Programming Network Stack for Middleboxes with Rubik

Hao Li[1], Changhao Wu[1,2], Guangda Sun[1],

Peng Zhang[1], Danfeng Shan[1], Tian Pan[3], Chengchen Hu[4]

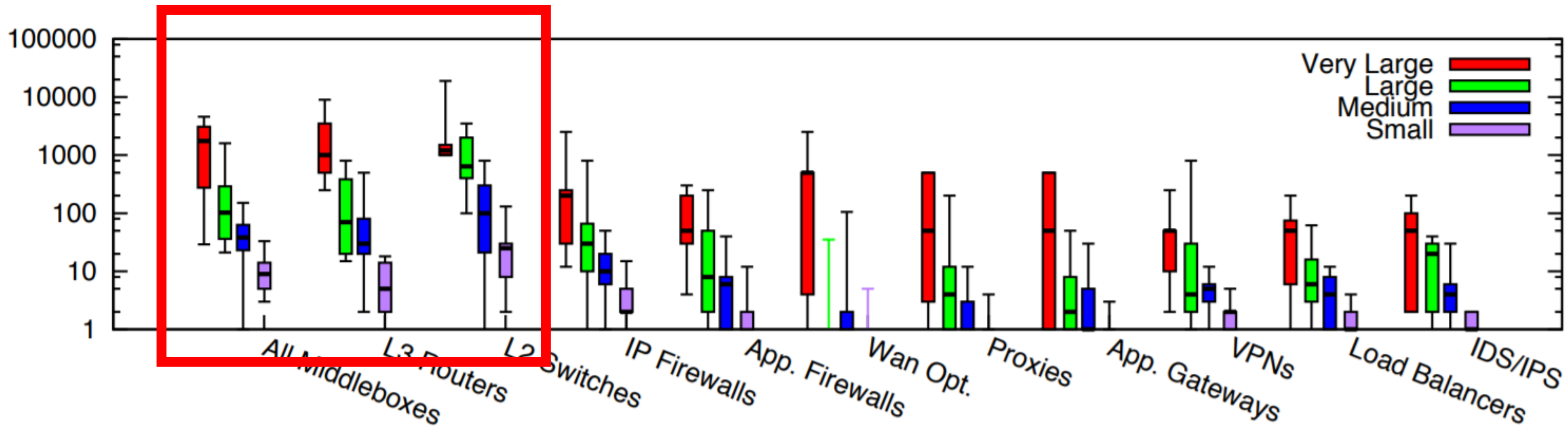
西安交通大學 XI'AN JIAOTONG UNIVERSITY

BROWN

BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS 北京邮电大学

XILINX

Middleboxes are Indispensable



Small: < 1K hosts

Medium: 1K~10K hosts

Large: 10K~100K hosts

Very Large: >100K hosts

…but are Hard to Develop

Huge number of LOC

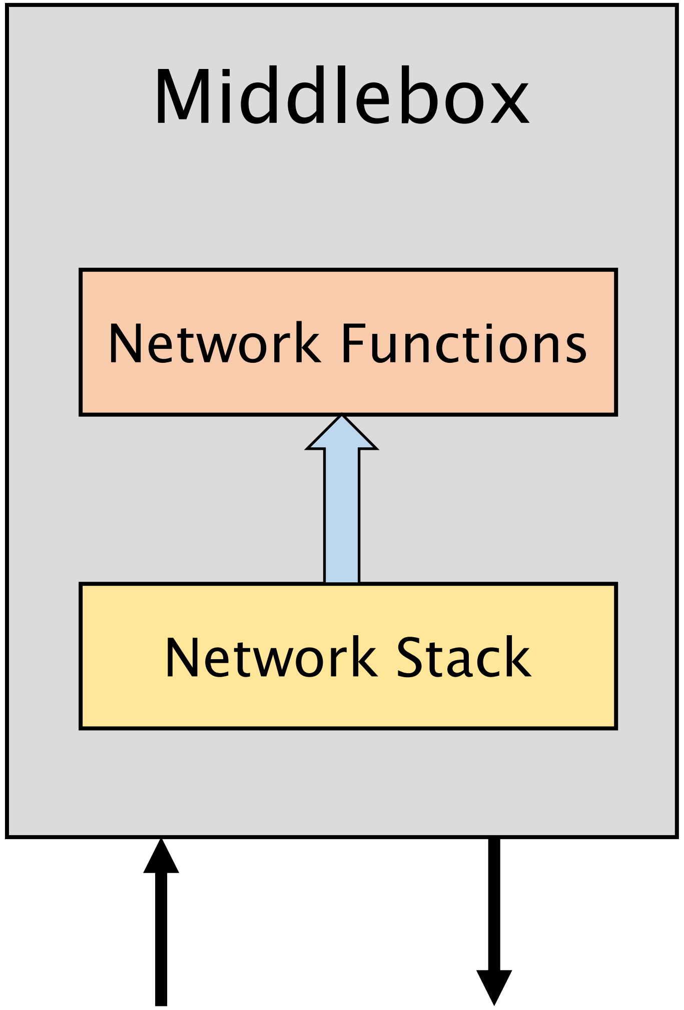
- Snort: 2.5K files, ~300K LOC
- nDPI: 300 files, ~50K LOC
- PRADS: 100 files, ~10K LOC

…in native (low-level) language

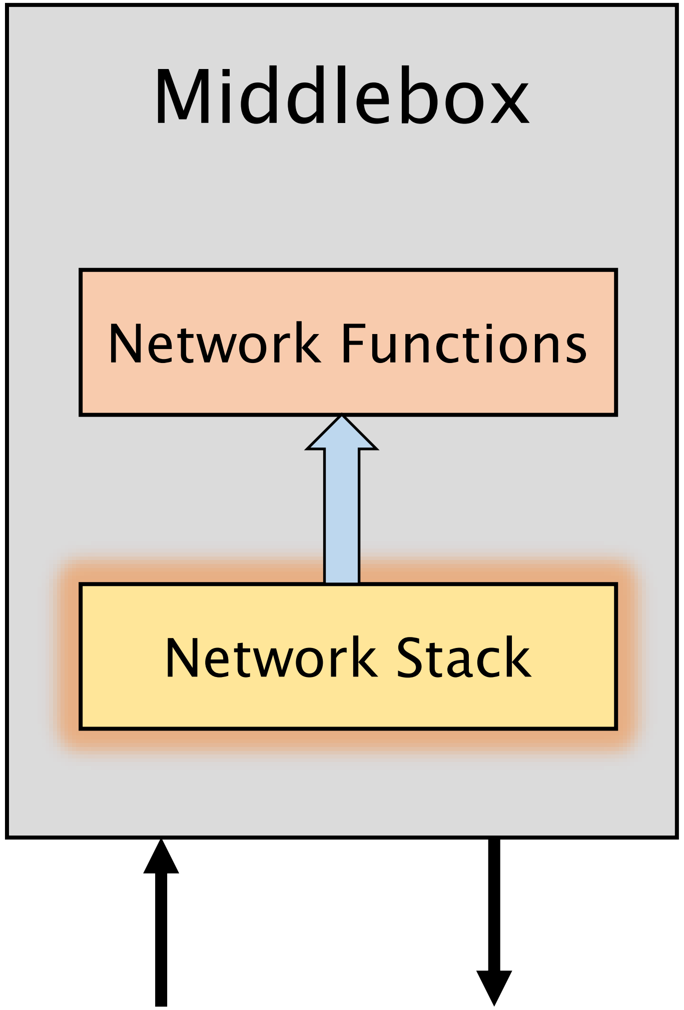
- To ensure the line-rate processing
- C/C++ dominates the implementation of middlebox

Why So Many LOC in a Middlebox?

Components of a Middlebox



Components of a Middlebox



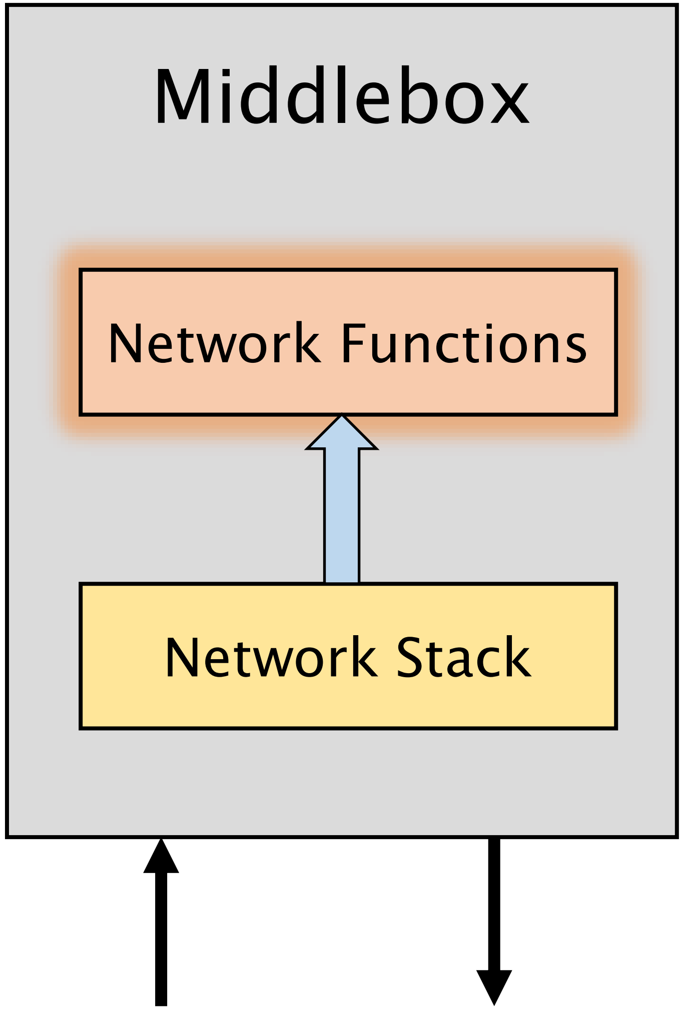
Parse L2-L4 protocols

- Eth, IP, TCP, UDP
- Connection established, teardown

Raise inherent events

- Assembled data
- Orphan packets

Components of a Middlebox



Perform network functions

- Stateful firewall
- Regular expression matching
- L7 proxy

Coding Efforts for Each Component

Network functions: usually <1K LOC

- Simple logic: LB ≈ hashing, IDS ≈ matching
- Reusable libraries: xxHash, PCRE, HyperScan
- Domain-specific tool: FlowSifter → L7 Parser

Network stack: >10K LOC

- Stacked layers instead of a single layer
- Complex logic in each layer: out-of-order pkts

Reduce Coding Efforts in Network Stack

Build a unified stack for all functions

- TCP/IP dominates the traffic (>95%)
- “Hide” the stack with a unified TCP/IP interface
- mOS [NSDI’17], Microboxes [SIGCOMM’18]

…but the stacks are not that unified

Diverse Stack Implementation

- Protocols for customized networks
  - 802.3/802.11 suit in industry/cellular networks
  - New transport: QUIC, SCTP, COTP
- Diverse needs for inherent events
  - A lost packet in TCP mirrored traffic
  - mOS: keep the hole, libnids: drop the flow
- New functions relying on the modified stack
  - Temporary layer for measuring like INT
  - Secured data inspection on encrypted data

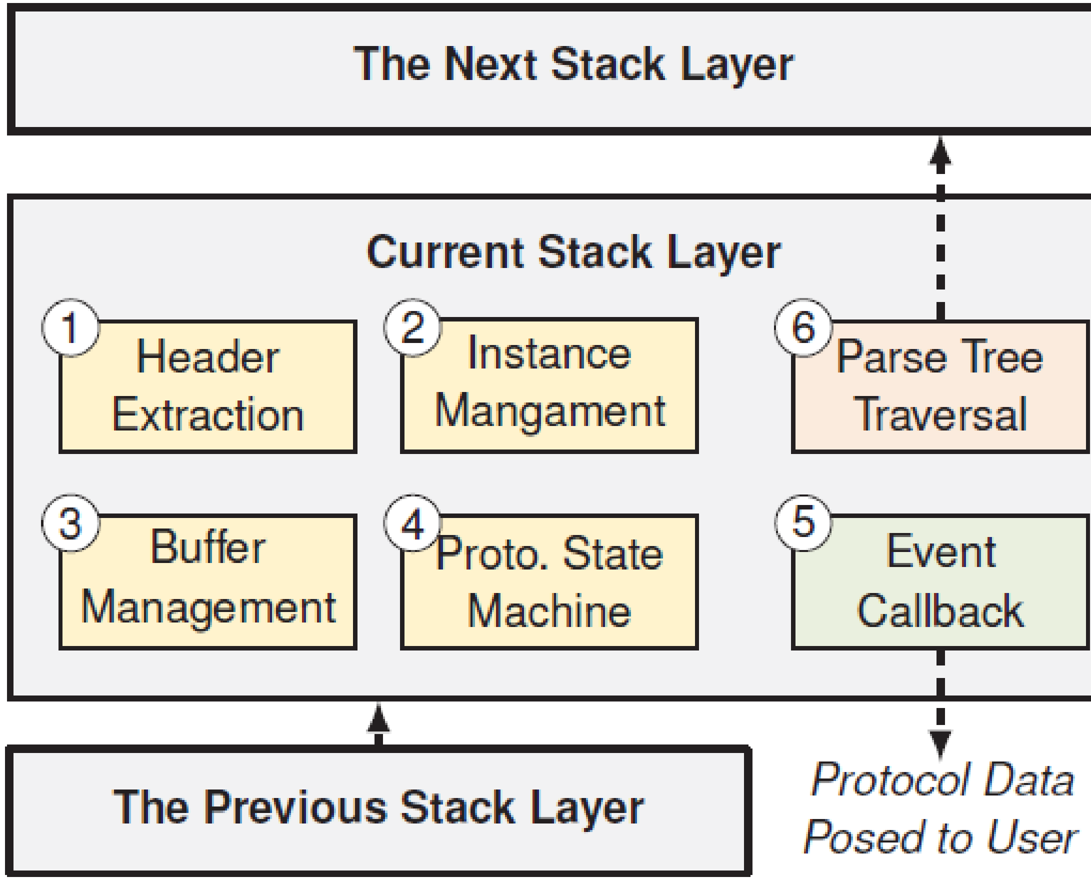
Reduce Coding Efforts in Network Stack

Build a unified stack for all functions

Program stack with domain-specific language

- Capture all semantics in stack processing
- Provide domain-specific abstractions for stack
- Write minor code but generate massive

A Seemingly Generalized Workflow



A Seemingly Generalized Workflow

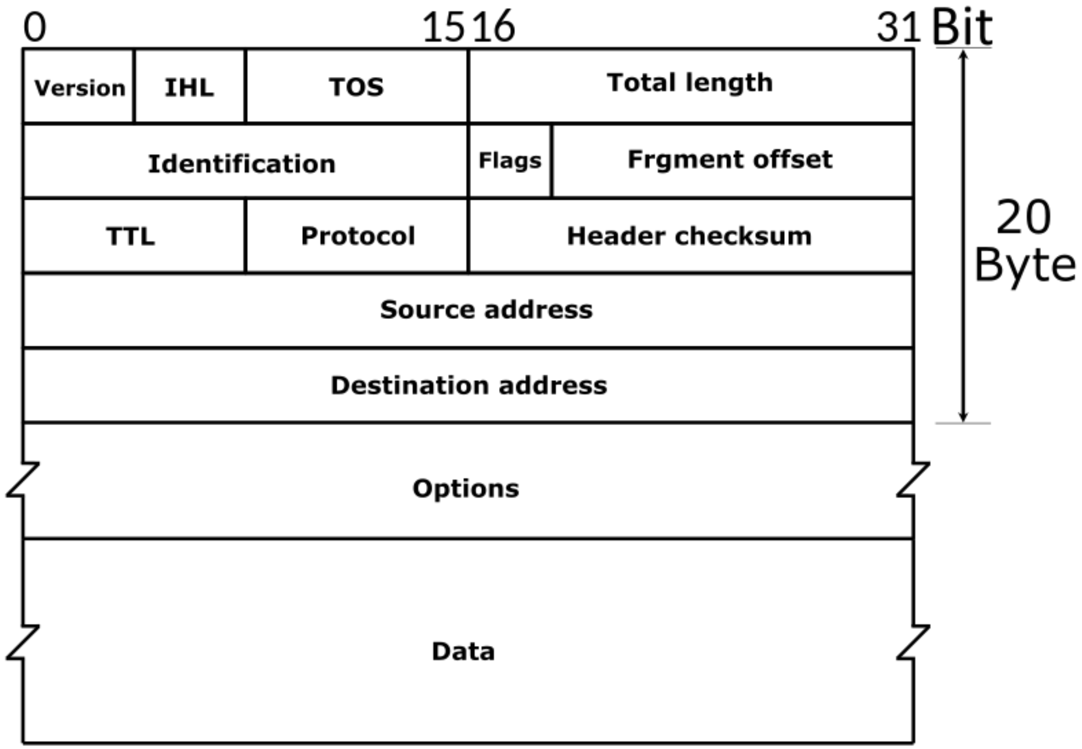
Header Extraction

Instance Management

Buffer Management

Protocol State Machine

Event Callback



A Seemingly Generalized Workflow

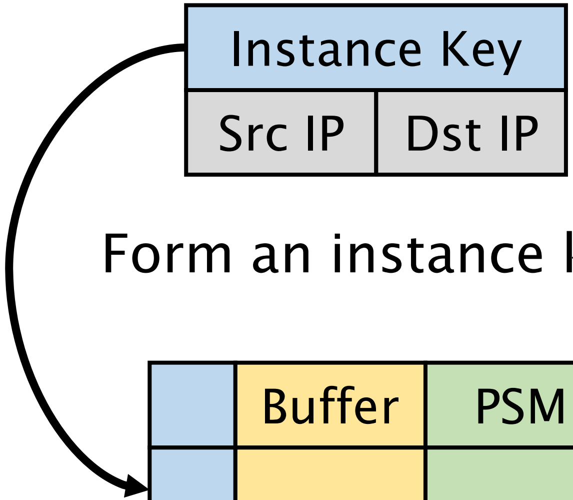
Header
Extraction

Instance
Management

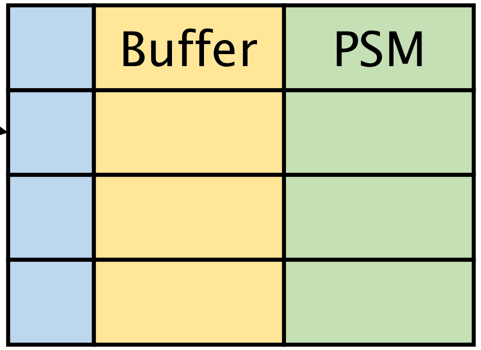
Buffer
Management

Protocol
State Machine

Event
Callback



Form an instance key



Lookup the instance table

Fetch/Create the instance

A Seemingly Generalized Workflow

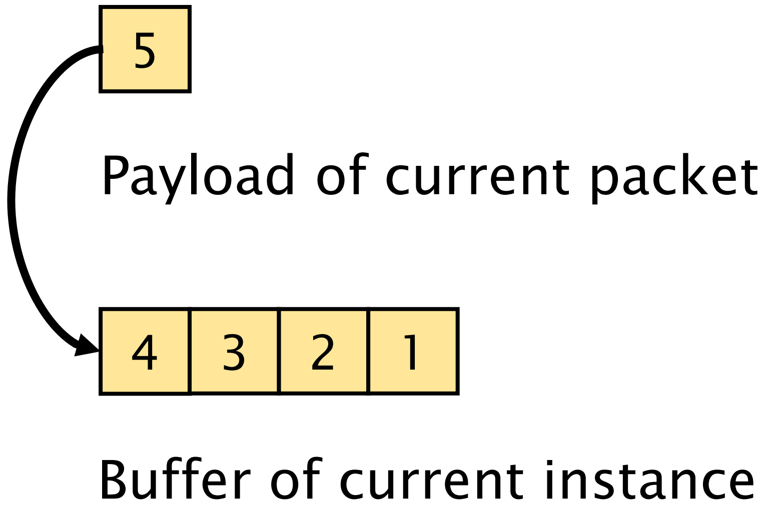
Header
Extraction

Instance
Management

Buffer
Management

Protocol
State Machine

Event
Callback



A Seemingly Generalized Workflow

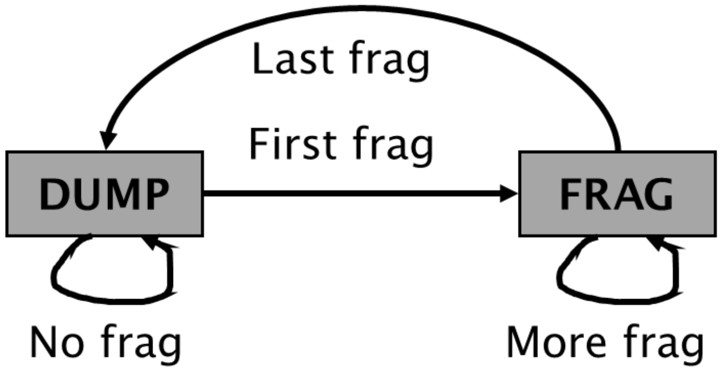
Header
Extraction

Instance
Management

Buffer
Management

Protocol
State Machine

Event
Callback



Simplified IP PSM

A Seemingly Generalized Workflow

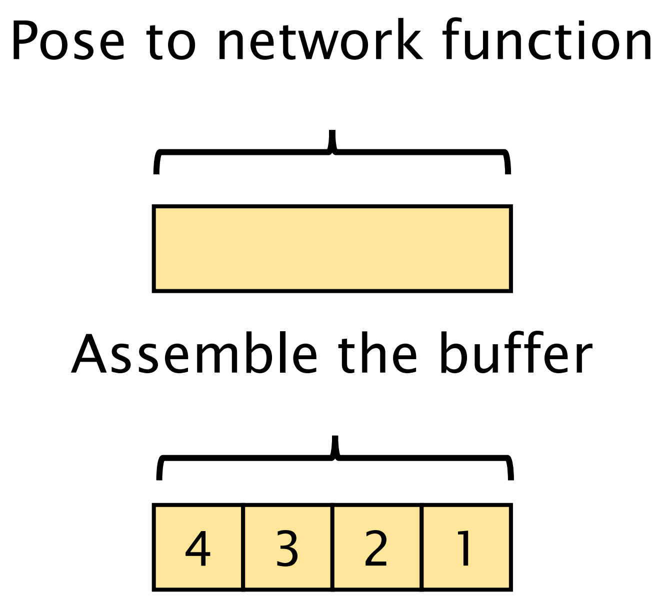
Header
Extraction

Instance
Management

Buffer
Management

Protocol
State Machine

Event
Callback

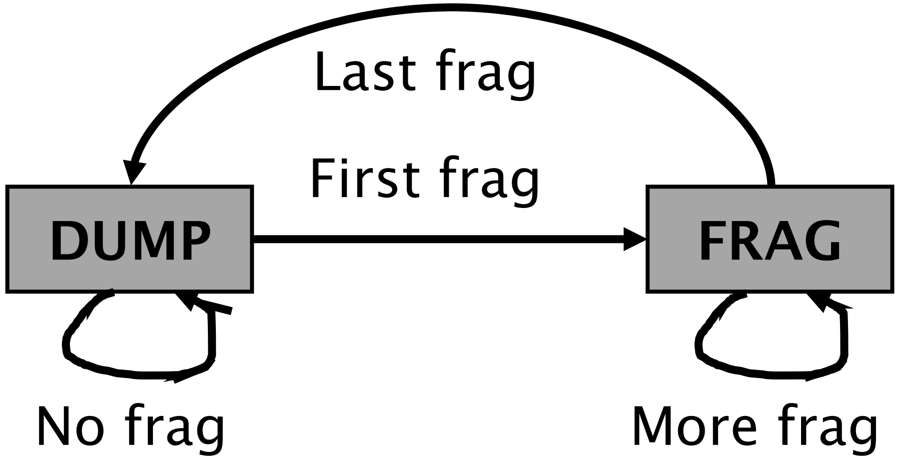


…But is Hard to Implement in a Neat way

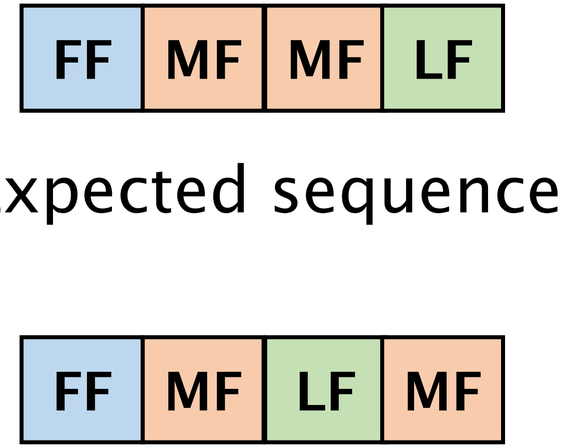
Challenges of Designing a DSL for Middlebox Stack

C1: L2-L4 exceptions mess around workflow

Out-of-order packets wrongly proceed the PSM



Simplified IP PSM

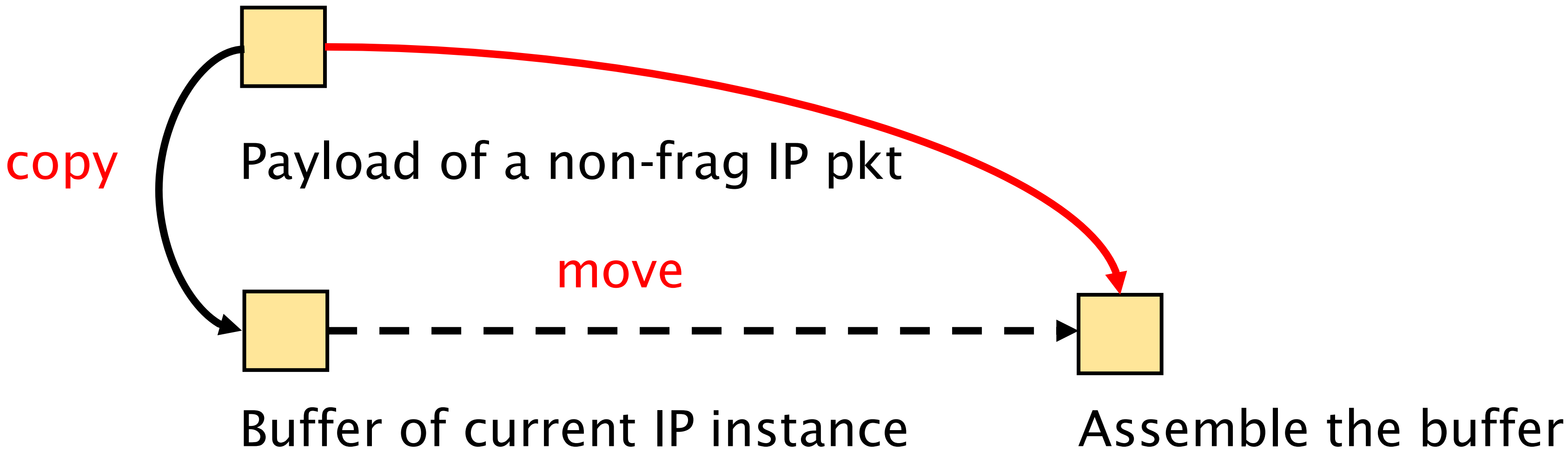


Early-arrived "last frag"

Challenges of Designing a DSL for Middlebox Stack

C2: Line-rate processing

Fast path for special cases breaks the workflow



~~Challenges~~ of Designing a DSL for Middlebox Stack

Dilemma

C1: L2-L4 exceptions mess around workflow

→ High-level abstractions to hide exceptions

C2: Line-rate processing

→ Low-level details to enable the fast path

Introducing Rubik

A Python-based DSL for middlebox stack

A language with domain-specific constructs

- *packet sequence*: buffer sorting, retransmission
- *virtual ordered packet*: out-of-order packet

A compiler with domain-specific optimization

- IR to bridge high-level syntax and low-level code
- Extendable domain-specific optimization

A Walk-through Example

How to write (complex) parser with Rubik?

An IP parser with data assemble and frag events

How to compose stack using existing parsers?

A ETH→IP/ARP stack

Write an IP parser with Rubik

```
# Declare IP layer
ip = Connectionless()
```

```
# Define the header layout
class ip_hdr(layout):
    version = Bit(4)
    ihl = Bit(4)
    ...
    dont_frag = Bit(1)
    more_frag = Bit(1)
    f1 = Bit(5)
    f2 = Bit(8)
    ...
    saddr = Bit(32)
    daddr = Bit(32)
```

Write an IP parser with Rubik

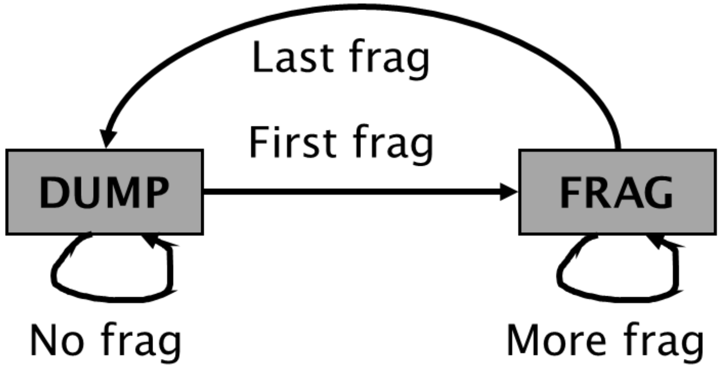
```
# Build header parser
ip.header = ip_hdr
# Specify instance key
ip.selector = [ip.header.src_addr, ip.header.dst_addr]

# Preprocess the instance using 'temp'
class ip_temp(layout):
    offset = Bit(16)
ip.temp = ip_temp
ip.prep = Assign(ip.temp.offset,
                 ((ip.header.f1<<8)+ip.header.f2)<<3)
```

Write an IP parser with Rubik

```
# Manage the packet sequence
ip.seq = Sequence(meta=ip.temp.offset,
                  data=ip.payload[:ip.payload_len])

# Define the PSM transitions
ip.psm.last = (FRAG >> DUMP) + Pred(~ip.header.more_frag)
```



Write an IP parser with Rubik

```
# Buffering event
ip.event.asm = If(ip.psm.last | ip.psm.dump) >> Assemble()

# Callback each IP fragment using 'ipc'
class ipc(layout):
    sip = Bit(32)
    dip = Bit(32)
ip.event.ip_frag = If(~ip.psm.dump) >> \
                   Assign(ipc.sip, ip.header.saddr) + \
                   Assign(ipc.dip, ip.header.daddr) + \
                   Callback(ipc)
```

Compose ETH→IP/ARP Stack

```
st = Stack()
st.eth = ethernet
st.ip  = ip
st.arp = arp
st += (st.eth>>st.ip)  + Pred(st.eth.header.type==0x0800)
st += (st.eth>>st.arp) + Pred(st.eth.header.type==0x0806)
```

Summary of the Example

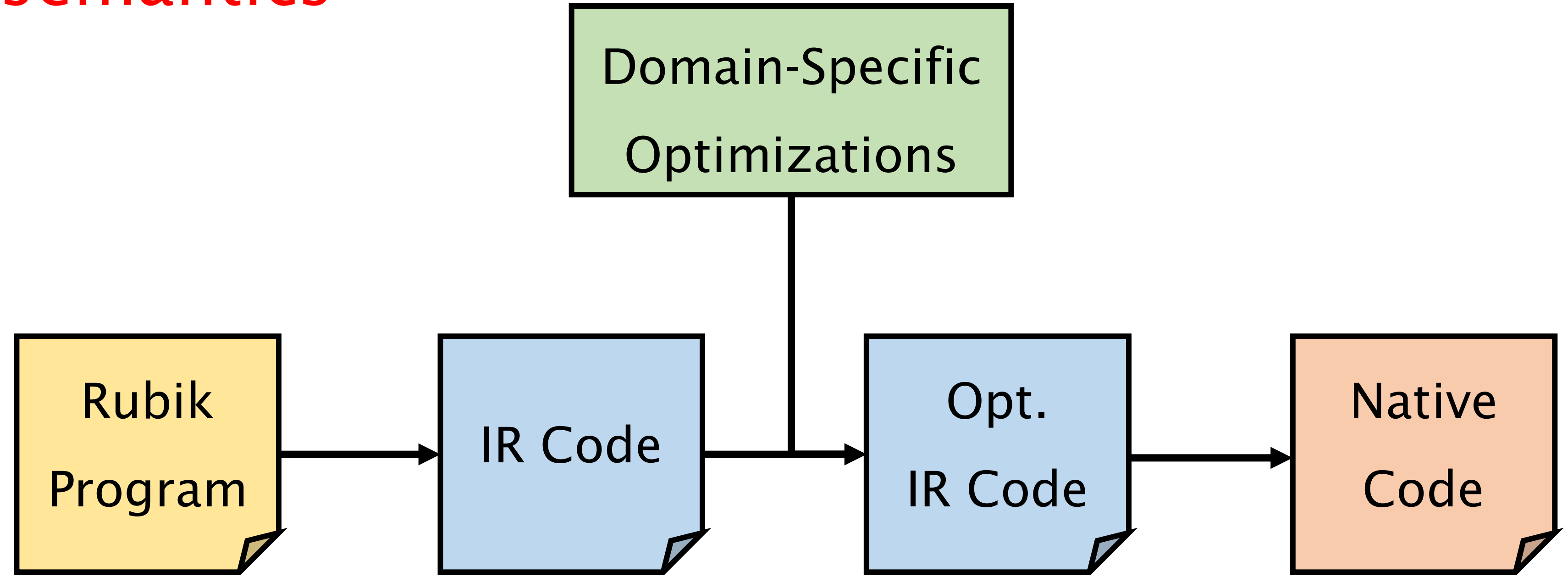
Minor coding efforts

- ~50 and 7 LOC for IP layer and its inherent events
- 6 LOC for building the stack
- *libnids* costs 1.2K C LOC for the similar stack

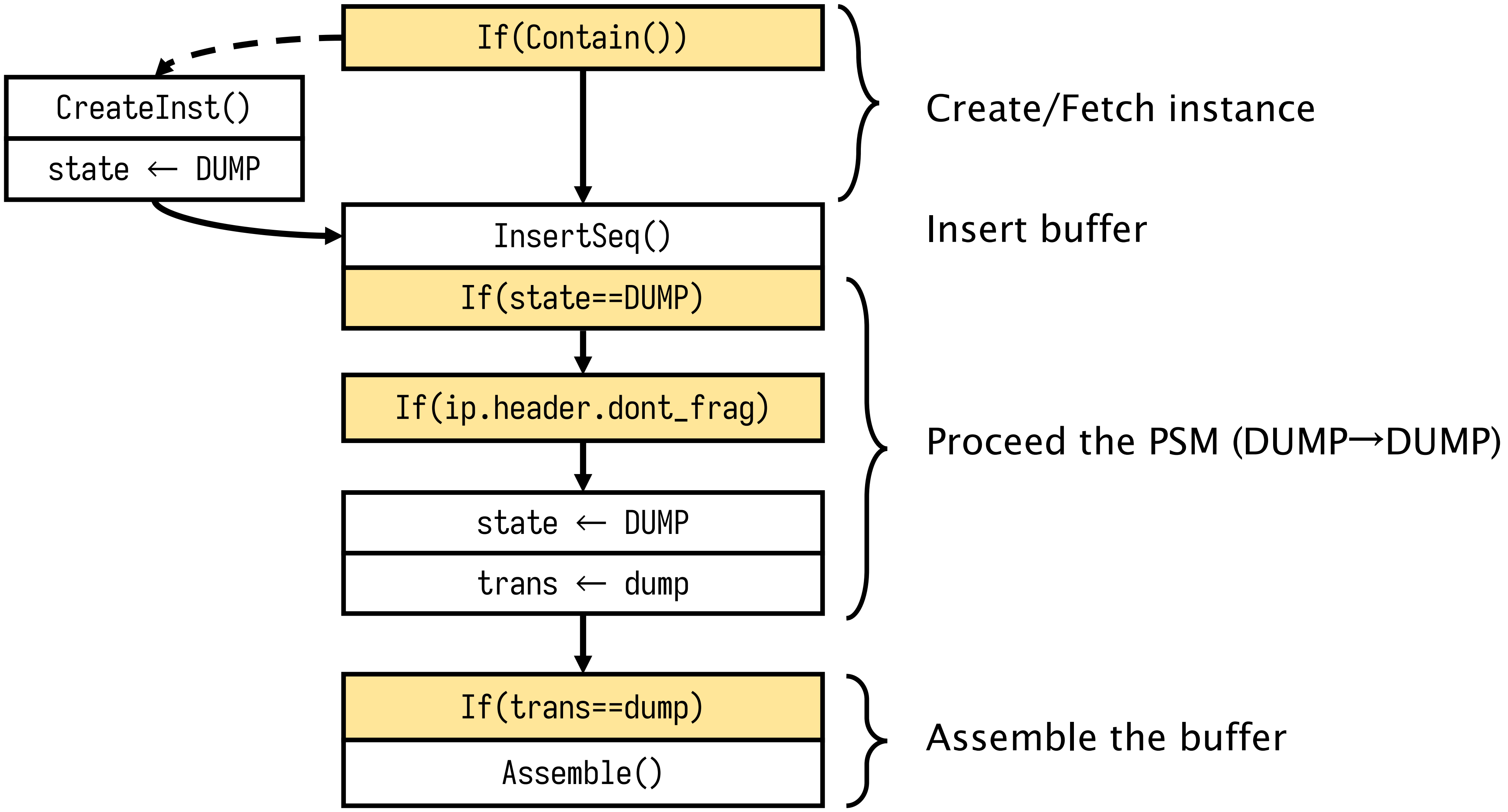
Handy and high-level abstractions are good,
but how to address the dilemma?

A Domain-Specific Compiler

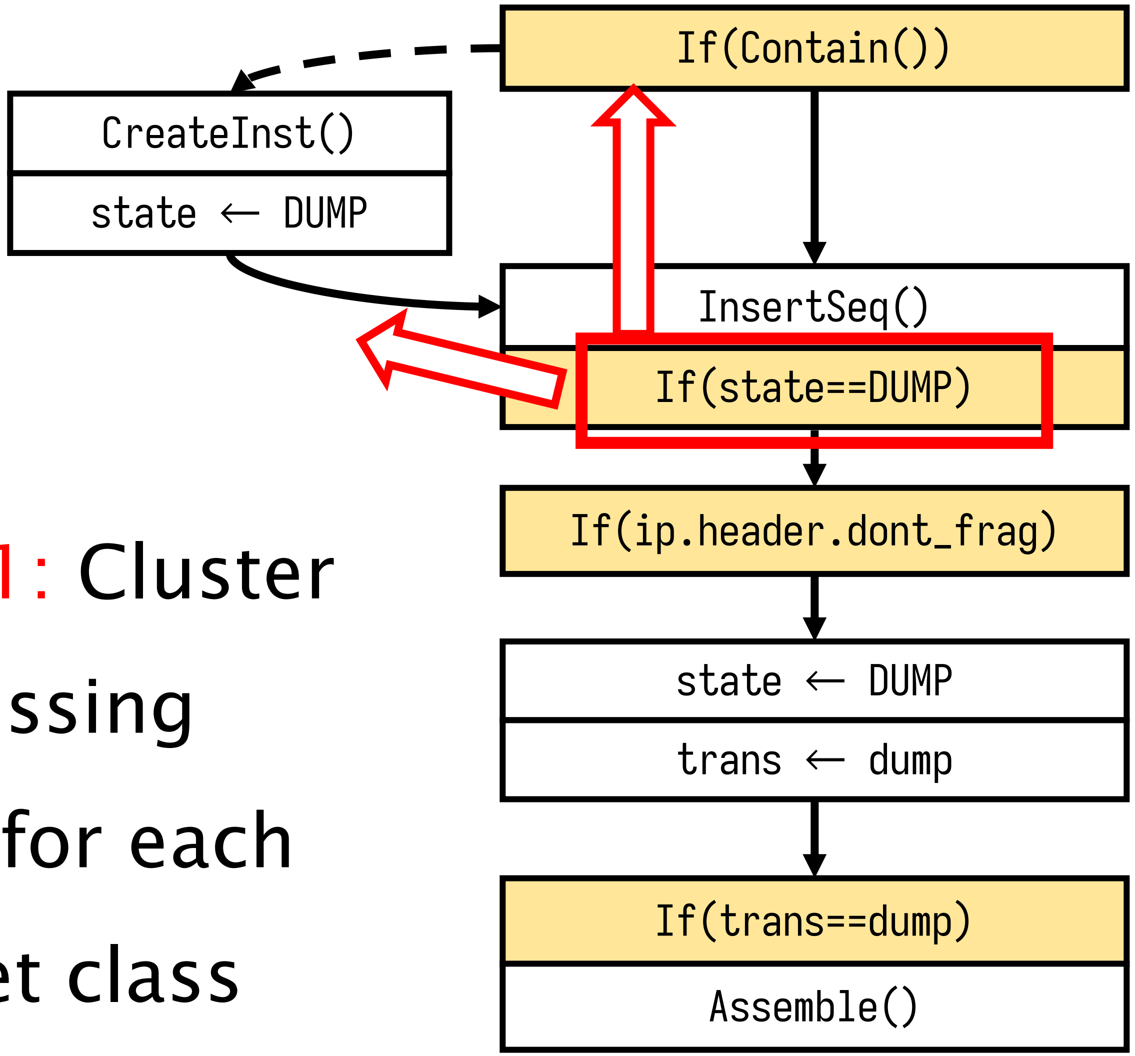
Key enabler: an IR that reveals enough low-level details while maintaining the high-level semantics



Intermediate Representation for IP Parser

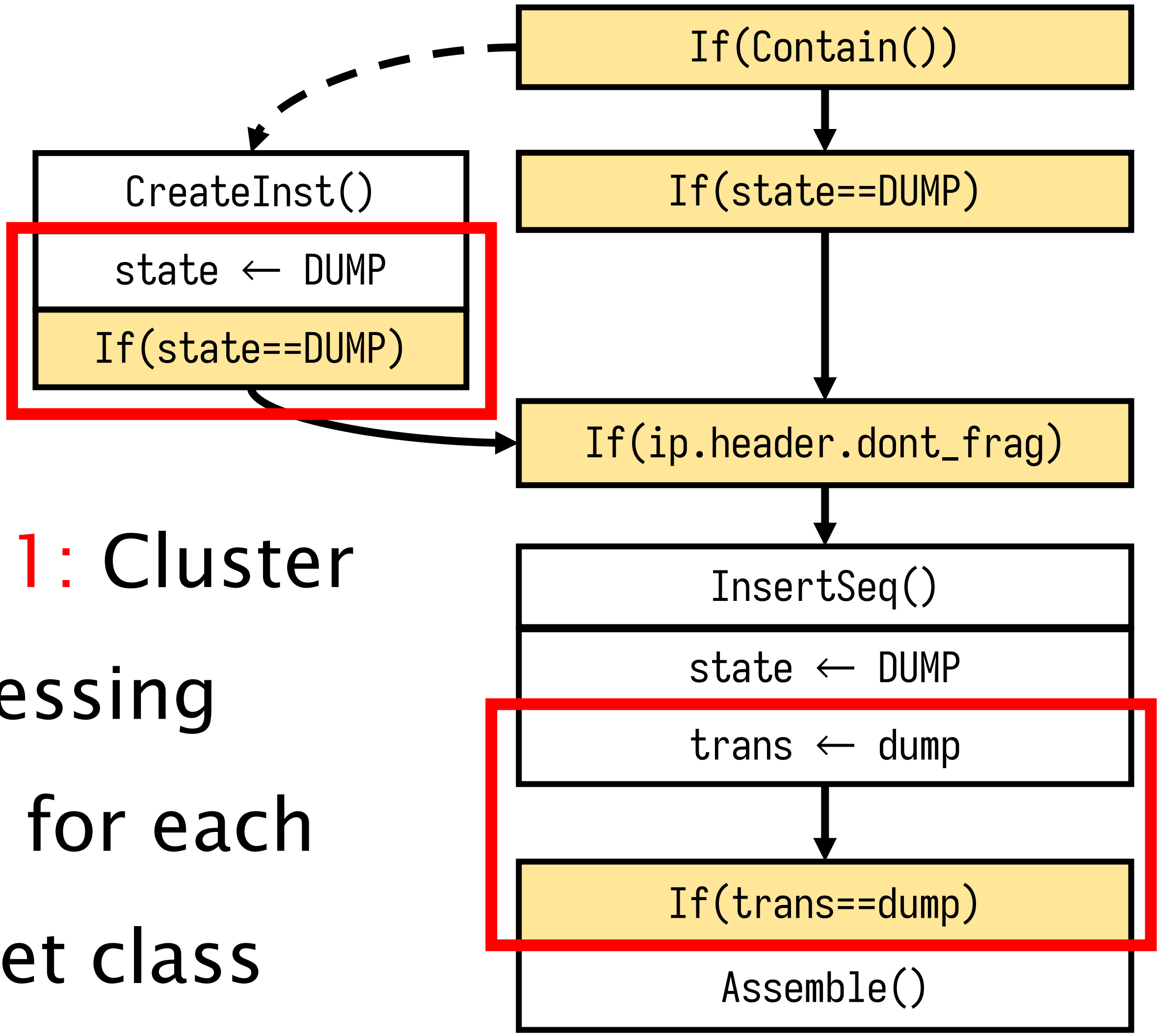


Optimize a Fast Path Automatically



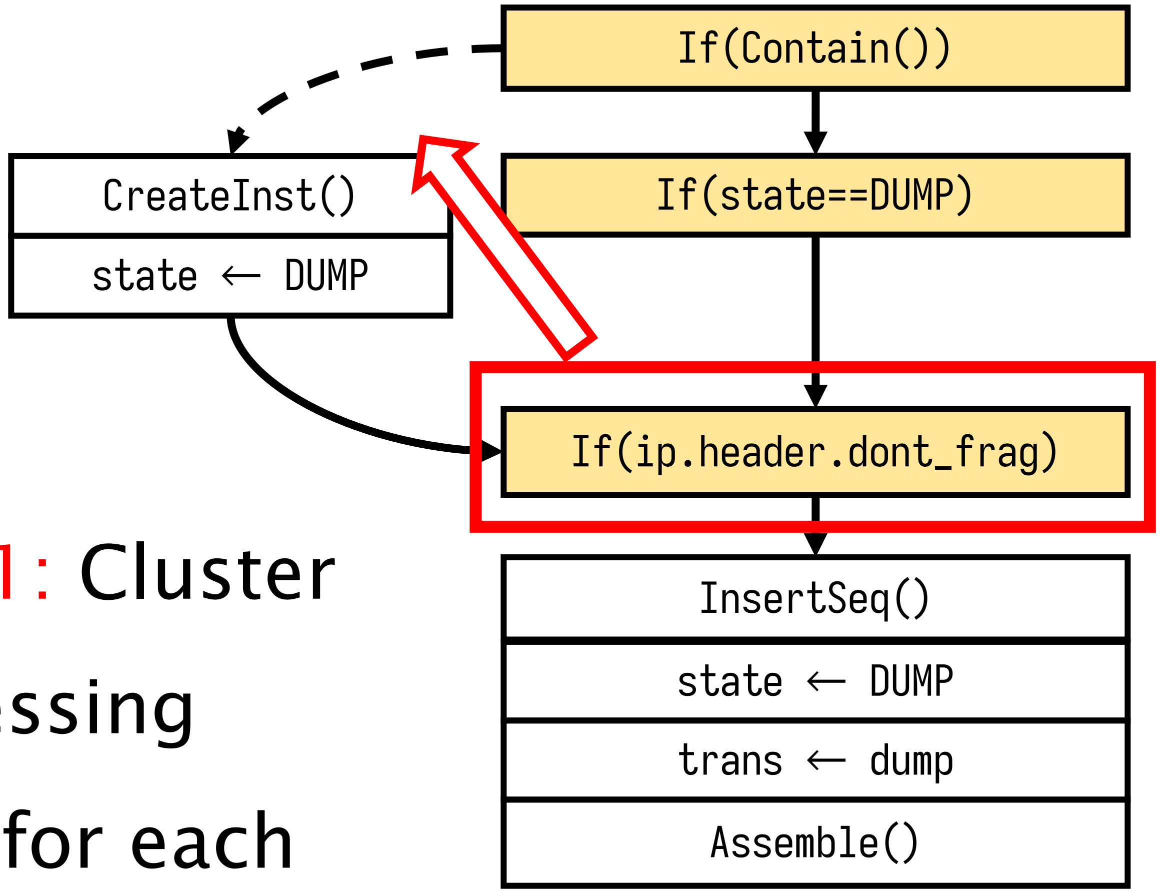
Step 1: Cluster processing logic for each packet class

Optimize a Fast Path Automatically



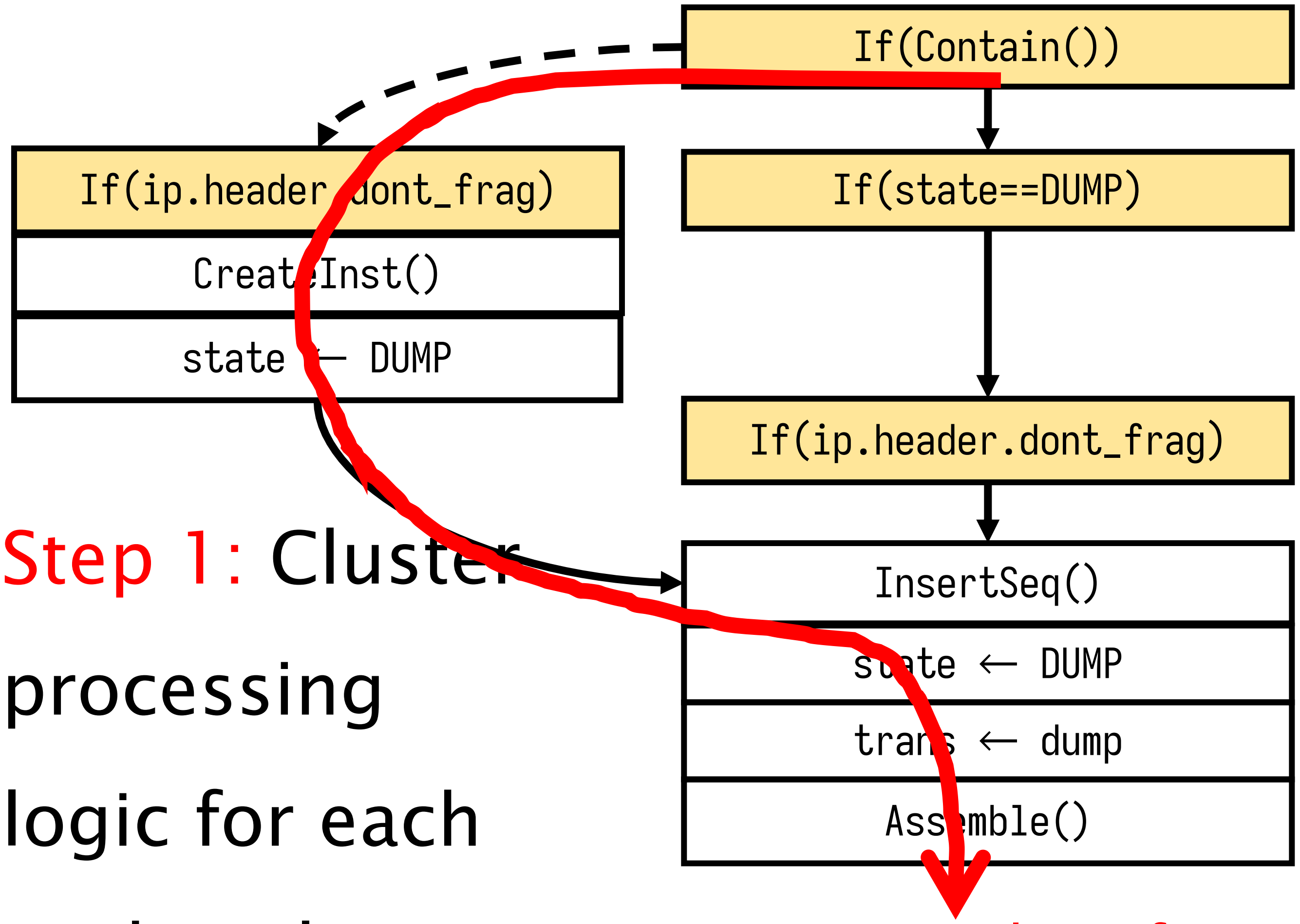
Step 1: Cluster processing logic for each packet class

Optimize a Fast Path Automatically



Step 1: Cluster processing logic for each packet class

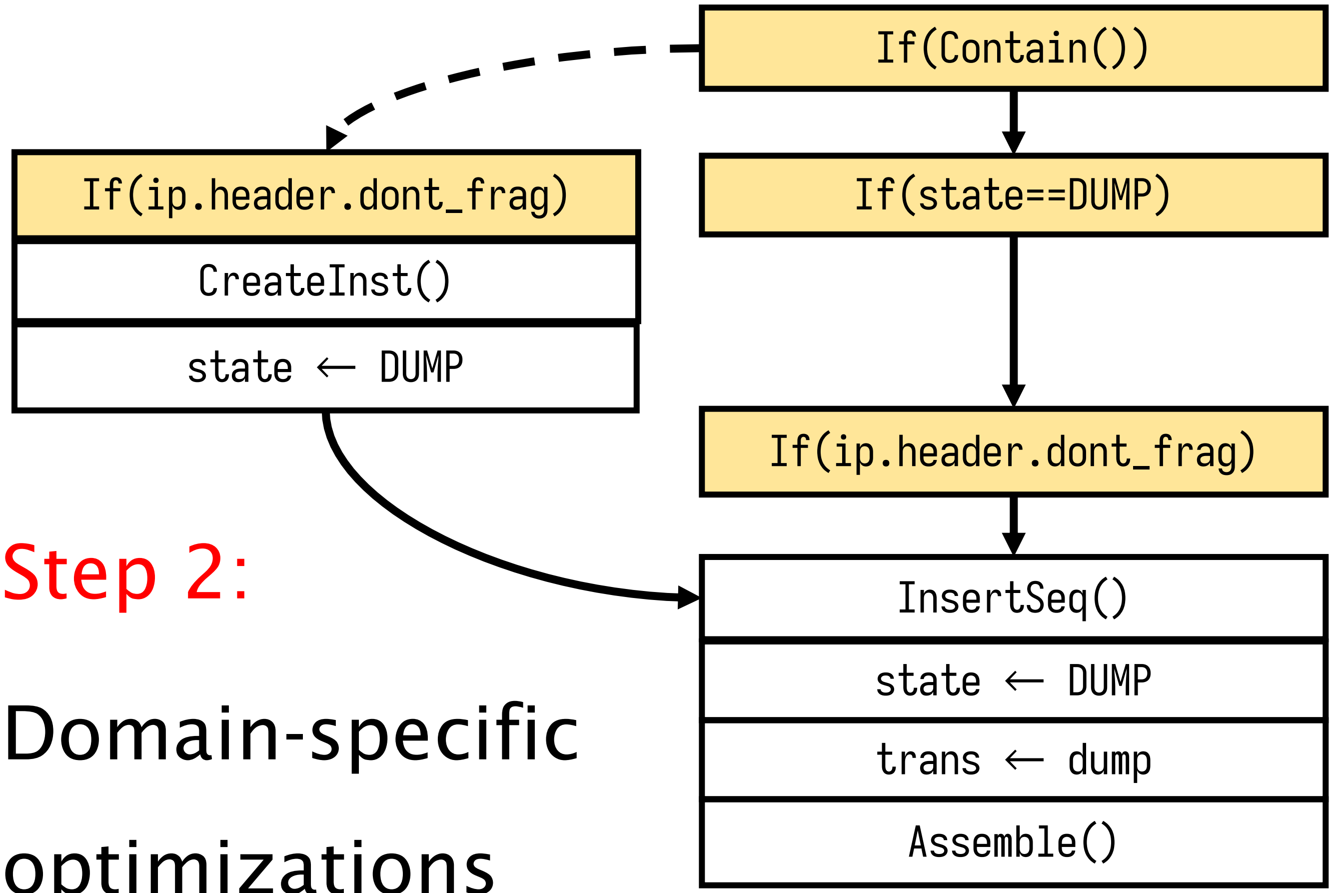
Optimize a Fast Path Automatically



Step 1: Cluster processing logic for each packet class

Processing logic for a non-frag IP packet

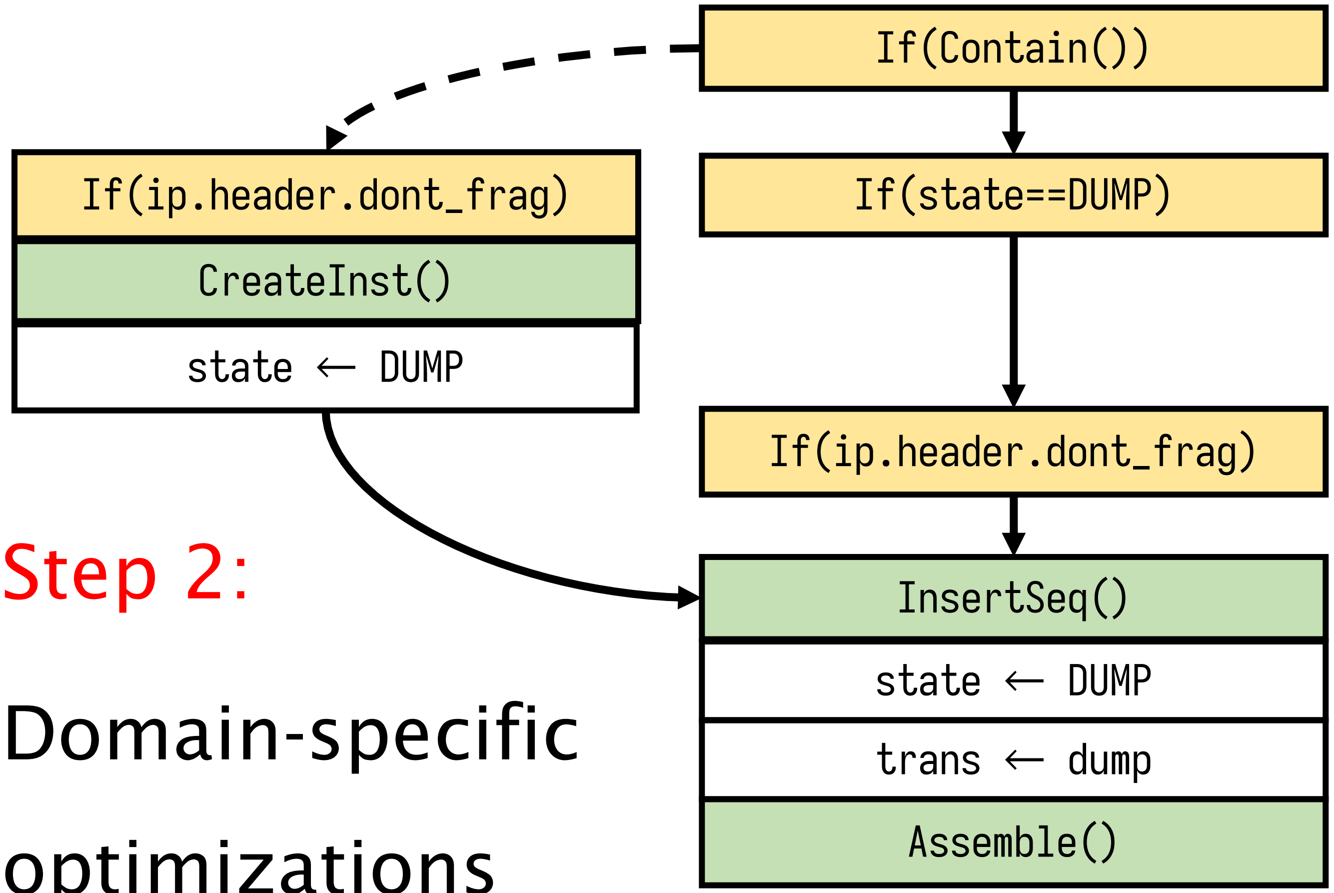
Optimize a Fast Path Automatically



Step 2:

Domain-specific optimizations

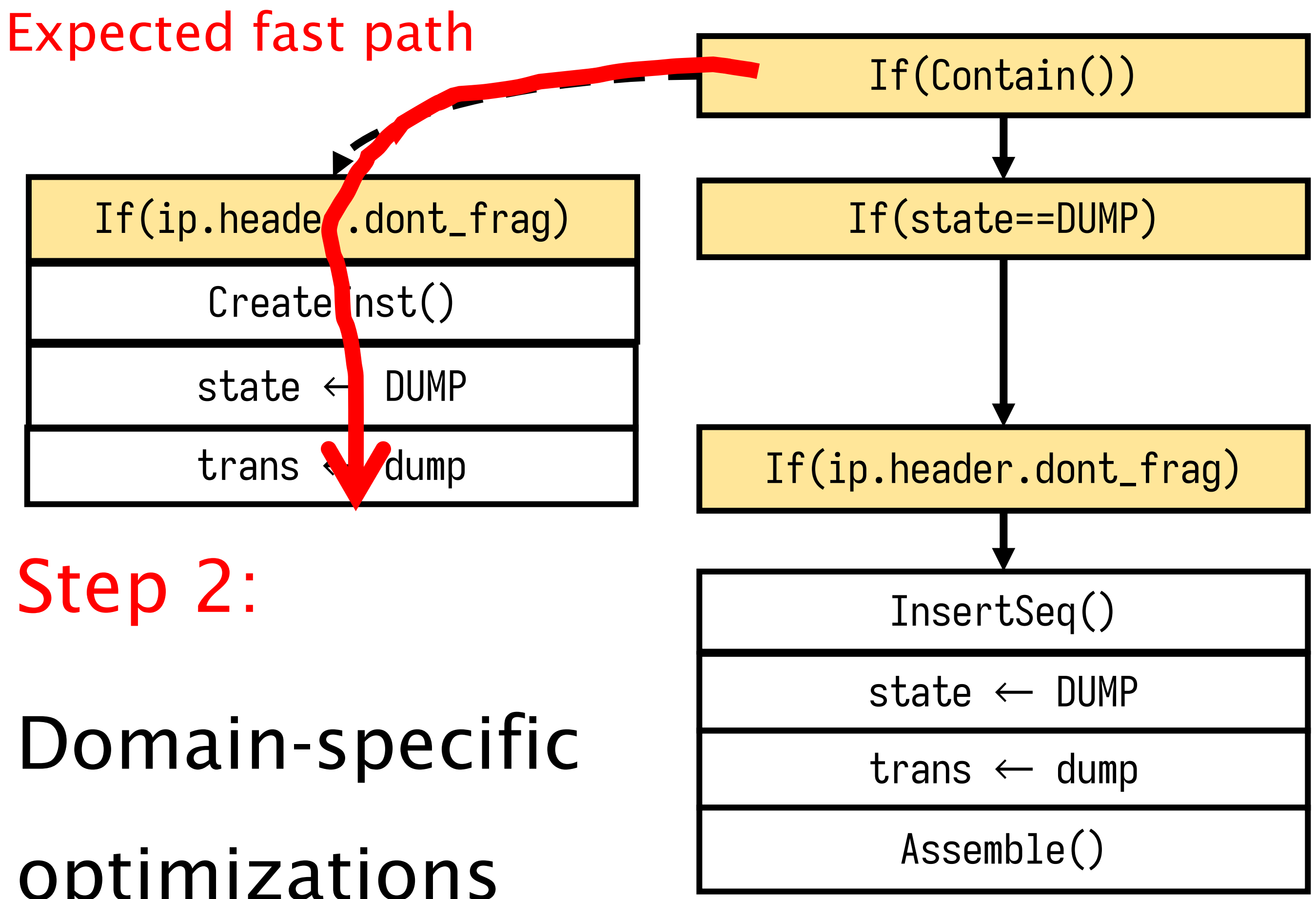
Optimize a Fast Path Automatically



Step 2:

Domain-specific optimizations

Optimize a Fast Path Automatically



Step 2:

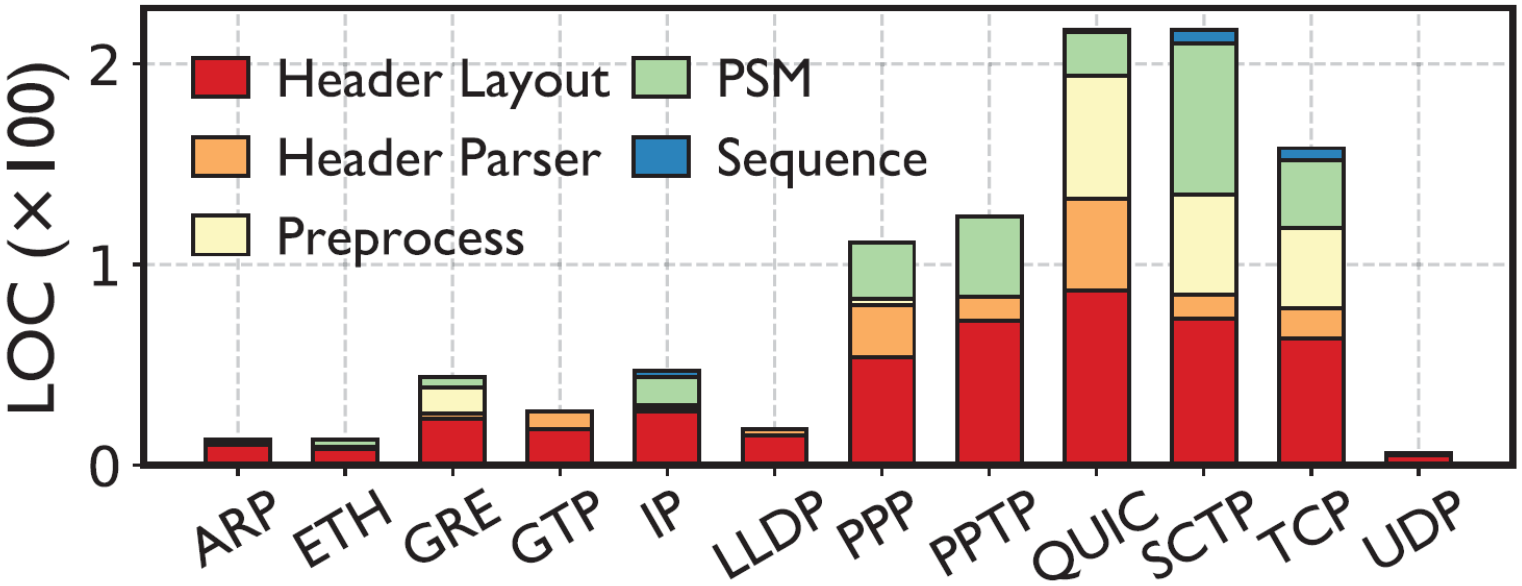
Domain-specific optimizations

Domain-Specific Optimizations

- Borrowed from the common wisdom
  - Currently 4 optimizations are employed
  - Focusing on the “heavy” instructions
- Optimizations ≈ instruction patterns
  - Easy to add more optimizations

Case Study and Evaluations

Case Study: Parsers



Connectionless: tens of LOC

Connection-oriented: a few hundreds of LOC

46% LOC are for defining headers

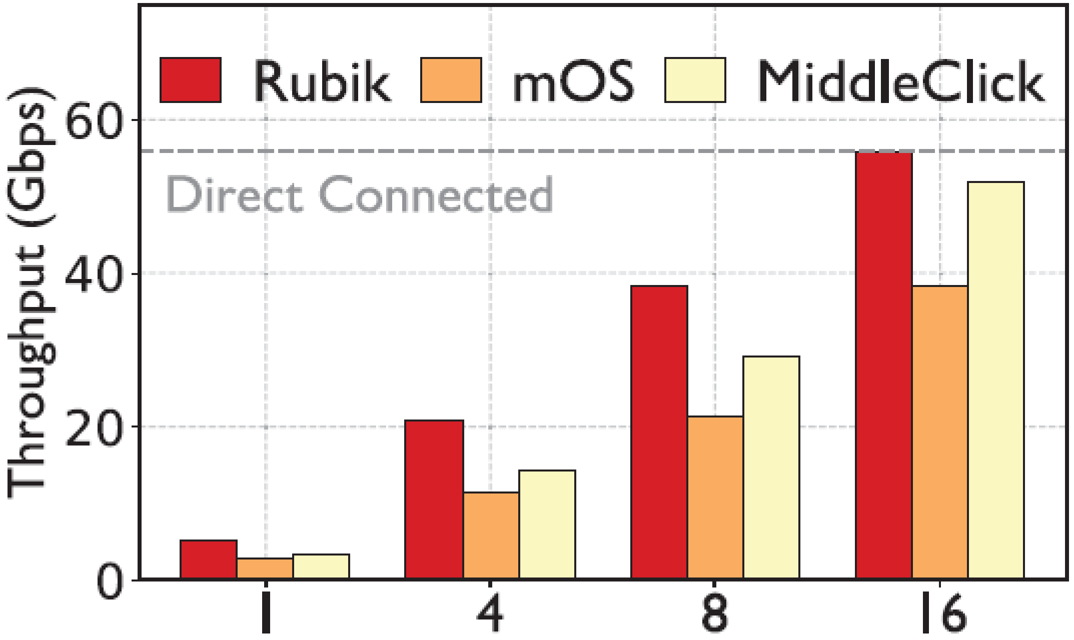
Case Study: Stacks

Stack	Parse Tree	Addi.	Total	Gen.
TCP/IP	ETH→IP→UDP/TCP	14	245	11061
GTP	ETH→IP→UDP→GTP→IP→TCP	18	304	11384
PPTP	ETH→IP→TCP→PPTP ETH→IP→GRE→PPP→IP→TCP	37	586	46546
QUIC	loopback→IP→UDP→QUIC	23	361	14007
SCTP	ETH→IP→SCTP	9	233	23863

Addi.: *additional Rubik LOC apart from the individual protocol parsers*
Total: *total Rubik LOC* **Gen.:** *generated native LOC*

Reusable parsers further facilitate composing the stack

Performance Evaluation: TCP



Rubik outperforms state-of-the-art by 30%-90%

Performance Evaluation: Other Stacks

	Snort	Rubik+Snort	nDPI	Rubik+nDPI	DA
TCP	20.41	26.86	25.94	25.26	117.76
GTP	15.36	22.79	18.87	18.37	113.42
PPTP	13.91	20.01	18.79	18.22	118.41
QUIC	-	-	-	-	116.29
SCTP	-	-	-	-	101.27

Rubik achieves 100Gbps for all involved stacks

Performance Evaluation: Optimizations

	TCP		GTP		PPTP		QUIC		SCTP	
Throughput	8.83	22.4	5.47	14.4	8.16	18.9	7.03	13.7	4.38	6.61
Rubik→C	0.06	0.28	0.07	0.31	0.10	1.19	0.07	0.42	0.06	1.22
C→Binary	3.32	3.47	3.47	5.46	3.73	13.1	3.47	4.02	3.27	6.86

Rubik gains 51%-153% from the optimizations

Conclusion

Programming middlebox stack is a necessity

Rubik, the first DSL for middlebox stack

- Various constructs to reduce coding effort
- Line-rate processing with domain-specific optimizations.

Rubik could be useful and fast

- 12 parsers and 5 stacks with minor LOC
- 30%-90% faster than state-of-the-art

Thanks for Your Attention

Hao Li

hao.li@xjtu.edu.cn