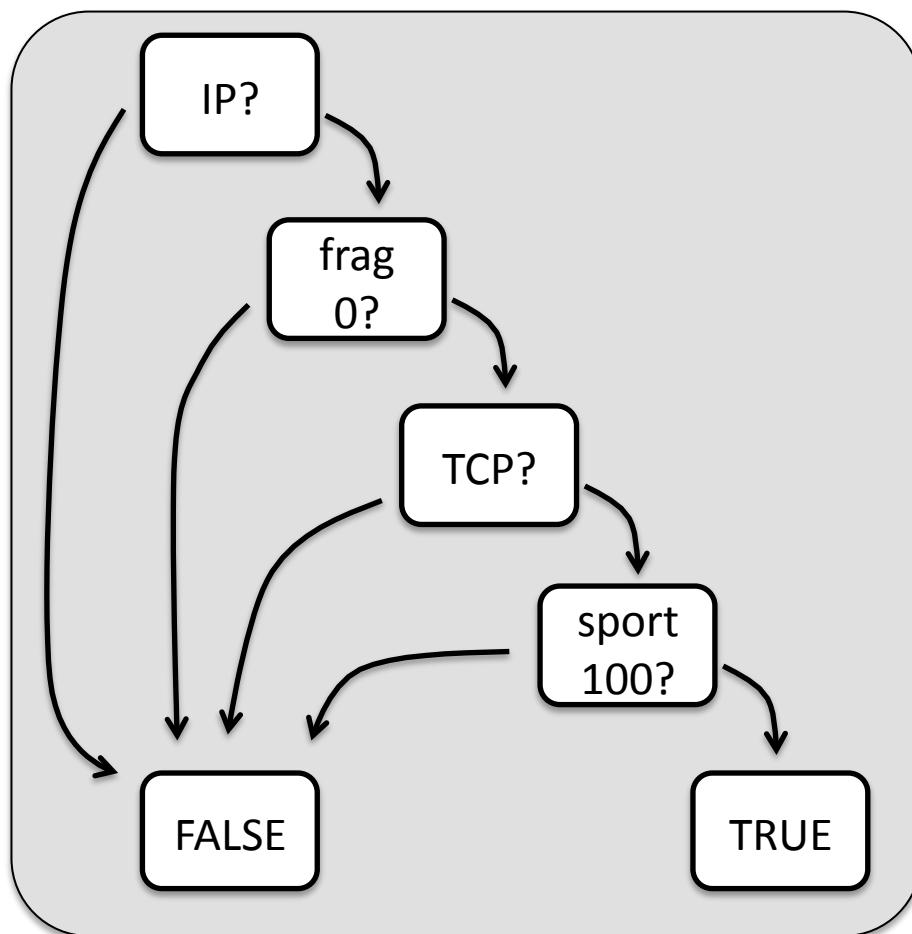
tcp src port 100



Compound Logic

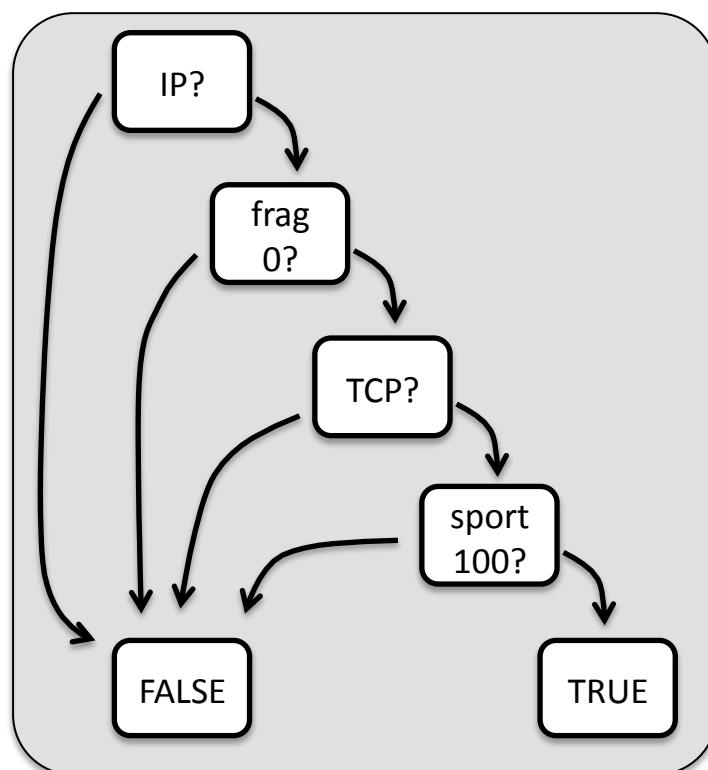
- Now, what if want traffic in either direction for port 100?
 - **tcp port 100**
 - **tcp src port 100**
 - **OR**
 - **tcp dst port 100**

tcp src port 100



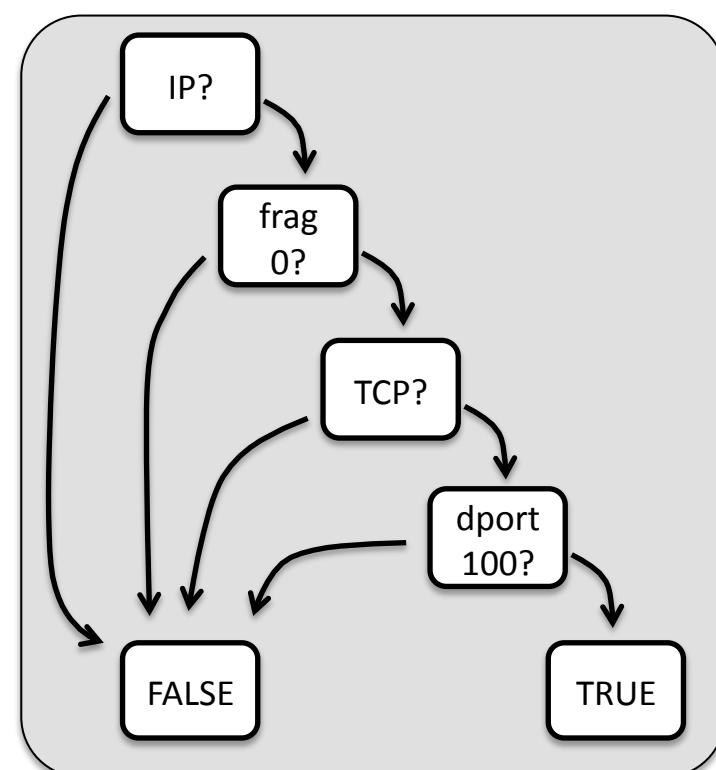
tcp port 100

tcp src port 100



tcp dst port 100

OR



FALSE

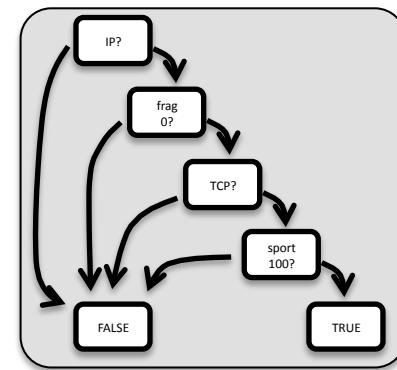
TRUE

tcp port 100

tcp src port 100

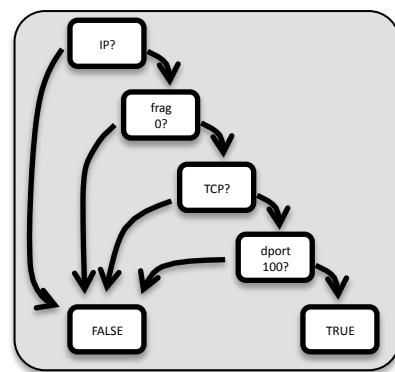
OR

tcp dst port 100



false

true



false

true

FALSE

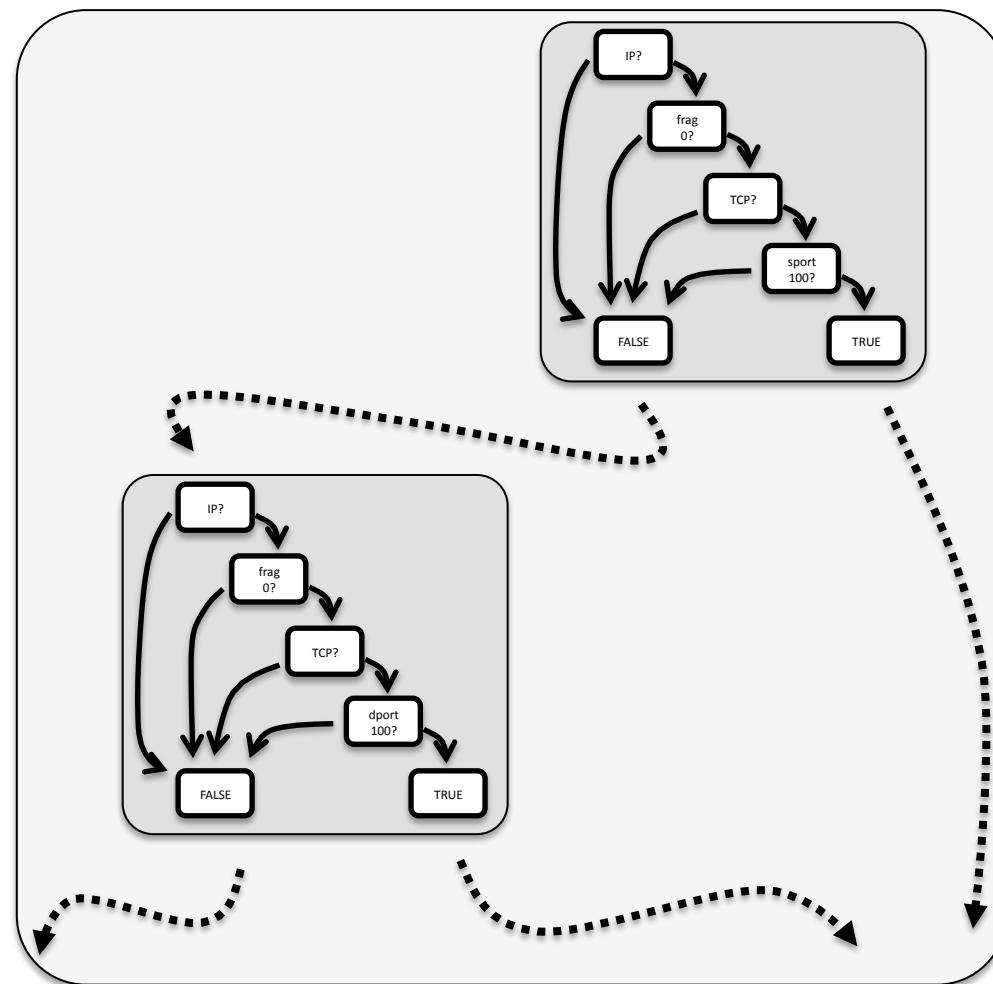
TRUE

Compound Logic

- What if I want packets *between* port 100 and some other specific port 200?
 - `tcp port 100`
 - **AND**
 - `tcp port 200`
- The output gets ever more complex

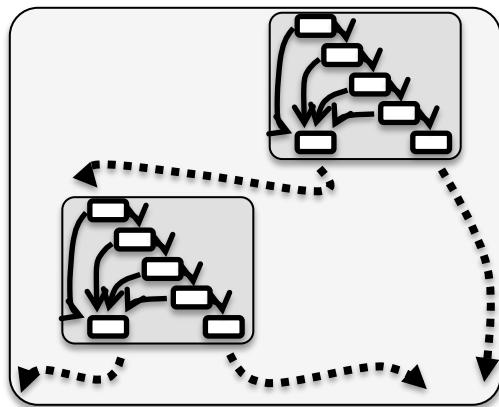
tcp port 100 and 200?

tcp port 100

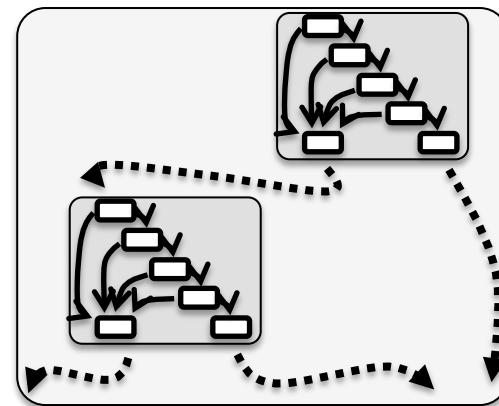


tcp port 100 and 200

tcp port 100



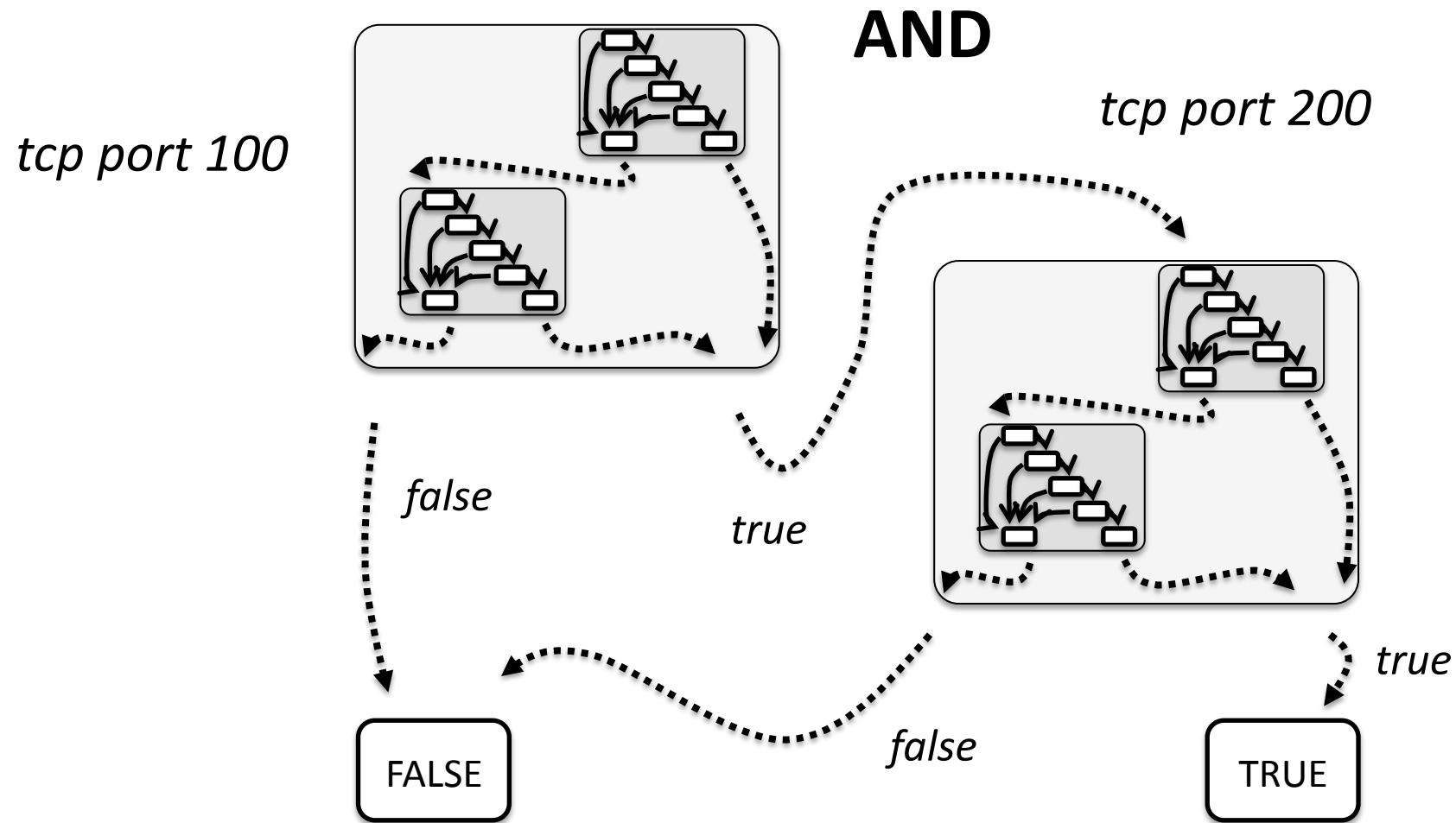
tcp port 200



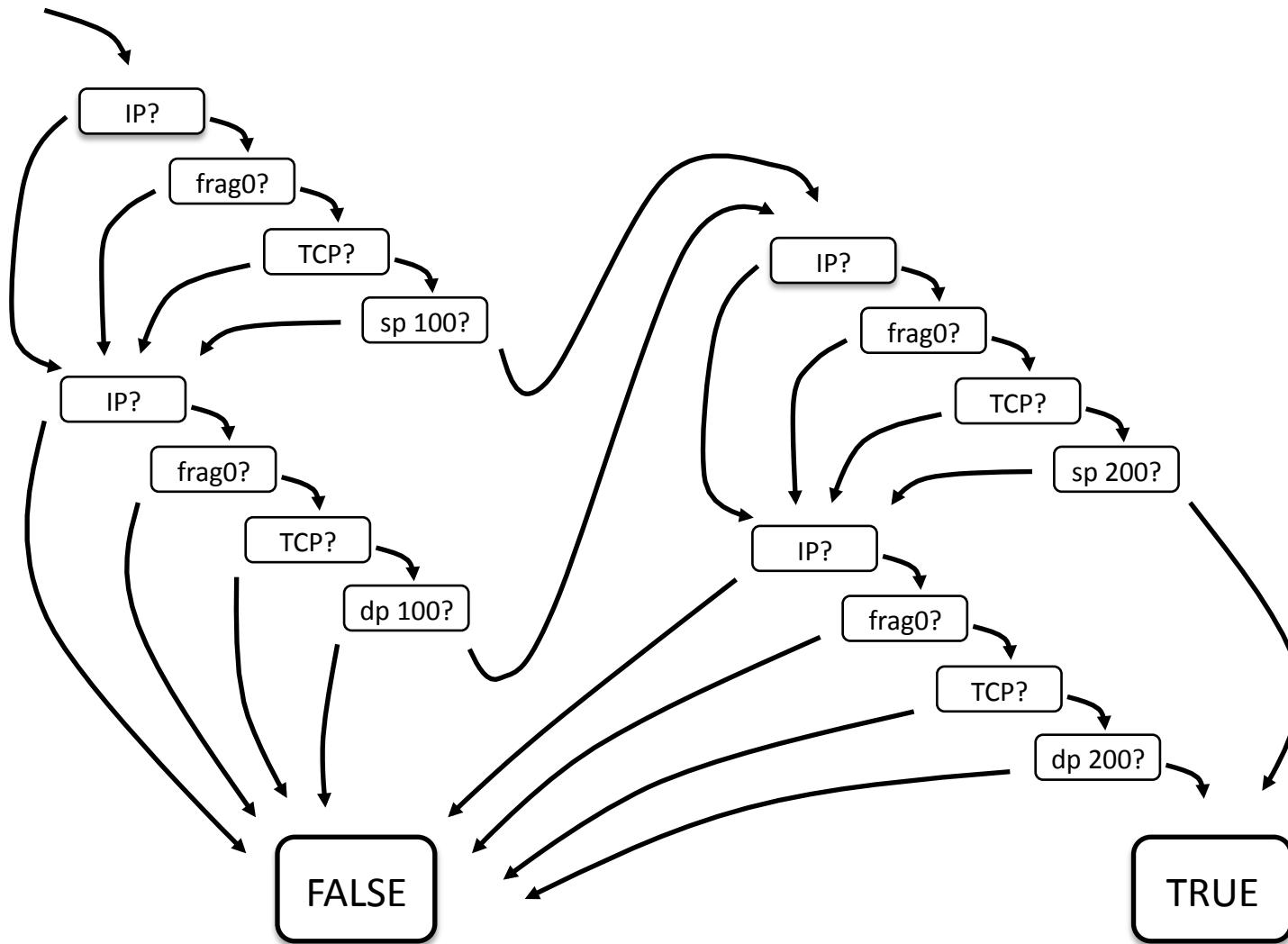
FALSE

TRUE

tcp port 100 and 200



tcp port 100 and 200

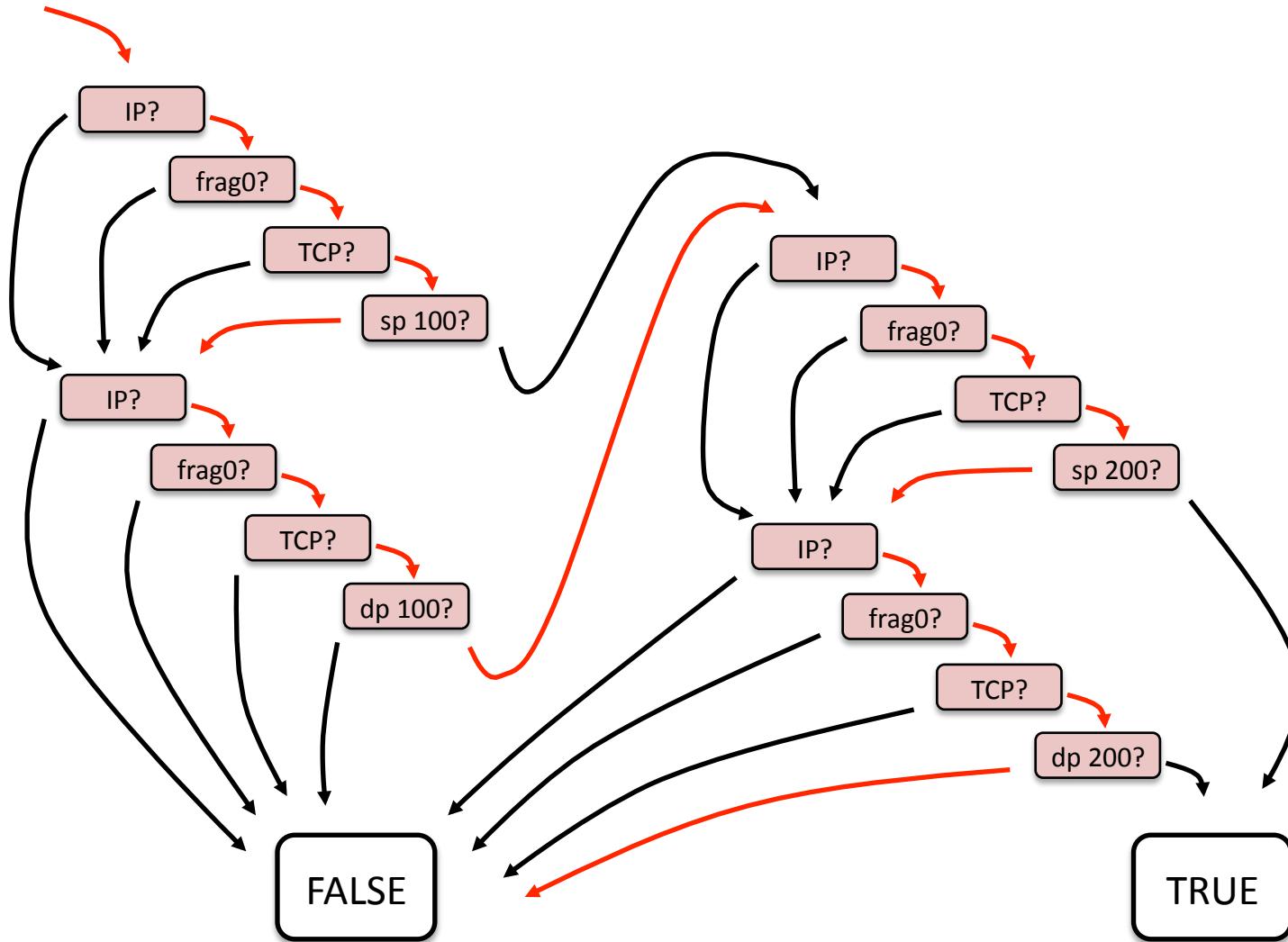


The Raw Code

```
(000) ldh [16]
(001) jeq #0x800      jt 2 jf 43
(002) ldh [16]
(003) jeq #0x86dd      jt 4 jf 10
(004) ldb [24]
(005) jeq #0x6       jt 6   jf 10
(006) ldh [58]
(007) jeq #0x64      jt 22jf 8
(008) ldh [60]
(009) jeq #0x64      jt 22jf 10
(010) ldh [16]
(011) jeq #0x800      jt 12 jf 43
(012) ldb [27]
(013) jeq #0x6       jt 14 jf 43
(014) ldh [24]
(015) jset #0x1fff      jt 43 jf 16
(016) idxb 4*([18]&0xf)
(017) ldh [x + 18]
(018) jeq #0x64      jt 22jf 19
(019) idxb 4*([18]&0xf)
(020) ldh [x + 20]
(021) jeq #0x64      jt 22jf 43
```

```
(022) ldh [16]
(023) jeq #0x86dd      jt 24 jf 30
(024) ldb [24]
(025) jeq #0x6       jt 26   jf 30
(026) ldh [58]
(027) jeq #0xc8      jt 42   jf 28
(028) ldh [60]
(029) jeq #0xc8      jt 42   jf 30
(030) ldh [16]
(031) jeq #0x800      jt 32   jf 43
(032) ldb [27]
(033) jeq #0x6       jt 34   jf 43
(034) ldh [24]
(035) jset #0x1fff      jt 43   jf 36
(036) idxb 4*([18]&0xf)
(037) ldh [x + 18]
(038) jeq #0xc8      jt 42   jf 39
(039) idxb 4*([18]&0xf)
(040) ldh [x + 20]
(041) jeq #0xc8      jt 42   jf 43
(042) ret #65535
(043) ret #0
```

Redundant and Inefficient

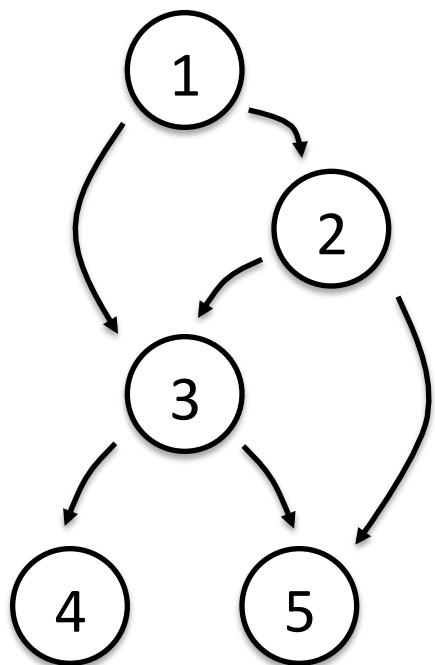


Enter Optimization

- Post-process generated code with optimization techniques
 - *libpcap/optimize.c*
- Leveraged a bunch of well known techniques from my compilers course

The Dominator Concept

- A well-known technique global data flow optimization at the time used dominators



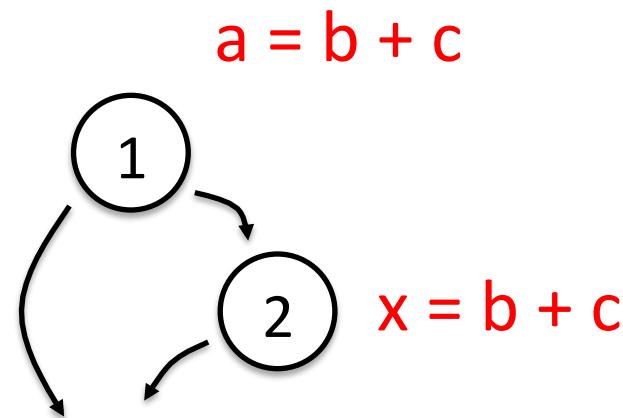
$$\text{DOM}(1) = \{ 2, 3, 4, 5 \}$$

$$\text{DOM}(2) = \{ \}$$

$$\text{DOM}(3) = \{ 4, 5 \}$$

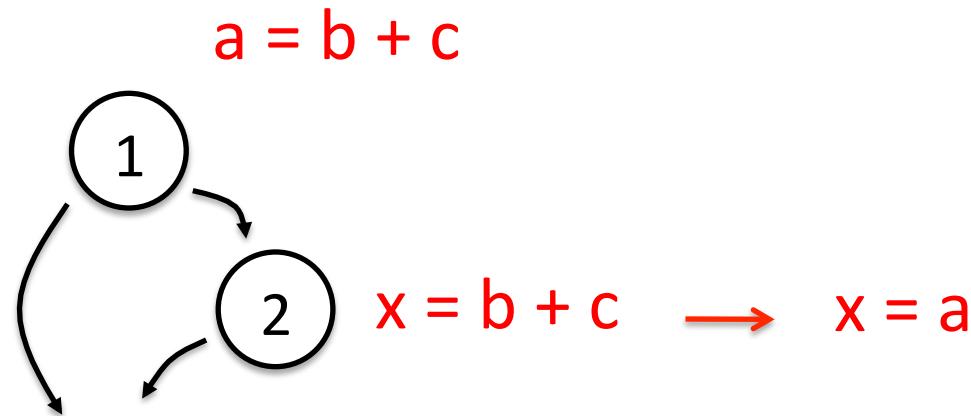
DOM Example

- Global common sub-expression elimination



DOM Example

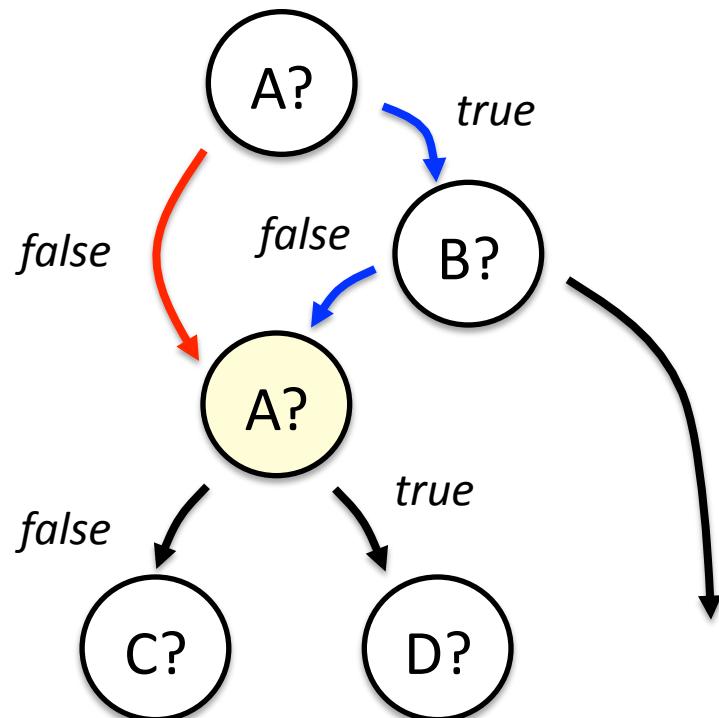
- Global common sub-expression elimination



- 2 in $\text{DOM}(1) \Rightarrow$ variables on entry to 2 same as on exit to 1, so we can replace $b + c$ with a

Not Quite Enough...

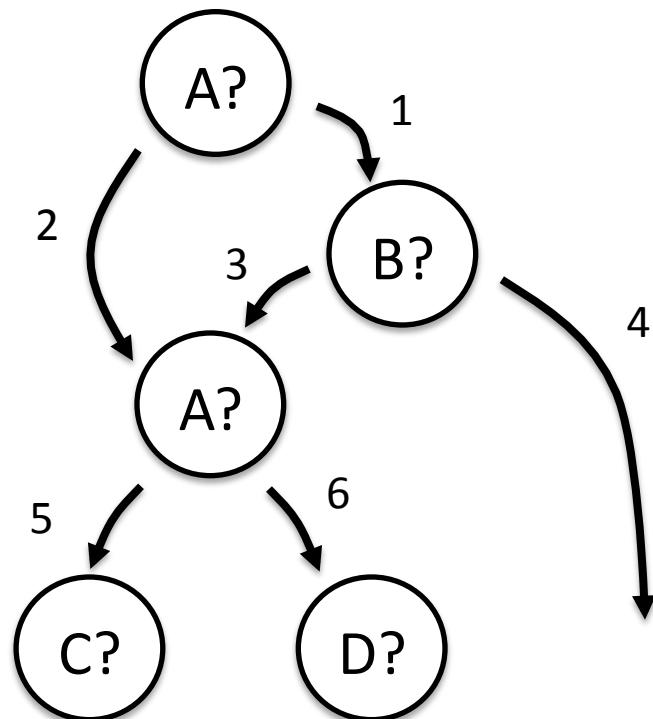
- While these traditional techniques are used by the BPF optimizer, they weren't enough...



Knowing the top node dominates yellow node doesn't let us eliminate the redundant test for A at the yellow node because either the red or blue path could happen

Edge Dominators

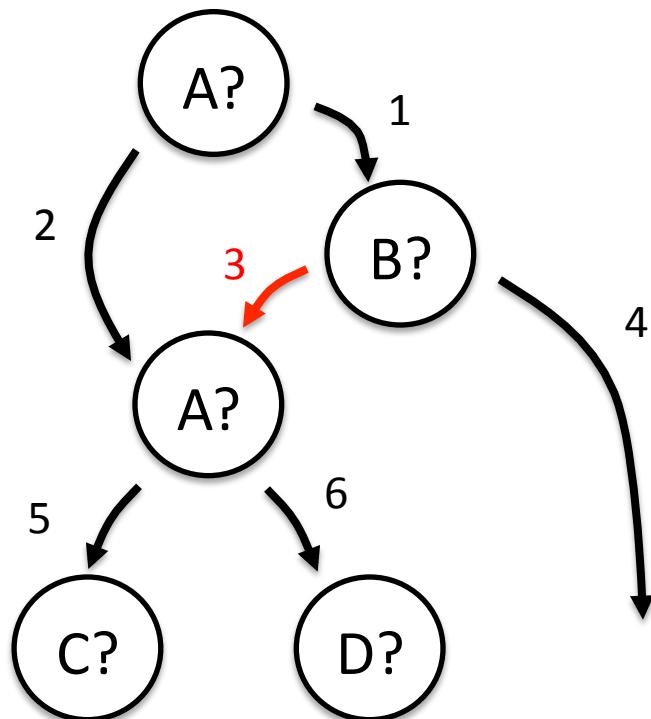
- But if we look at edge relationships instead of node relationships, we can solve the problem



$$\begin{aligned} \text{EDOM}(1) &= \{ 3, 4 \} \\ \text{EDOM}(2) &= \{ \} \\ \text{EDOM}(3) &= \{ \} \end{aligned}$$

Edge Optimization

- With this knowledge, we can safely move edges to optimize the code...



$$\text{EDOM}(1) = \{ 3, 4 \}$$

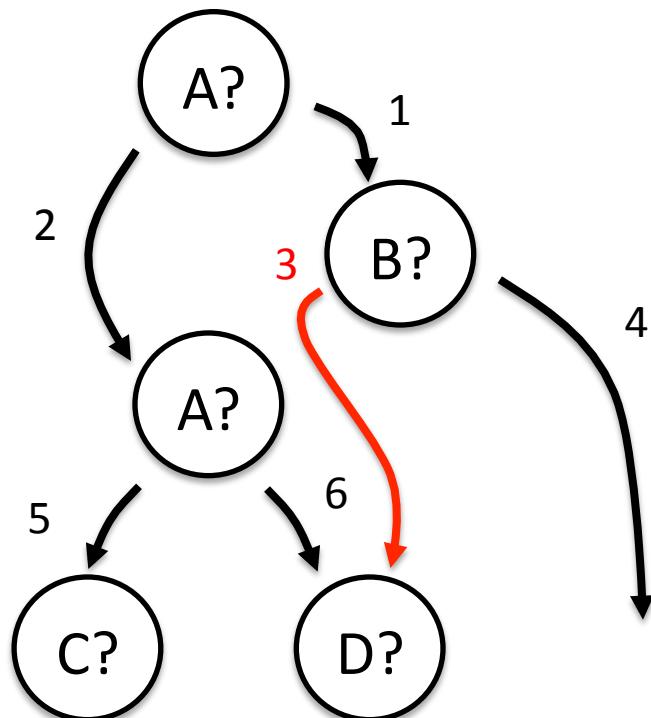
$$\text{EDOM}(2) = \{ \}$$

$$\text{EDOM}(3) = \{ \}$$

3 in $\text{EDOM}(1) \Rightarrow$
know A is true at 3 \Rightarrow
we can move 3 past
second check

Edge Optimization

- With this knowledge, we can safely move edges to optimize the code...



$$\text{EDOM}(1) = \{ 3, 4 \}$$

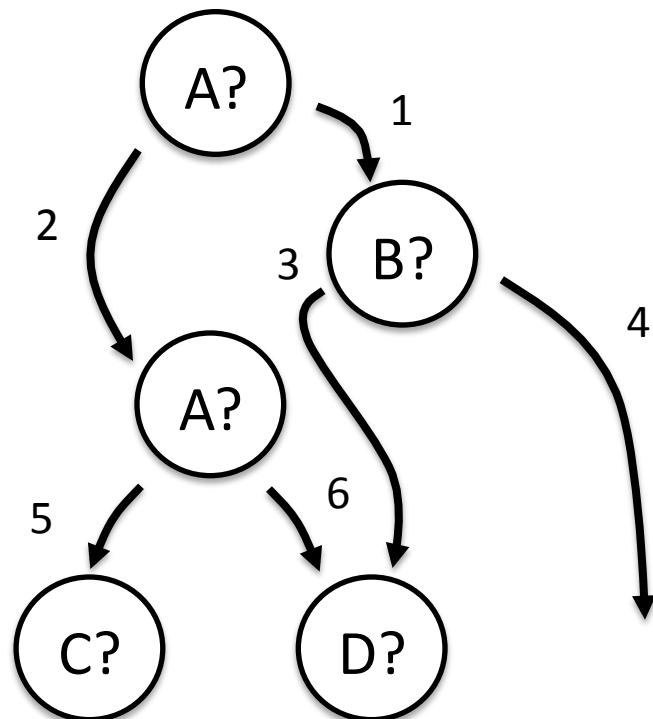
$$\text{EDOM}(2) = \{ \}$$

$$\text{EDOM}(3) = \{ \}$$

3 in $\text{EDOM}(1) \Rightarrow$
know A is true at 3 \Rightarrow
we can move 3 past
second check

Edge Optimization

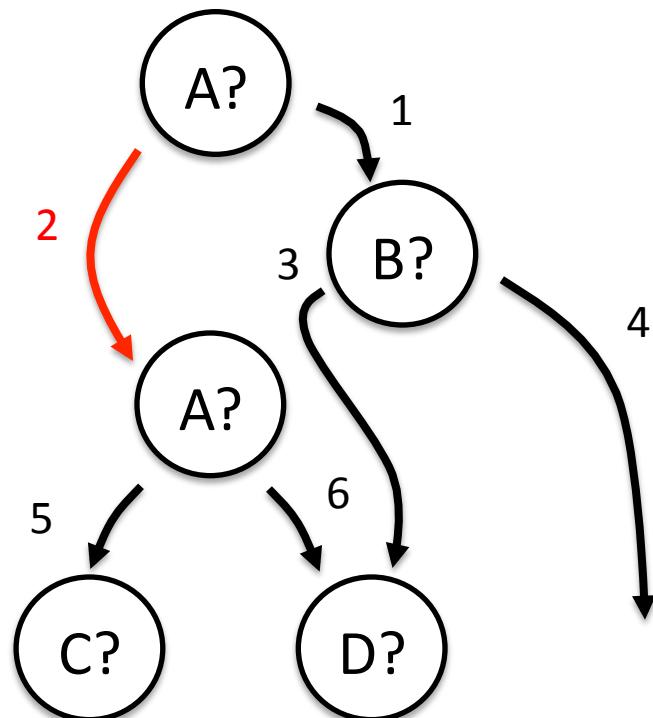
- Movements create new opportunities.
Update EDOM and repeat...



$\text{EDOM}(1) = \{ 3, 4 \}$
 $\text{EDOM}(2) = \{ \}$
 $\text{EDOM}(3) = \{ \}$

Edge Optimization

- Movements create new opportunities.
Update EDOM and repeat...



$$\text{EDOM}(1) = \{ 3, 4 \}$$

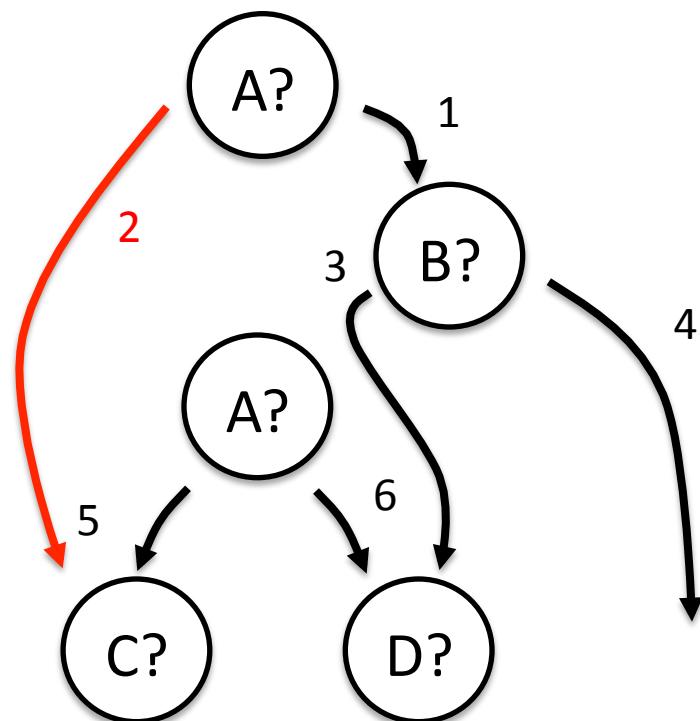
$$\text{EDOM}(2) = \{ 5, 6 \}$$

$$\text{EDOM}(3) = \{ \}$$

5 in $\text{EDOM}(2)$ =>
know A is false at 5 =>
we can move 2

Edge Optimization

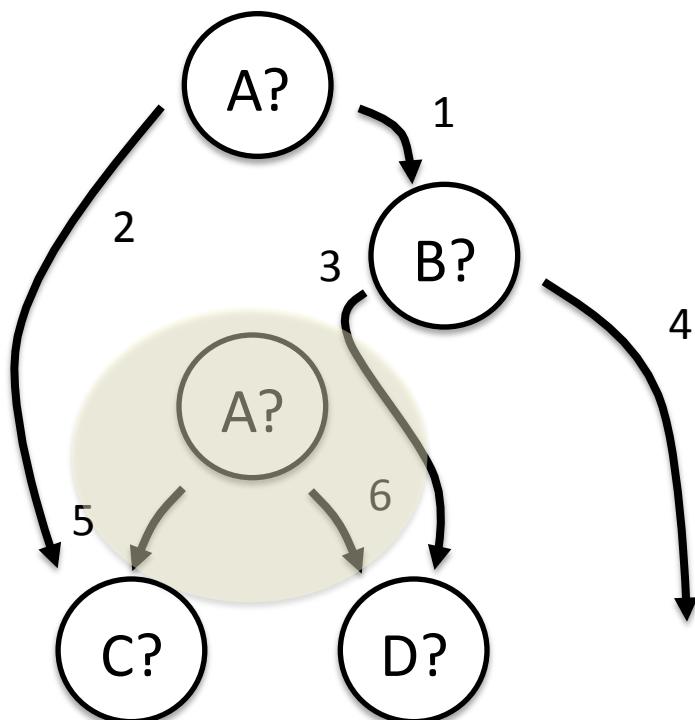
- Movements create new opportunities.
Update EDOM and repeat...



$$\begin{aligned} \text{EDOM}(1) &= \{ 3, 4 \} \\ \text{EDOM}(2) &= \{ 5, 6 \} \\ \text{EDOM}(3) &= \{ \} \end{aligned}$$

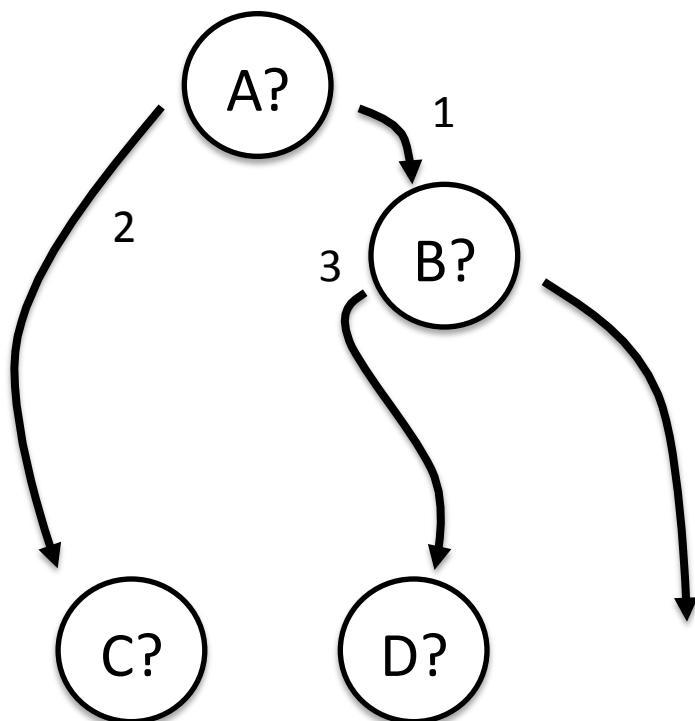
Edge Optimization

- Now we can delete unreachable code....



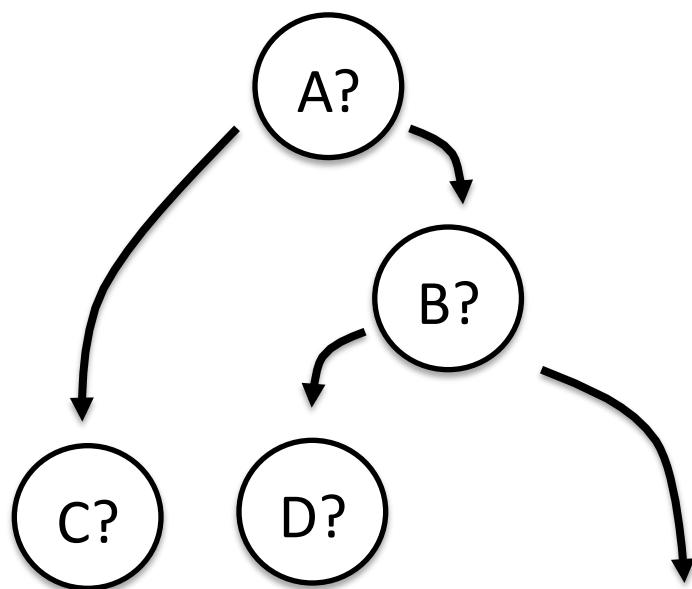
Edge Optimization

- Now we can delete unreachable code....

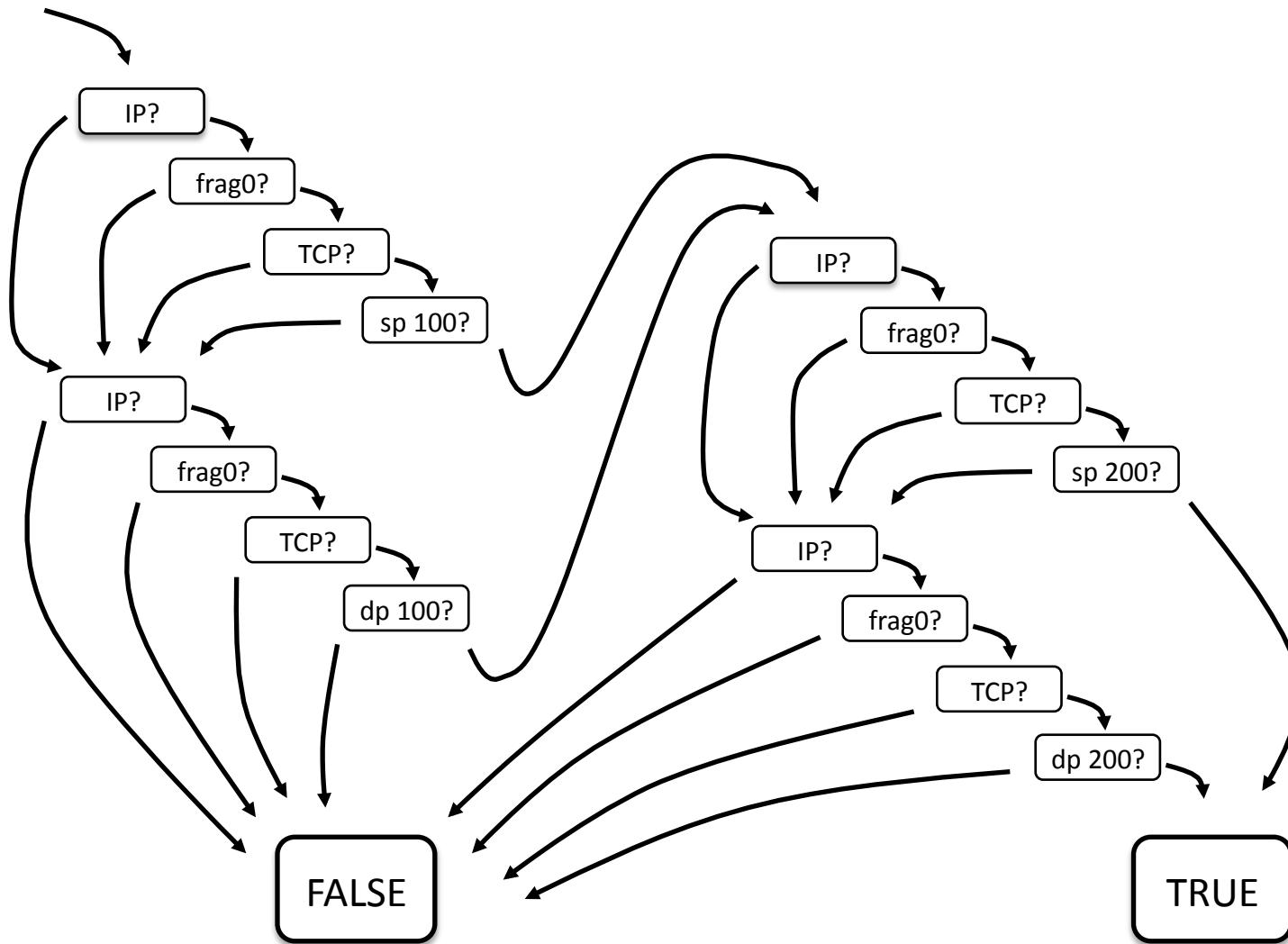


Edge Optimization

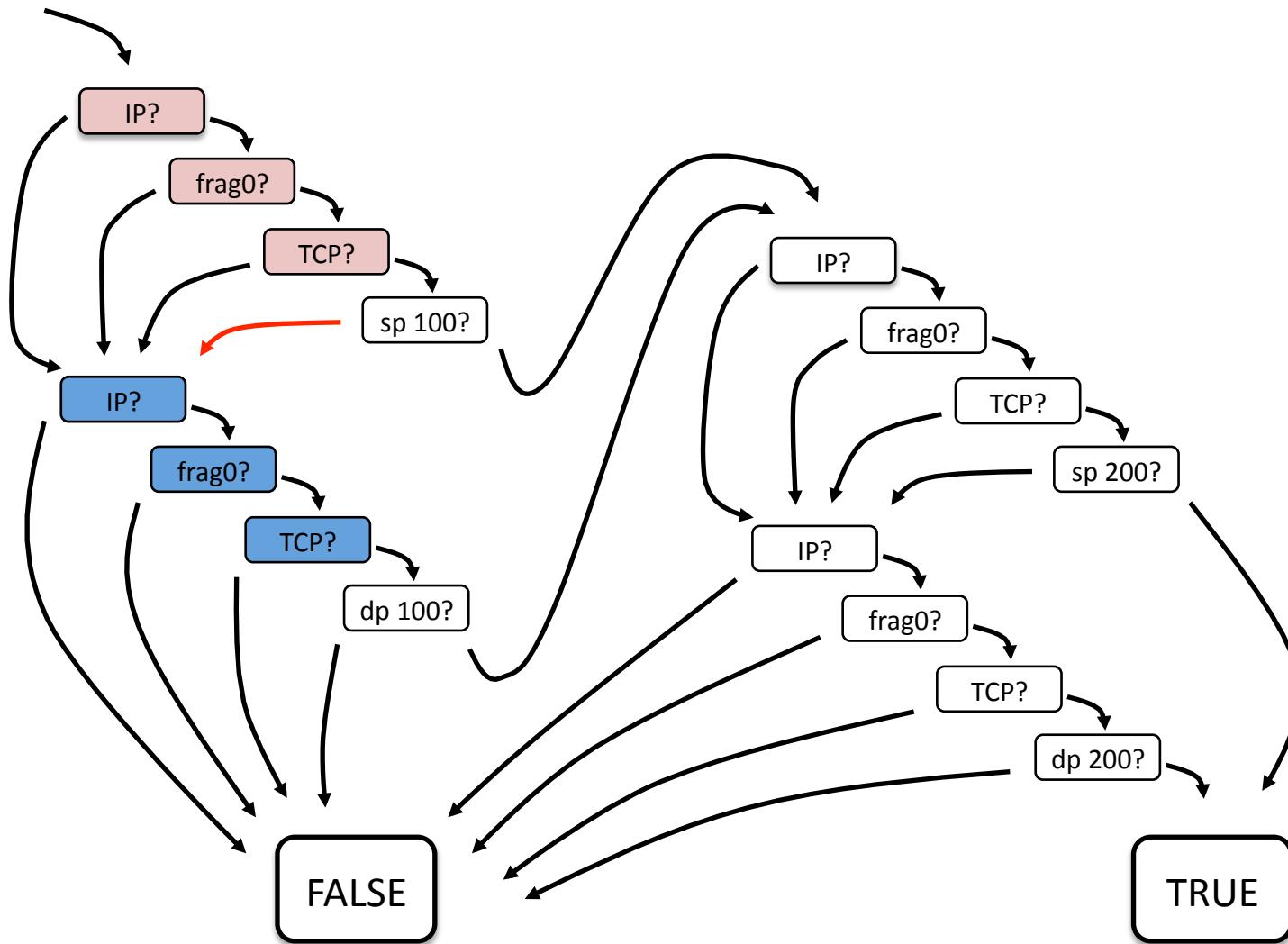
- and simplify the graph...



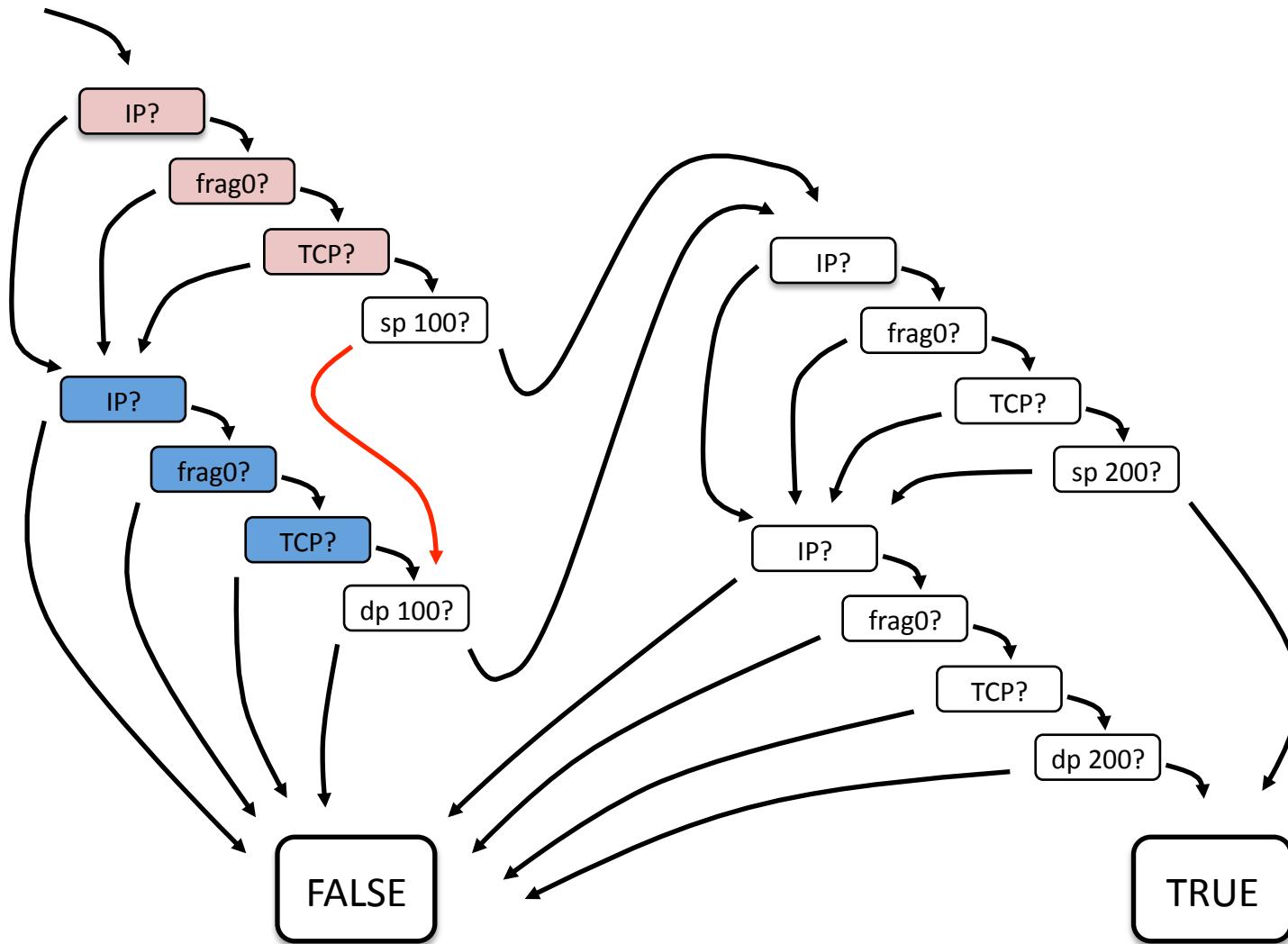
tcp port 100 and 200



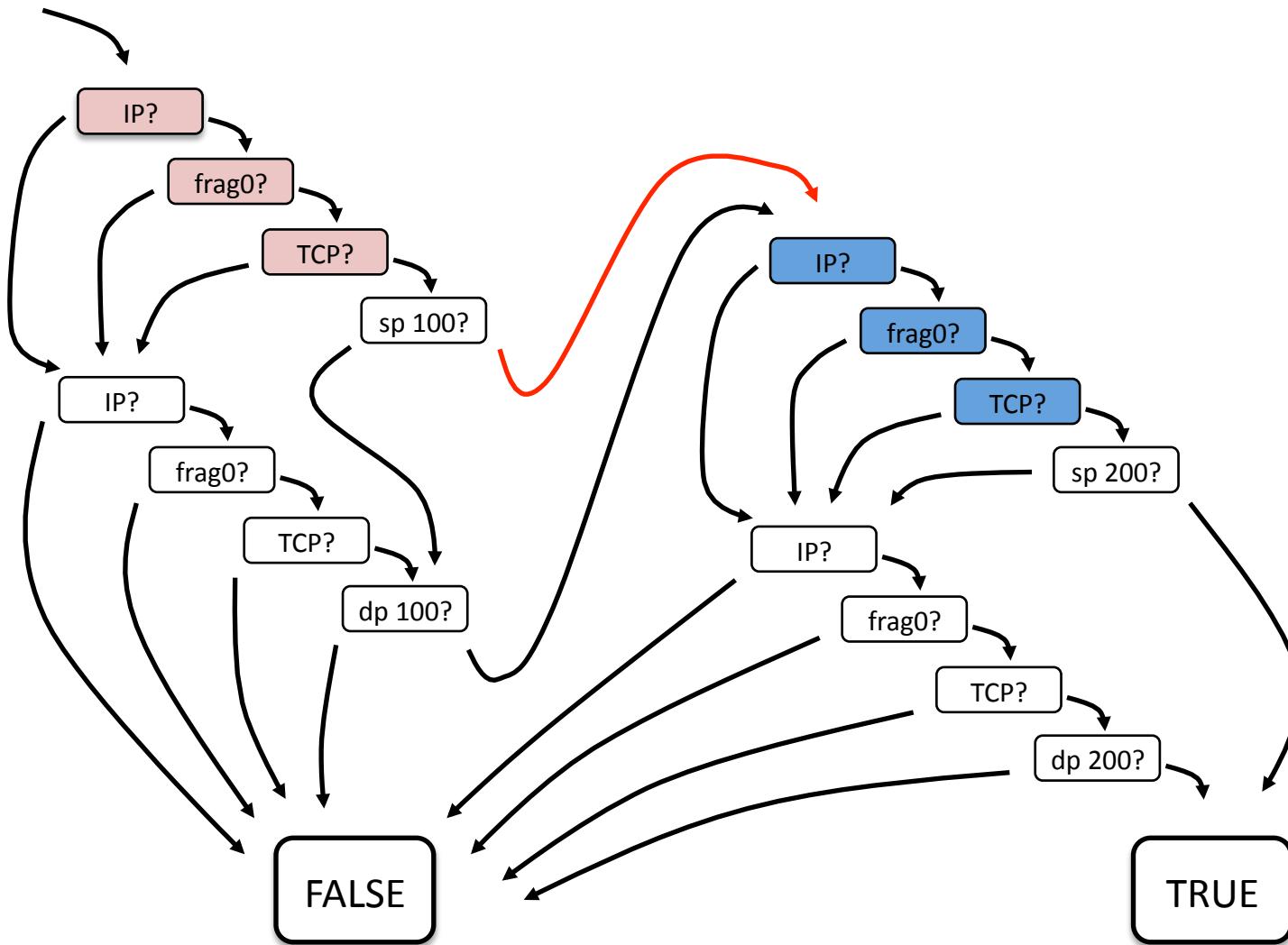
tcp port 100 and 200



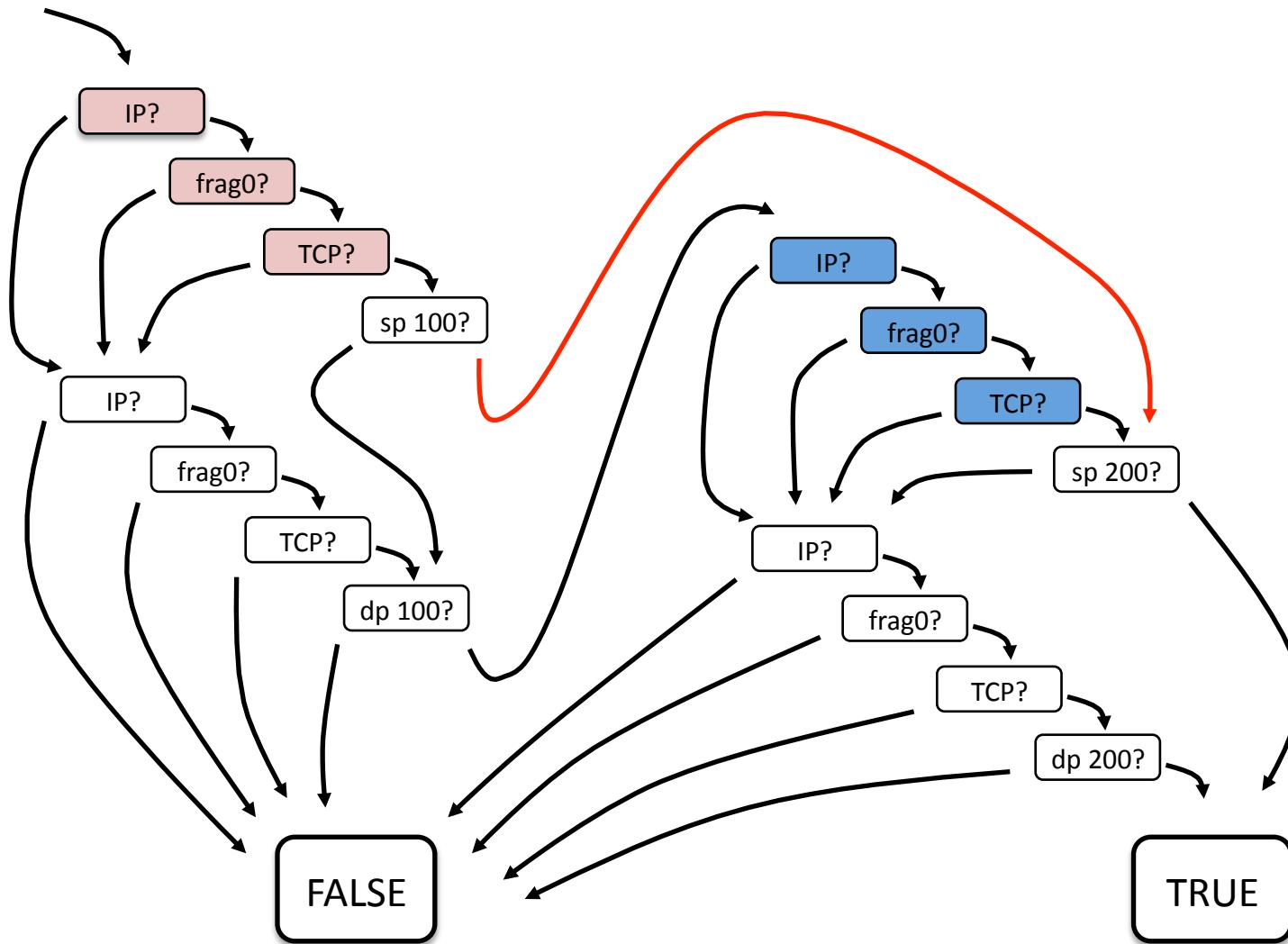
tcp port 100 and 200



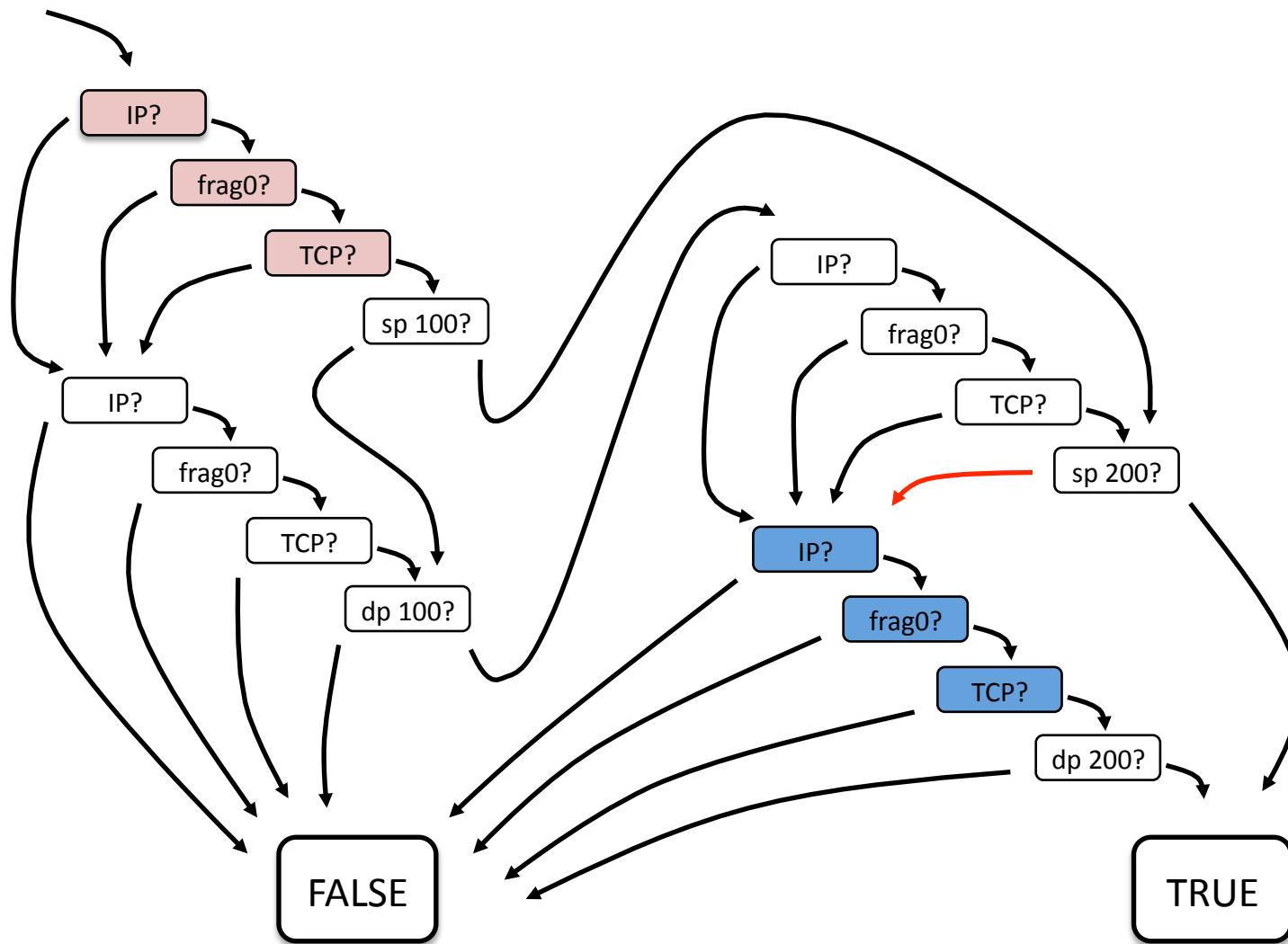
tcp port 100 and 200



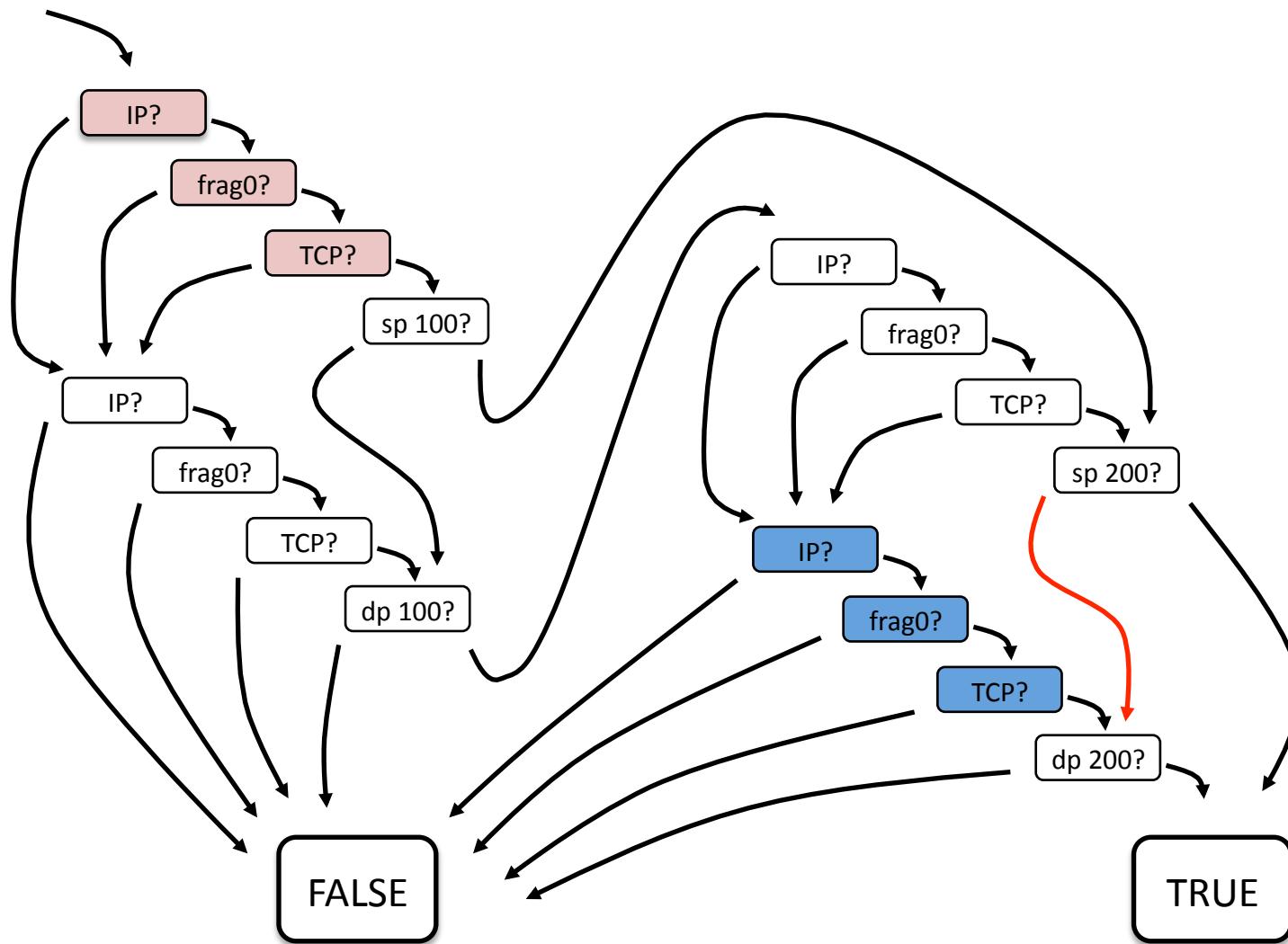
tcp port 100 and 200



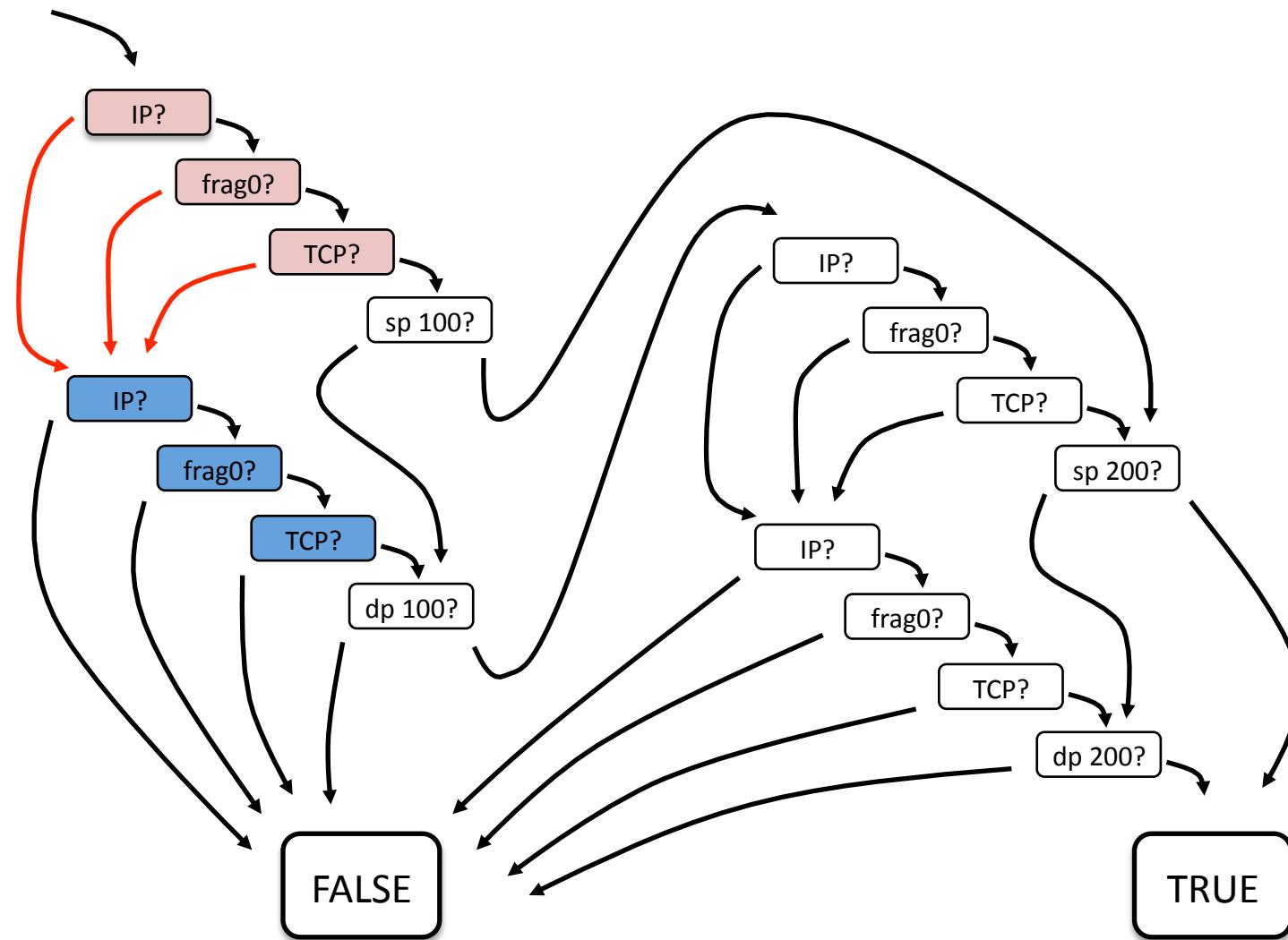
tcp port 100 and 200



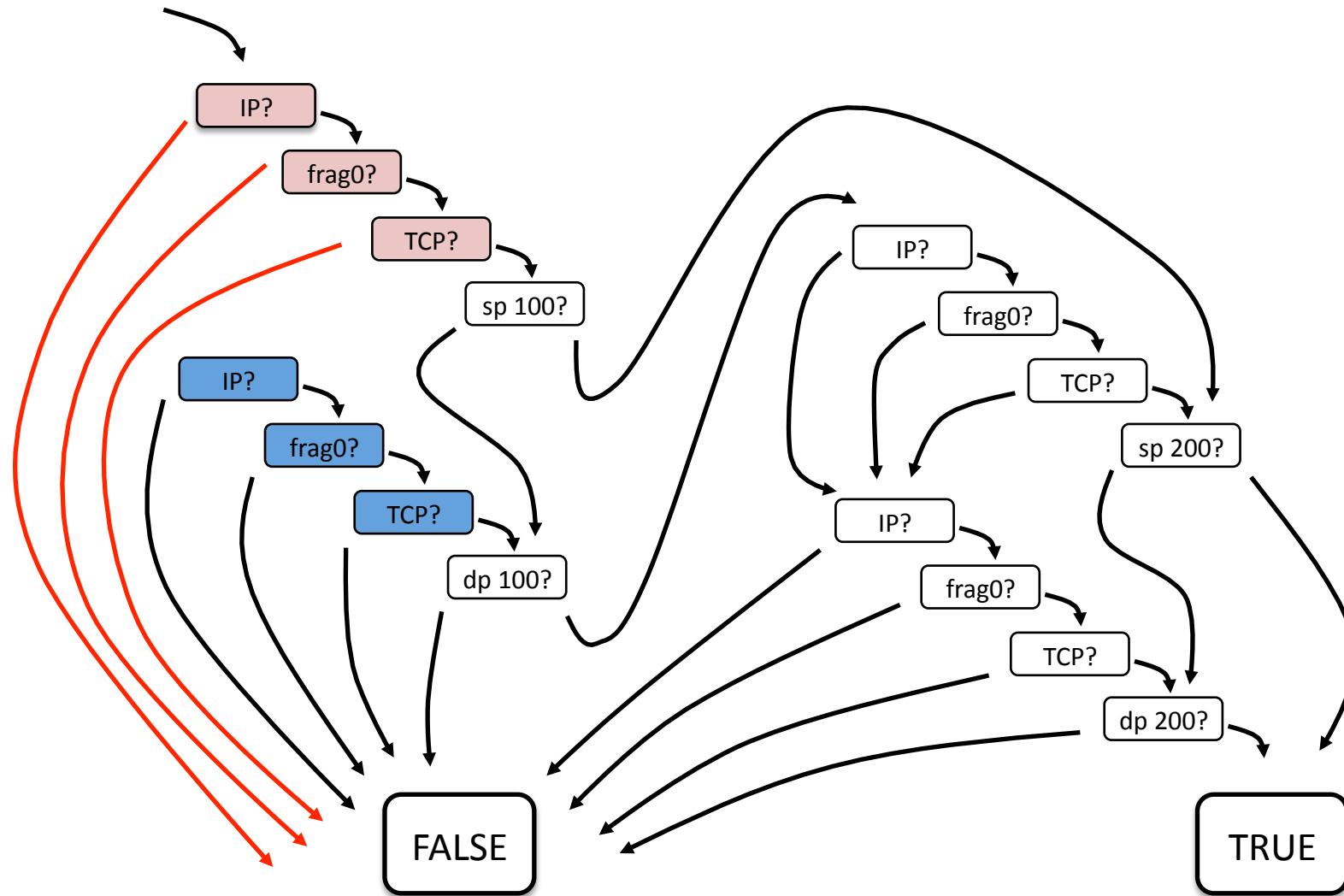
tcp port 100 and 200



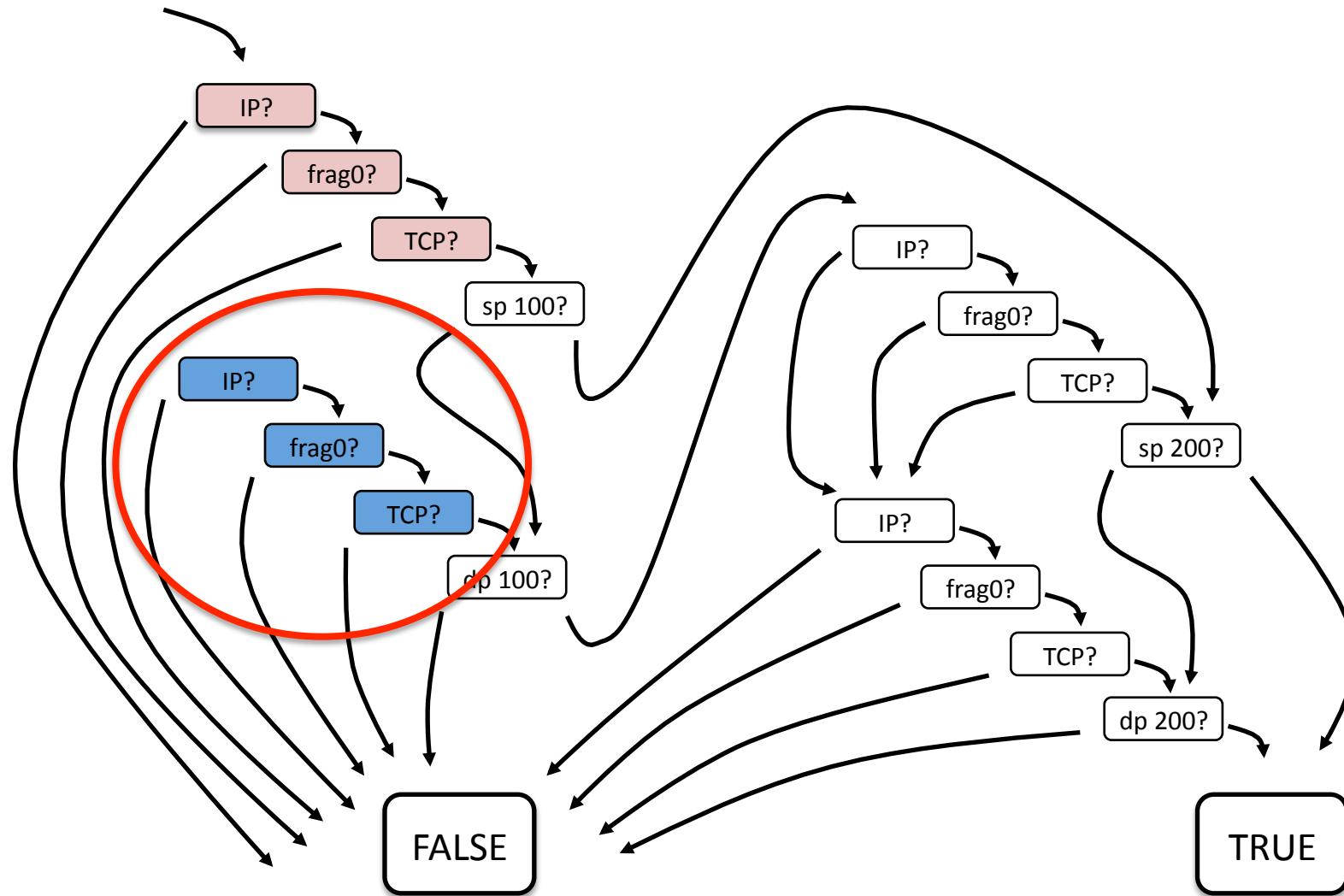
tcp port 100 and 200



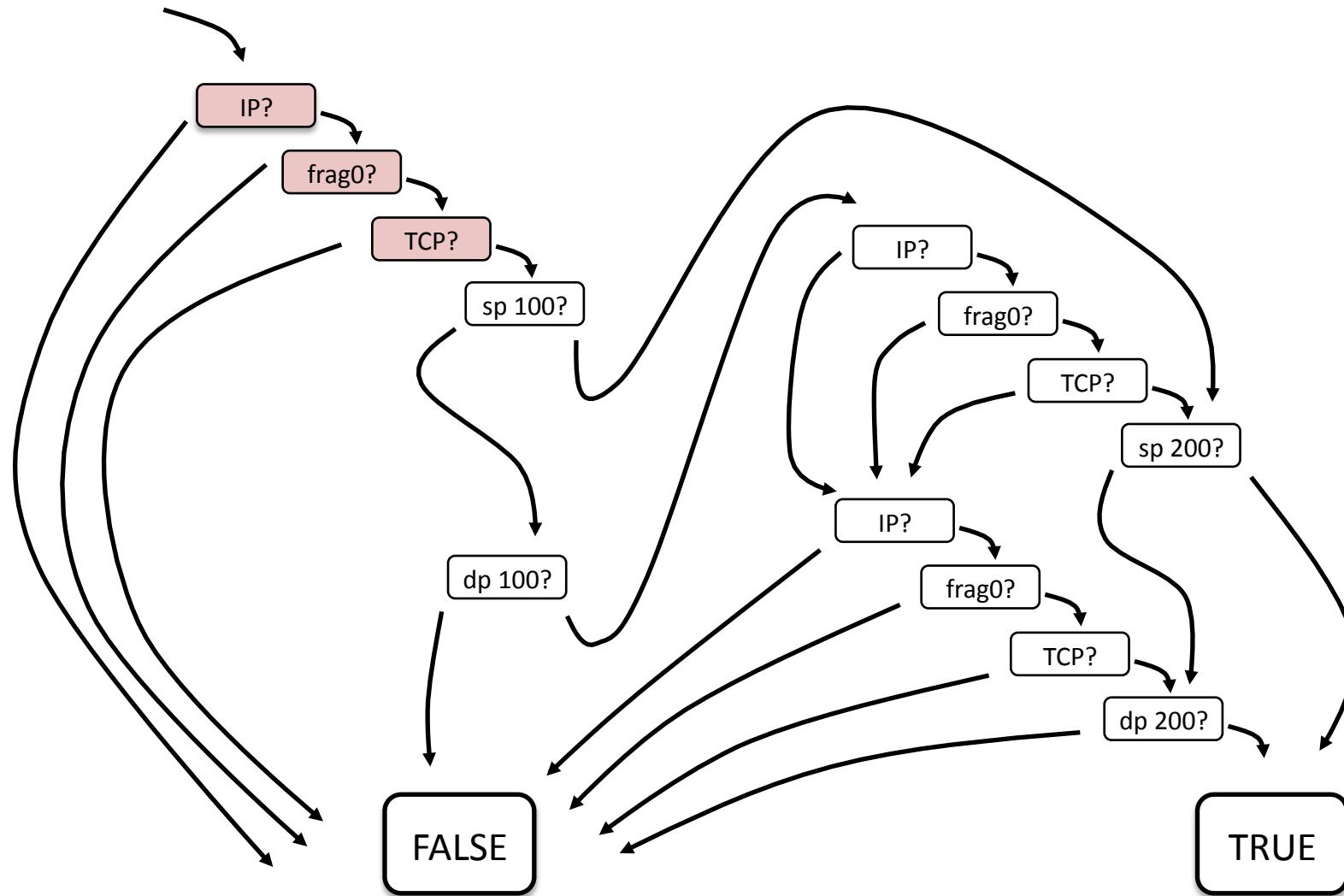
tcp port 100 and 200



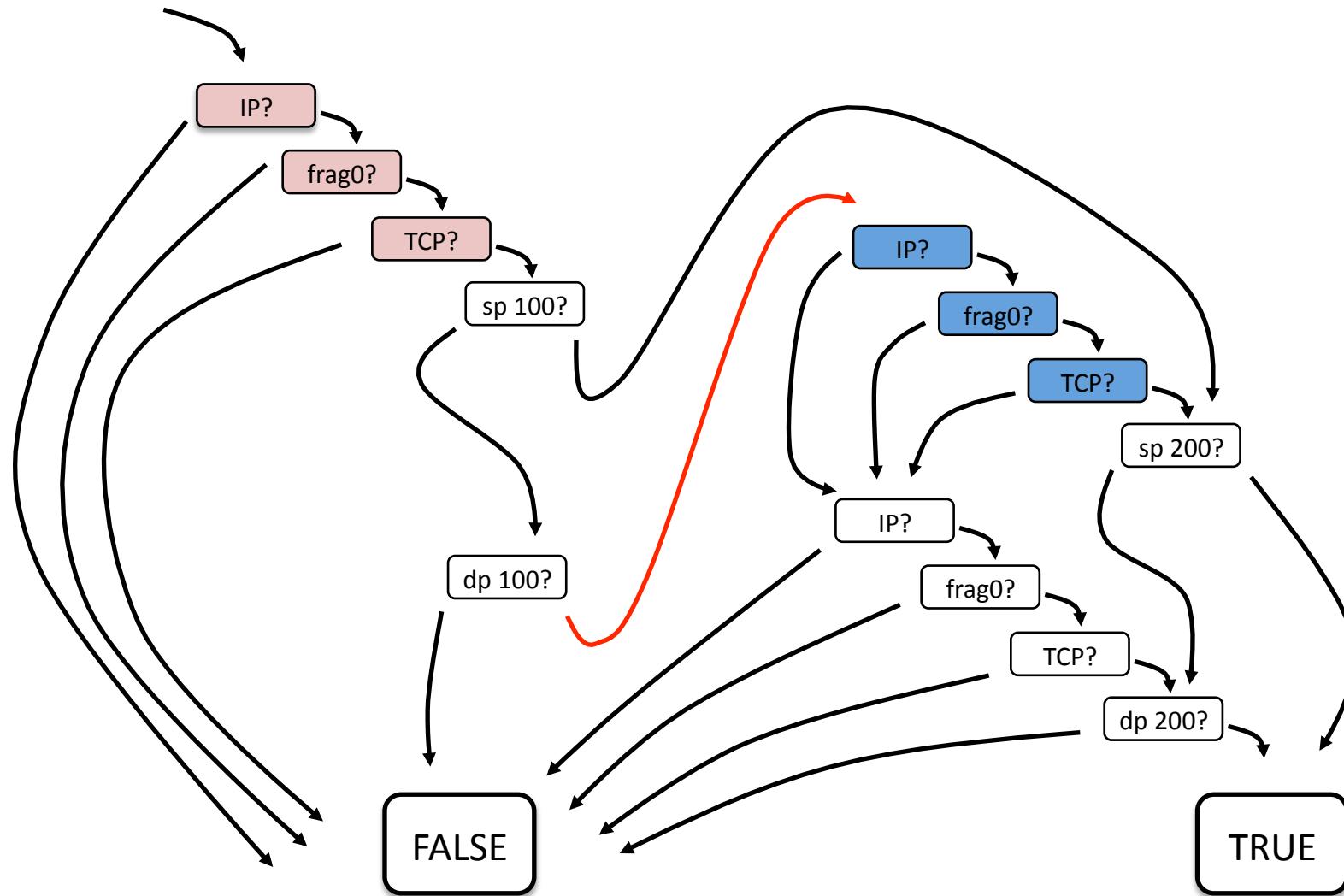
tcp port 100 and 200



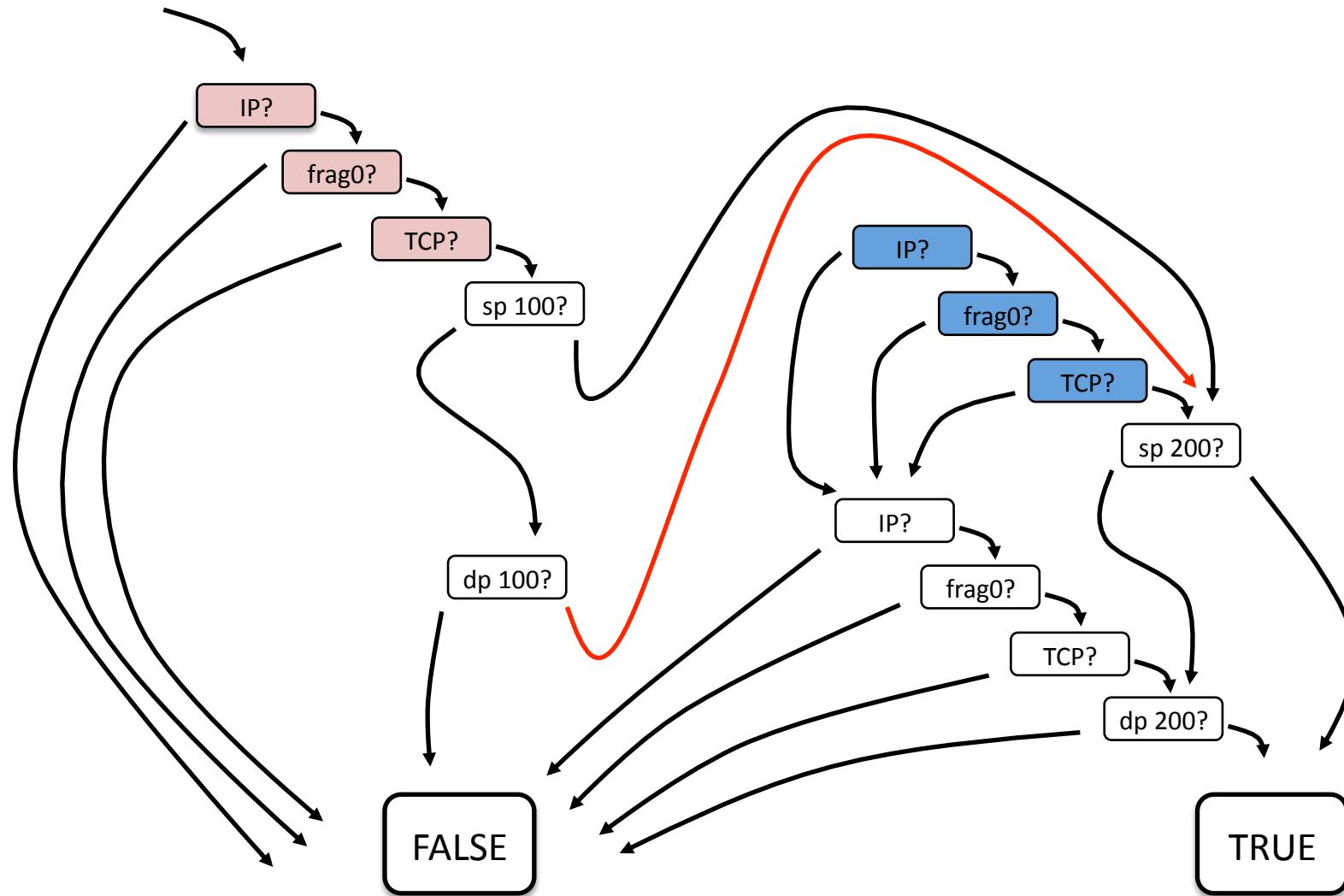
tcp port 100 and 200



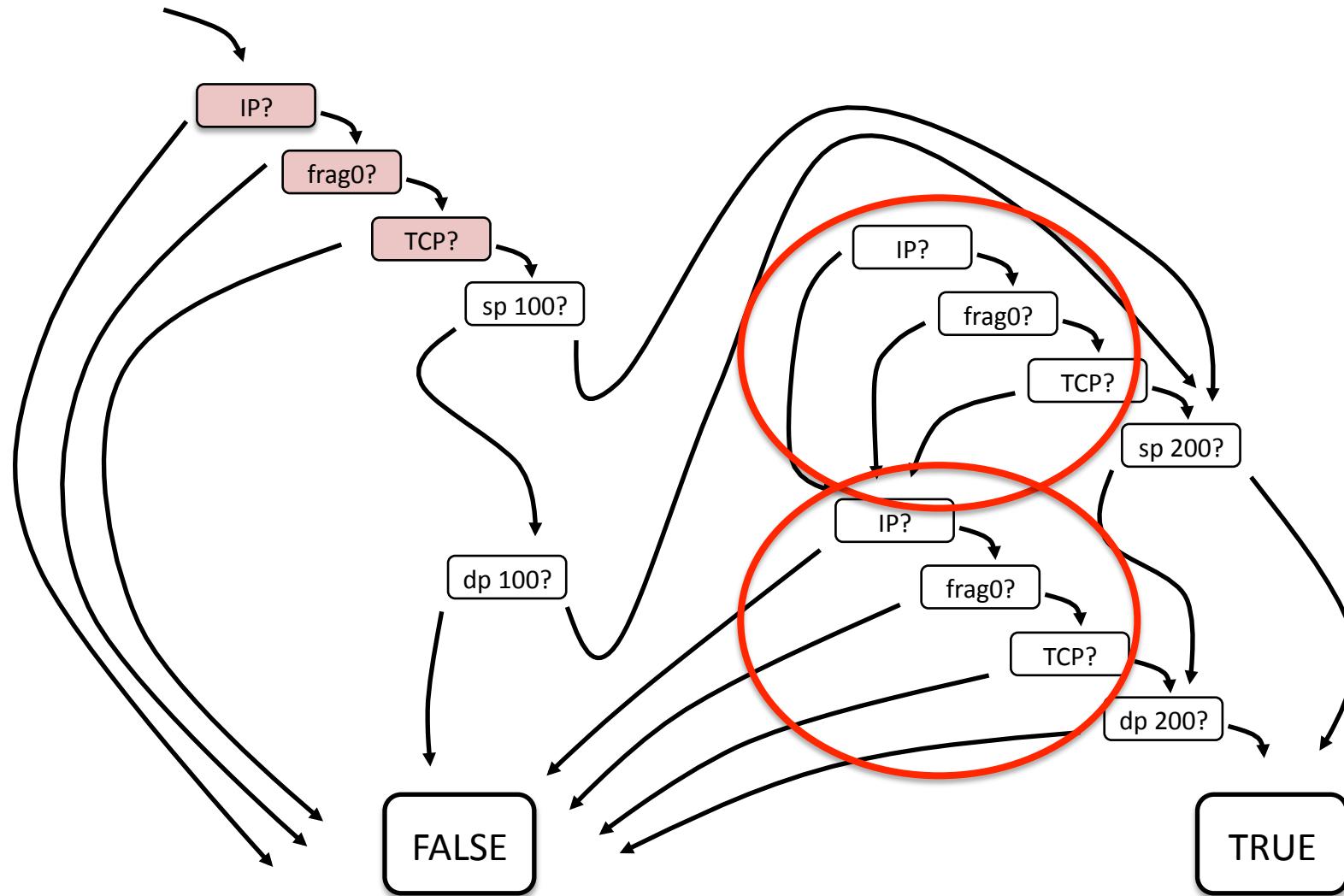
tcp port 100 and 200



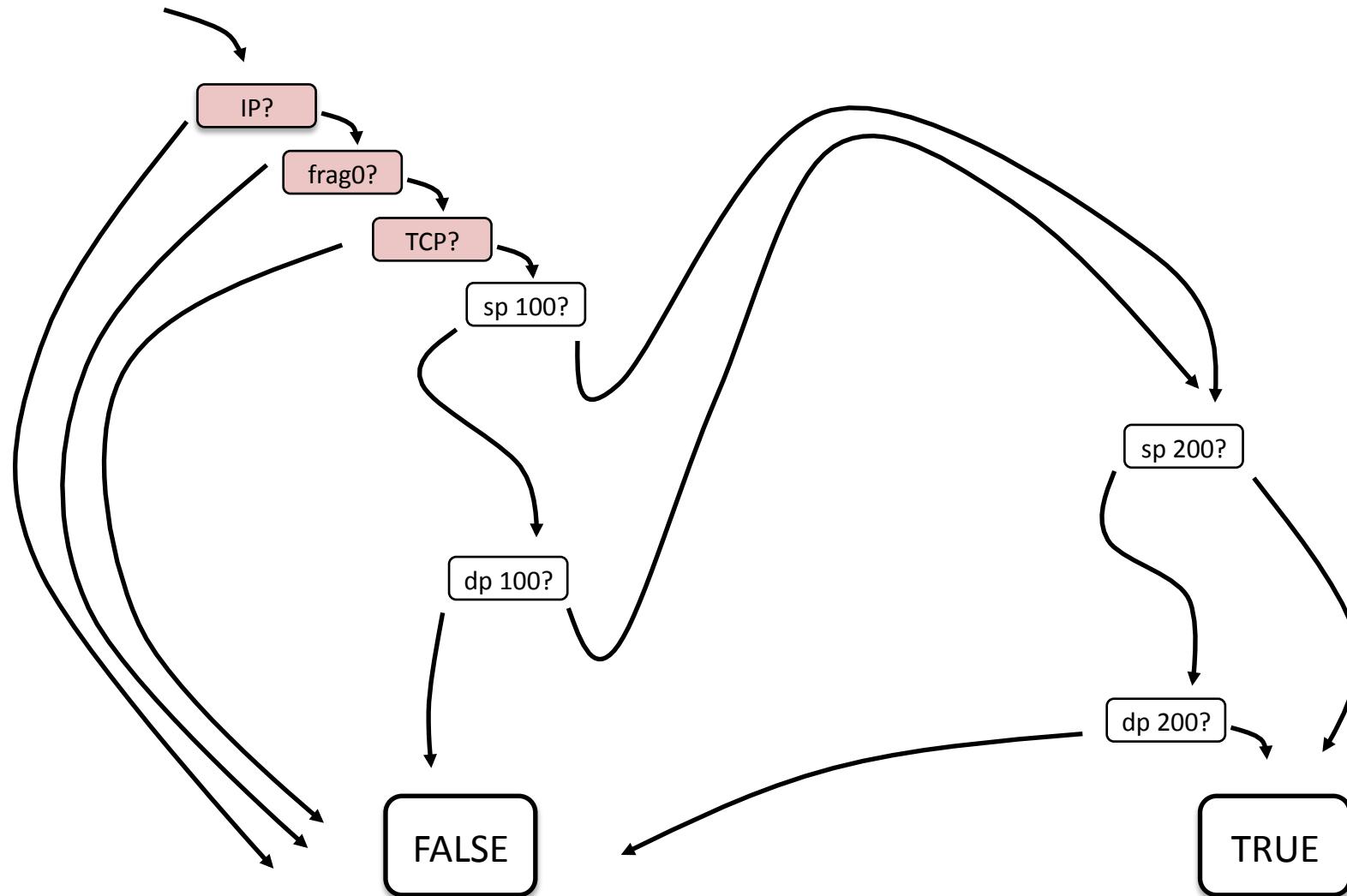
tcp port 100 and 200



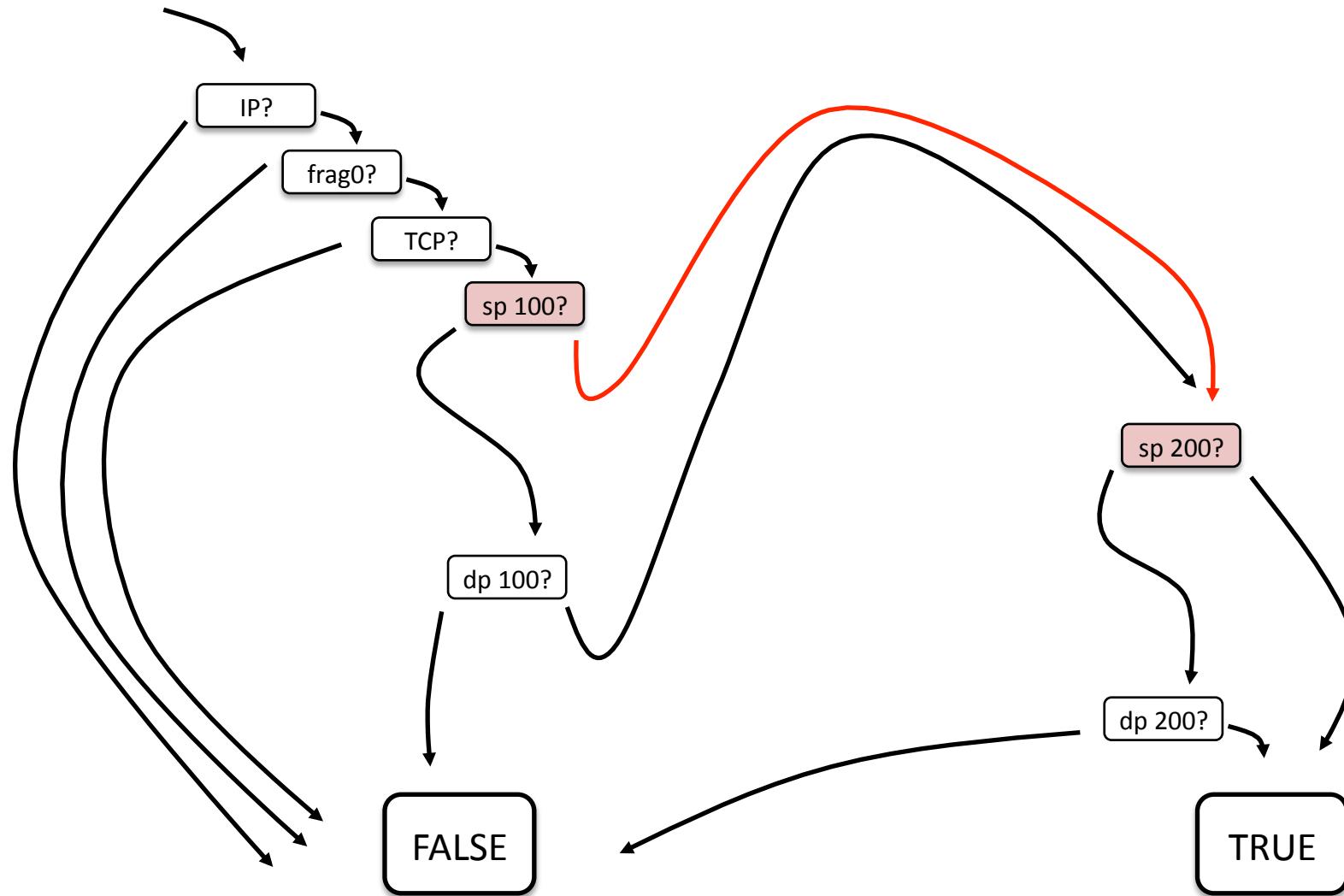
tcp port 100 and 200



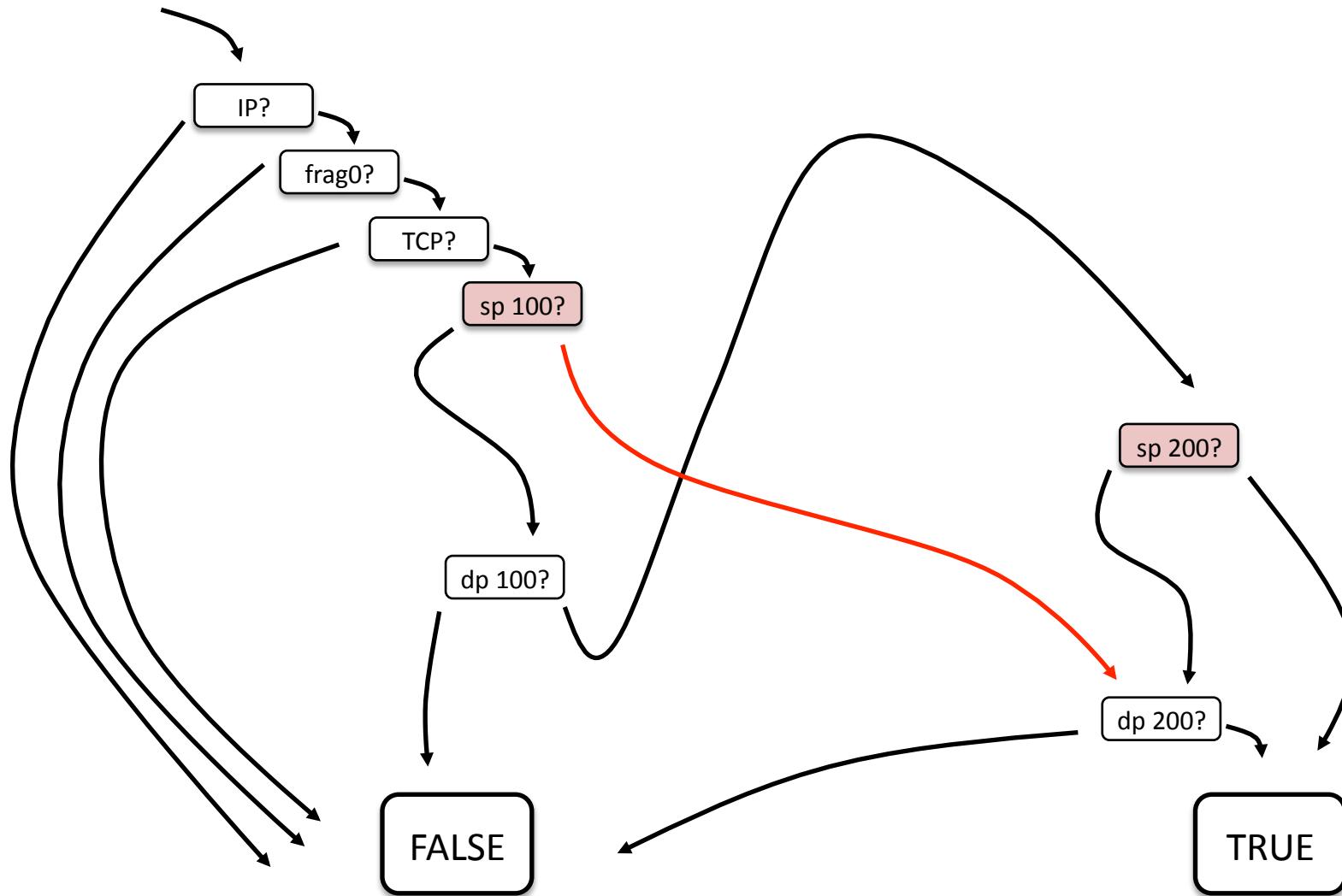
tcp port 100 and 200



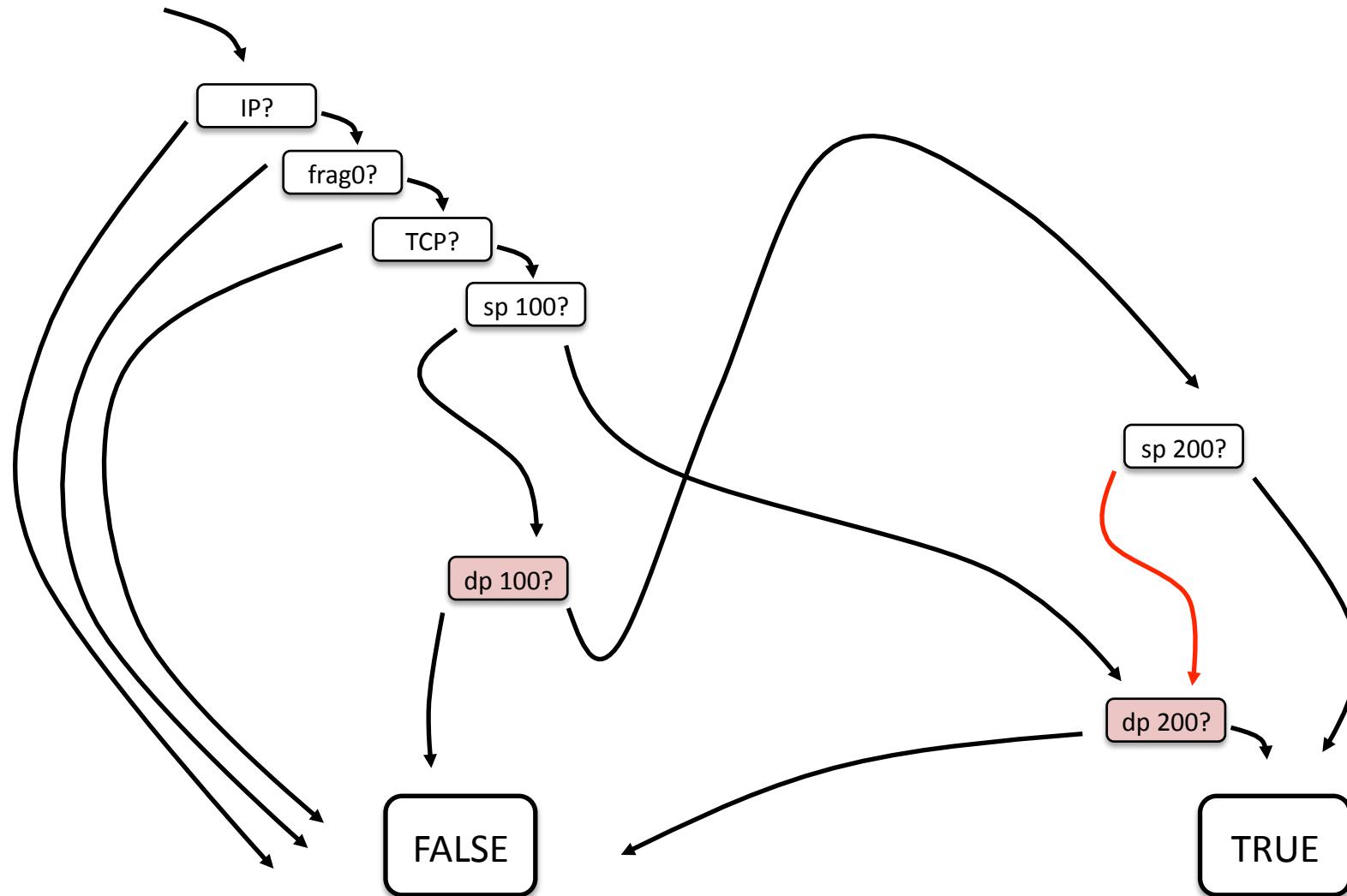
tcp port 100 and 200



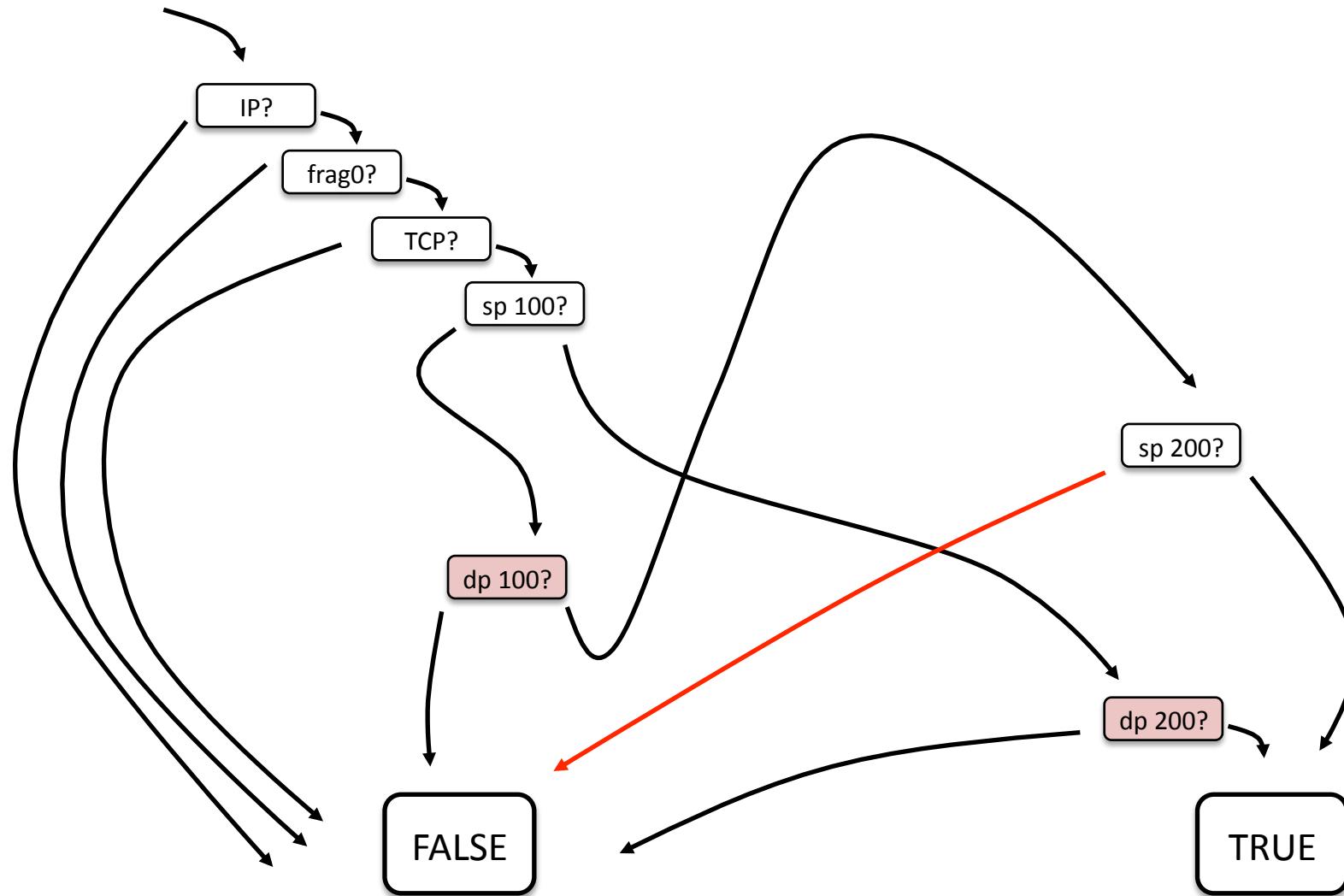
tcp port 100 and 200



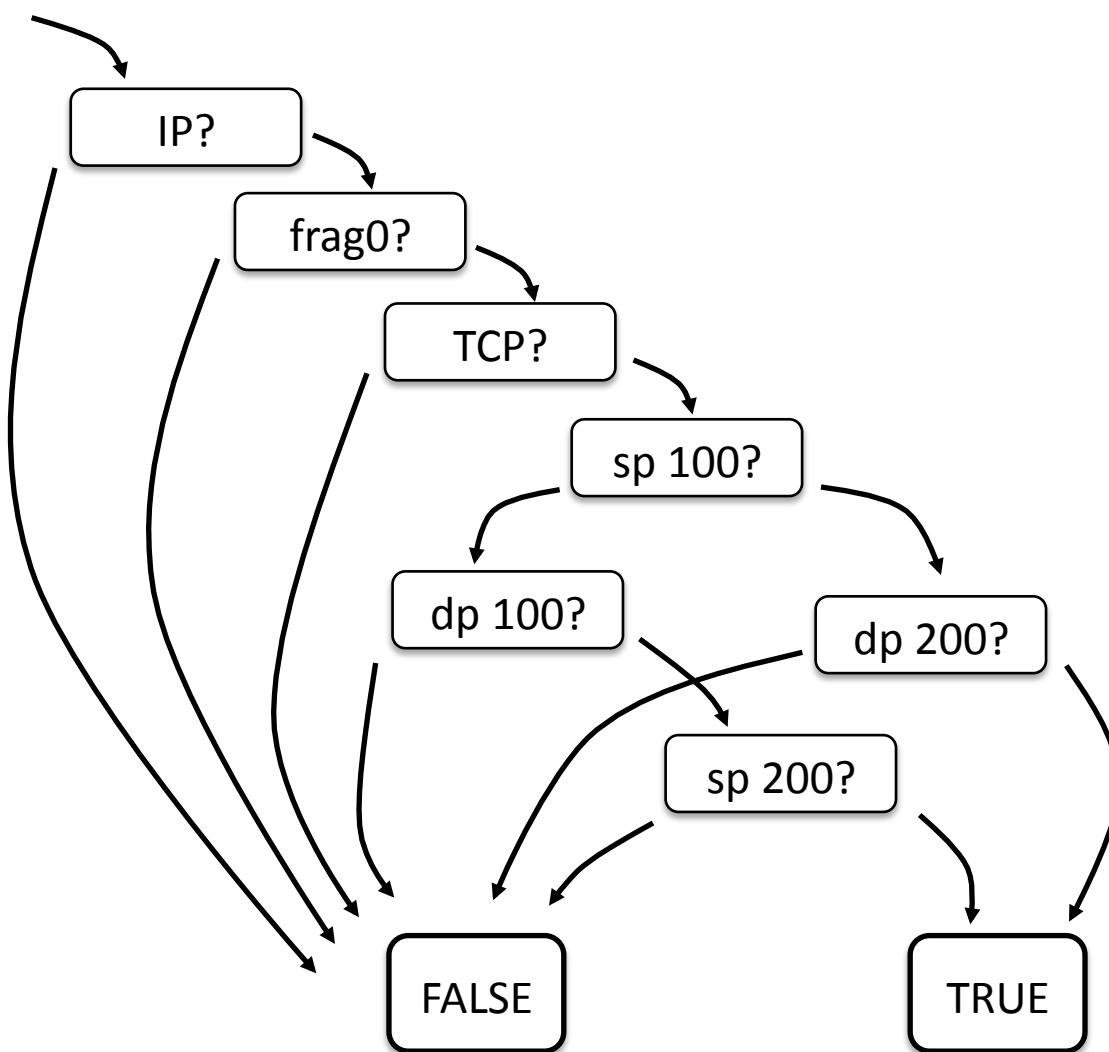
tcp port 100 and 200



tcp port 100 and 200



tcp port 100 and 200



Before

```
(000) ldh [16]
(001) jeq #0x800      jt 2 jf 43
(002) ldh [16]
(003) jeq #0x86dd      jt 4 jf 10
(004) ldb [24]
(005) jeq #0x6       jt 6   jf 10
(006) ldh [58]
(007) jeq #0x64      jt 22jf 8
(008) ldh [60]
(009) jeq #0x64      jt 22jf 10
(010) ldh [16]
(011) jeq #0x800      jt 12 jf 43
(012) ldb [27]
(013) jeq #0x6       jt 14 jf 43
(014) ldh [24]
(015) jset #0x1fff      jt 43 jf 16
(016) idxb 4*([18]&0xf)
(017) ldh [x + 18]
(018) jeq #0x64      jt 22jf 19
(019) idxb 4*([18]&0xf)
(020) ldh [x + 20]
(021) jeq #0x64      jt 22jf 43
```

```
(022) ldh [16]
(023) jeq #0x86dd      jt 24 jf 30
(024) ldb [24]
(025) jeq #0x6       jt 26   jf 30
(026) ldh [58]
(027) jeq #0xc8      jt 42   jf 28
(028) ldh [60]
(029) jeq #0xc8      jt 42   jf 30
(030) ldh [16]
(031) jeq #0x800      jt 32   jf 43
(032) ldb [27]
(033) jeq #0x6       jt 34   jf 43
(034) ldh [24]
(035) jset #0x1fff      jt 43   jf 36
(036) idxb 4*([18]&0xf)
(037) ldh [x + 18]
(038) jeq #0xc8      jt 42   jf 39
(039) idxb 4*([18]&0xf)
(040) ldh [x + 20]
(041) jeq #0xc8      jt 42   jf 43
(042) ret #65535
(043) ret #0
```

After

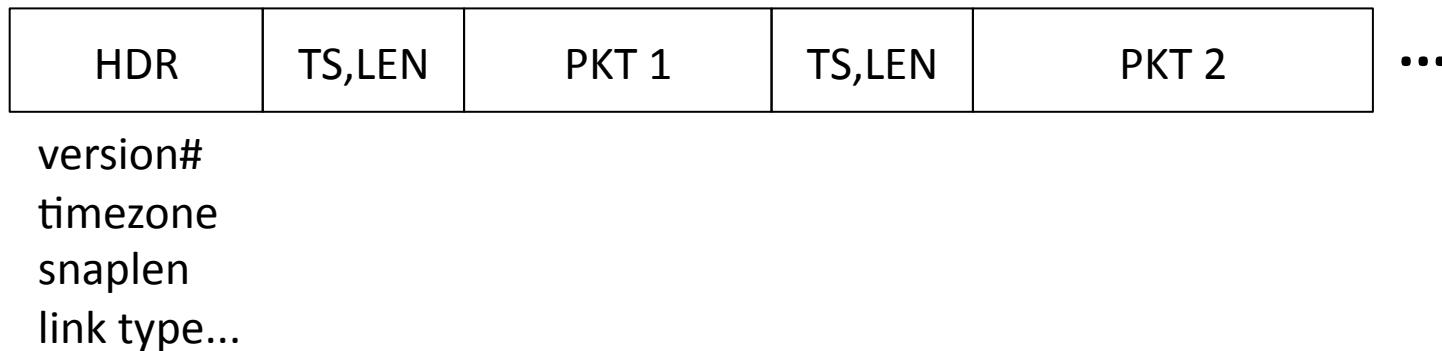
```
(000) ldh [12]
(001) jeq #0x800      jt 2 jf 15
(002) ldb [23]
(003) jeq #0x6      jt 4 jf 15
(004) ldh [20]
(005) jset #0x1fff    jt 15 jf 6
(006) idxb 4*([14]&0xf)
(007) ldh [x + 14]
(008) jeq #0x64      jt 9 jf 11
(009) ldh [x + 16]
(010) jeq #0xc8      jt 14 jf 15
(011) jeq #0xc8      jt 12 jf 15
(012) ldh [x + 16]
(013) jeq #0x64      jt 14 jf 15
(014) ret #65535
(015) ret #0
```

libpcap

- We realized we wanted to build other packet capture applications beyond tcpdump
 - Pulled compiler system and filtering engine out of tcpdump
 - Created an “API” and reusable library
 - Released as “libpcap”
- If different apps were going to be built around this common library, we should have an interchangeable file format for packets traces

pcap File Format

- Elaboration of the “-w” flag to tcpdump
 - `tcpdump -w http.pcap port 80`
 - Bypass protocol decoding logic in tcpdump
 - Write packets straight to disk
 - Run as fast as possible to minimize drops



An Open Approach

- Released tcpdump, BPF, libpcap as open source
 - Ported to various operating systems
 - Berkeley Unix (BSD), SunOS, HP, SGI, DEC
- Eventually adopted in Linux and Mac OS X
- Published in USENIX 93, SIGCOMM 99
- My apologies... escaped before it was done
 - never quite finished libpcap API, then I read about it in Rich Stevens' TCP/IP Illustrated
 - Loris tells me I messed up the pcap file format ☹

Summary

- So, that's my story of libpcap
- I'm very honored and excited to return to the packet capture community after all these years...
 - I would have thought all the problems were solved but as we dig deeper every day, it's clear there is tons of opportunity for innovation...
 - I am looking forward to working with Loris, Gerald, and the community to continue to push the envelope
- It's hard to make things easy, but it's worth it in the end