Programming Network Stack for Middleboxes with Rubik

Hao Li\textsuperscript{1}, Changhao Wu\textsuperscript{1,2}, Guangda Sun\textsuperscript{1}, Peng Zhang\textsuperscript{1}, Danfeng Shan\textsuperscript{1}, Tian Pan\textsuperscript{3}, Chengchen Hu\textsuperscript{4}
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

Middlebox

Network Functions

Network Stack
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 \approx \text{hashing}, \text{IDS} \approx \text{matching}

  Reusable libraries: xxHash, PCRE, HyperScan

  Domain-specific tool: FlowSifter \rightarrow \text{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

The Next Stack Layer

Current Stack Layer

1. Header Extraction
2. Instance Management
3. Buffer Management
4. Proto. State Machine
5. Event Callback
6. Parse Tree Traversal

The Previous Stack Layer

Protocol Data Passed to User
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

Instance Key

- Src IP
- Dst IP

Form an instance key

Buffer PSM

Lookup the instance table

Fetch/Create the instance
A Seemingly Generalized Workflow

- Header Extraction
- Instance Management
- Buffer Management
- Protocol State Machine
- Event Callback

Payload of current packet

Buffer of current instance
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

Pose to network function

Assemble the buffer

4 3 2 1
...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**

- **DUMP**
  - Last frag
  - First frag
  - No frag

- **FRAG**
  - More frag

**Expected sequence**

- FF
- MF
- MF
- LF

**Early-arrived “last frag”**

- FF
- MF
- LF
- MF
Challenges of Designing a DSL for Middlebox Stack

C2: Line-rate processing

Fast path for special cases breaks the workflow

Payload of a non-frag IP pkt → Buffer of current IP instance

Copy

Move

Assemble the buffer
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

```python
# 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

CreateInst()
state ← DUMP

If(Contain())

InsertSeq()
If(state==DUMP)

If(ip.header.dont_frag)
state ← DUMP
trans ← dump

If(trans==dump)
Assemble()

Create/Fetch instance
Insert buffer
Proceed the PSM (DUMP → DUMP)
Assemble the buffer
Step 1: Cluster processing logic for each packet class
Optimize a Fast Path Automatically

Step 1: Cluster processing logic for each packet class

If(Contain())
CreateInst()
state ← DUMP
If(state==DUMP)
If(ip.header.dont_frag)
InsertSeq()
state ← DUMP
trans ← dump
If(trans==dump)
Assemble()
Optimize a Fast Path **Automatically**

**Step 1:** Cluster processing logic for each packet class

- **If(Contain())**
  - CreateInst()
  - state $\leftarrow$ DUMP

- **If(state==DUMP)**
  - **If(ip.header.dont_frag)**
    - InsertSeq()
    - state $\leftarrow$ DUMP
    - trans $\leftarrow$ dump
    - Assemble()
Optimize a Fast Path **Automatically**

**Step 1: Cluster processing logic for each packet class**

- If (ip.header.dont_frag)
  - CreateInst()
  - state ← DUMP

- If (state == DUMP)
  - InsertSeq()
  - state ← DUMP
  - trans ← dump
  - Assemble()

Processing logic for a non-frag IP packet
Optimize a Fast Path Automatically

Step 2:
Domain-specific optimizations

If(ip.header.dont_frag)
CreateInst()
state ← DUMP

If(Contain())

If(state==DUMP)

If(ip.header.dont_frag)
InsertSeq()
state ← DUMP
tran ← dump
Assemble()
Optimize a Fast Path Automatically

Step 2: Domain-specific optimizations

If(ip.header.dont_frag)

CreateInst()

state ← DUMP

If(Contain())

InsertSeq()

state ← DUMP

trans ← dump

Assemble()
Optimize a Fast Path Automatically

Step 2:
Domain-specific optimizations

Expected fast path:

If (ip.header.dont_frag)
  CreateInst()
  state ← DUMP
  trans ← dump

If (state==DUMP)

If (ip.header.dont_frag)
  InsertSeq()
  state ← DUMP
  trans ← dump
  Assemble()
Domain-Specific Optimizations

Borrowed from the common wisdom

Currently 4 optimizations are employed

Focusing on the “heavy” instructions

Optimizations \( \approx \) 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

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

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

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Snort</th>
<th>Rubik+Snort</th>
<th>nDPI</th>
<th>Rubik+nDPI</th>
<th>DA</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>20.41</td>
<td>26.86</td>
<td>25.94</td>
<td>25.26</td>
<td>117.76</td>
</tr>
<tr>
<td>GTP</td>
<td>15.36</td>
<td>22.79</td>
<td>18.87</td>
<td>18.37</td>
<td>113.42</td>
</tr>
<tr>
<td>PPTP</td>
<td>13.91</td>
<td>20.01</td>
<td>18.79</td>
<td>18.22</td>
<td>118.41</td>
</tr>
<tr>
<td>QUIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>116.29</td>
</tr>
<tr>
<td>SCTP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>101.27</td>
</tr>
</tbody>
</table>

Rubik achieves 100Gbps for all involved stacks
Performance Evaluation: Optimizations

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