

# **Developing Flow-processing Middleboxes with the mOS Networking Stack**

Project homepage: <http://mos.kaist.edu/>

---

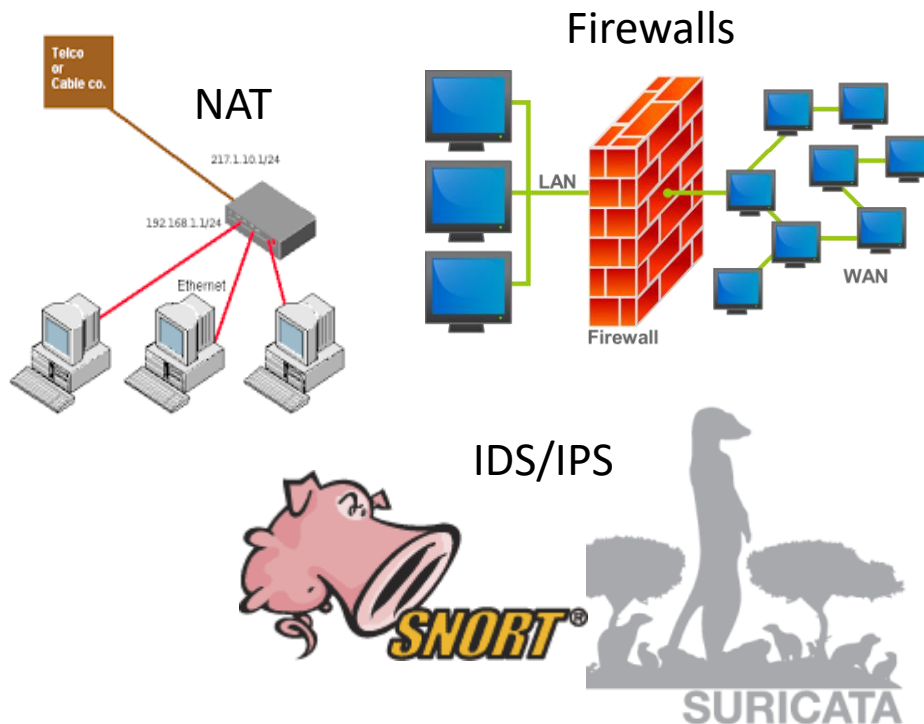
*KYOUNGSOO PARK*

SCHOOL OF ELECTRICAL ENGINEERING, KAIST

# Network Middlebox

Networking devices that provide **extra** functionalities

- Switches/routers = L2/L3 devices
- All others are called middleboxes



Web/SSL proxies

L7 protocol analyzers

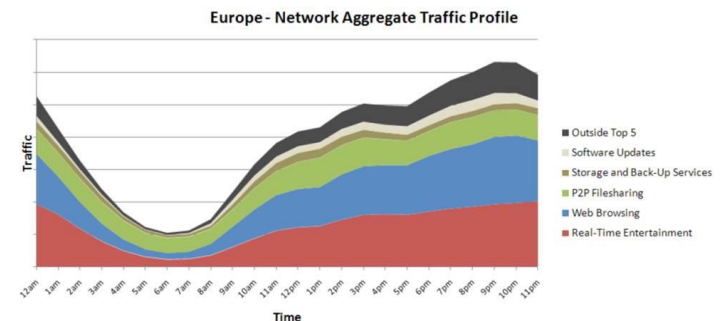
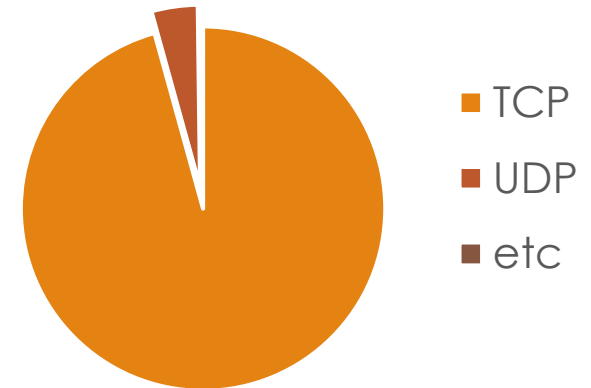


Figure 1. Network aggregate traffic profile. Source: Sandvine

# Most Middleboxes Deal with TCP Flows

- TCP dominates the Internet
  - 95+% of traffic is TCP [1]
- Flow-processing middleboxes
  - Stateful firewalls
  - Protocol analyzers
  - Cellular data accounting
  - Intrusion detection/prevention systems
  - Network address translation
  - And many others!

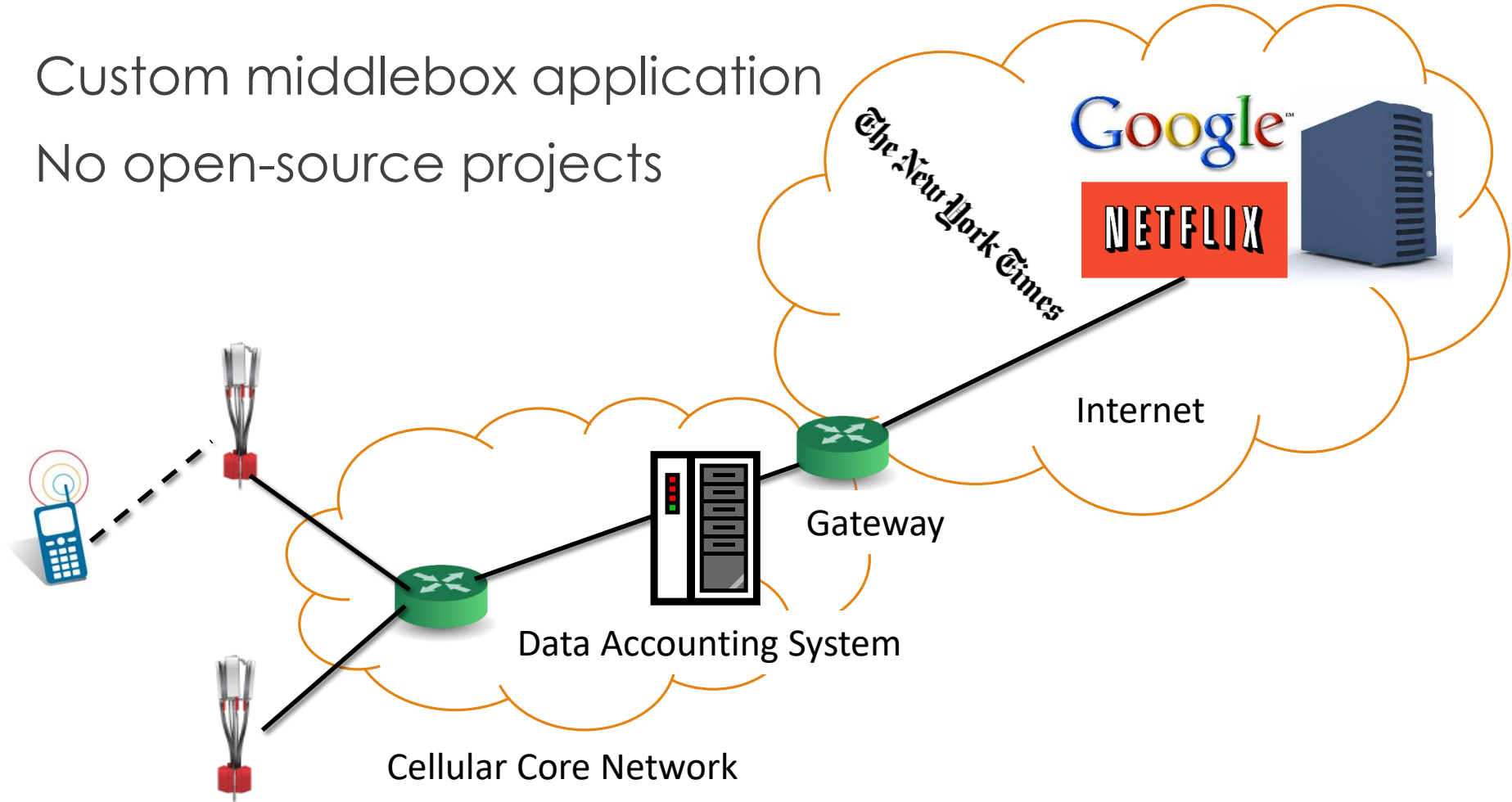


[1] "Comparison of Caching Strategies in Modern Cellular Backhaul Networks", ACM MobiSys 2013.

TCP state management is **complex and error-prone!**

# Example: Cellular Data Accounting System

Custom middlebox application  
No open-source projects



# Develop Cellular Data Accounting System

```
For every IP packet, p
sub = FindSubscriber(p.srcIP, p.destIP);
sub.usage += p.length;
```

Charge for  
retransmission?

```
For every IP packet, p
  if (p is not retransmitted){
    sub = FindSubscriber(p.srcIP, p.destIP);
    sub.usage += p.length;
  }
```

South Korea

TCP tunneling  
attack? [NDSS'14]

```
For every IP packet, p
  if (p is not retransmitted){
    sub = FindSubscriber(p.srcIP, p.destIP);
    sub.usage += p.length;
  } else { // if p is retransmitted
    if (p's payload != original payload) {
      report abuse by the subscriber;
    }
  }
```

Attack Detection

Logically, simple  
process!

# Cellular Data Accounting Middlebox

---

## Core logic

- Determine if a packet is retransmitted
- Remember the original payload (e.g, by sampling)
- Key: TCP flow management

## How to implement?

- Borrow code from open-source IDS (e.g., Snort/Suricata)
- Problem: 50~100K code lines tightly coupled with their IDS logic

## Another option?

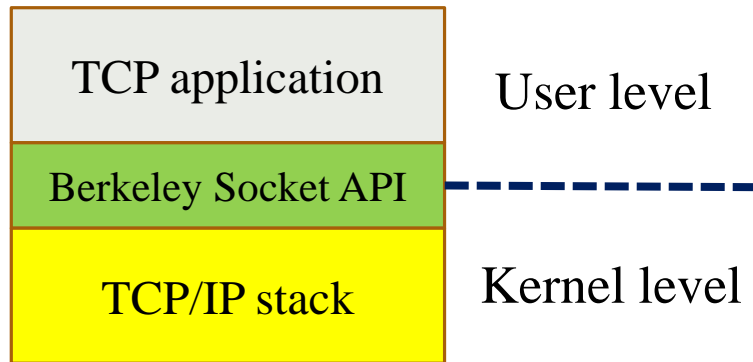
- Borrow code from open-source kernel (e.g., Linux/FreeBSD)
- Problem: kernel is for one end, so it lacks middlebox semantics

## What is the common practice? state-of-the-art?

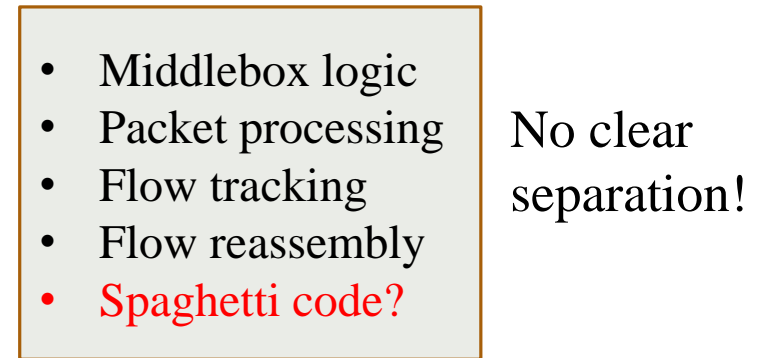
- Implement your own flow management
- Problem: repeat it for every custom middlebox

# Programming TCP End-Host Application

- Typical TCP end-host applications



- Typical TCP middleboxes?



## Berkeley socket API

- Nice abstraction that separates flow management from application
- Write better code if you know TCP internals
- **Never** requires you to write *TCP stack itself*

# mOS Networking Stack

---

## Reusable networking stack for middleboxes

- Programming abstraction & a well-defined set of API

## Key concepts

- Separation of flow management from custom logic
- Event-based middlebox development (event/action)
- Per-flow flexible resource consumption

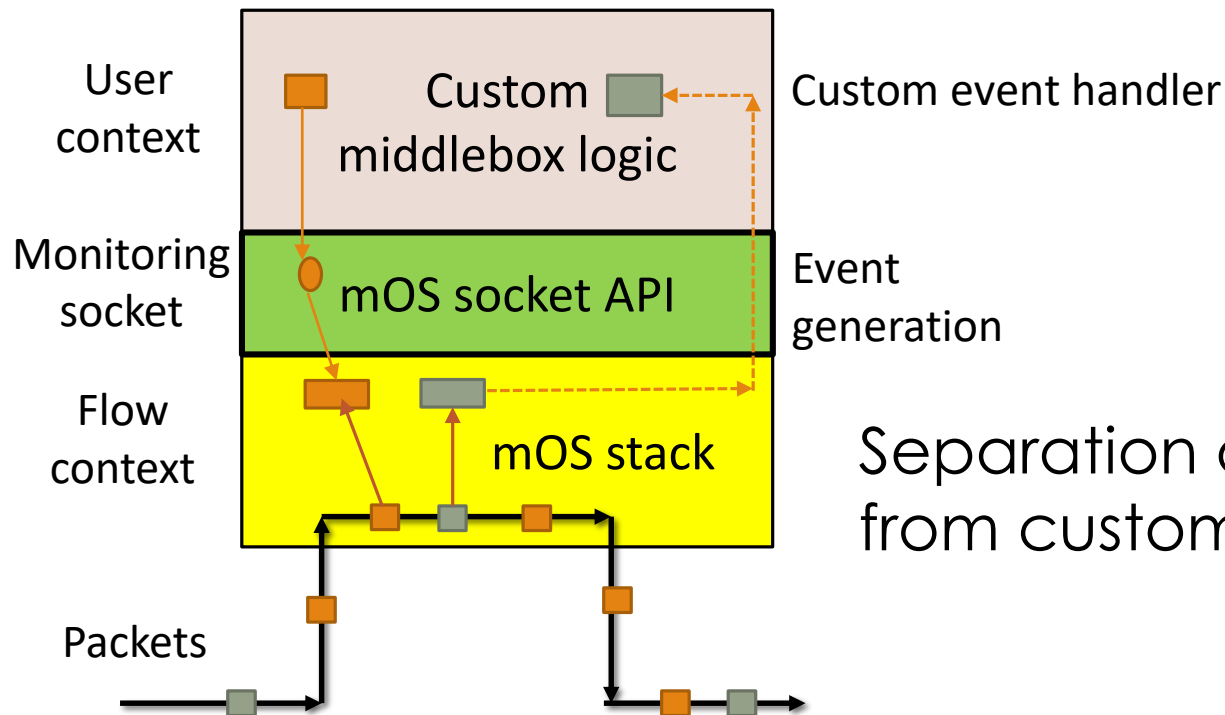
## Benefits

- Clean, modular development of stateful middleboxes
- Developers focus on core logic rather than flow management
- High performance flow management on mTCP stack

# Key Abstraction: mOS Monitoring Socket

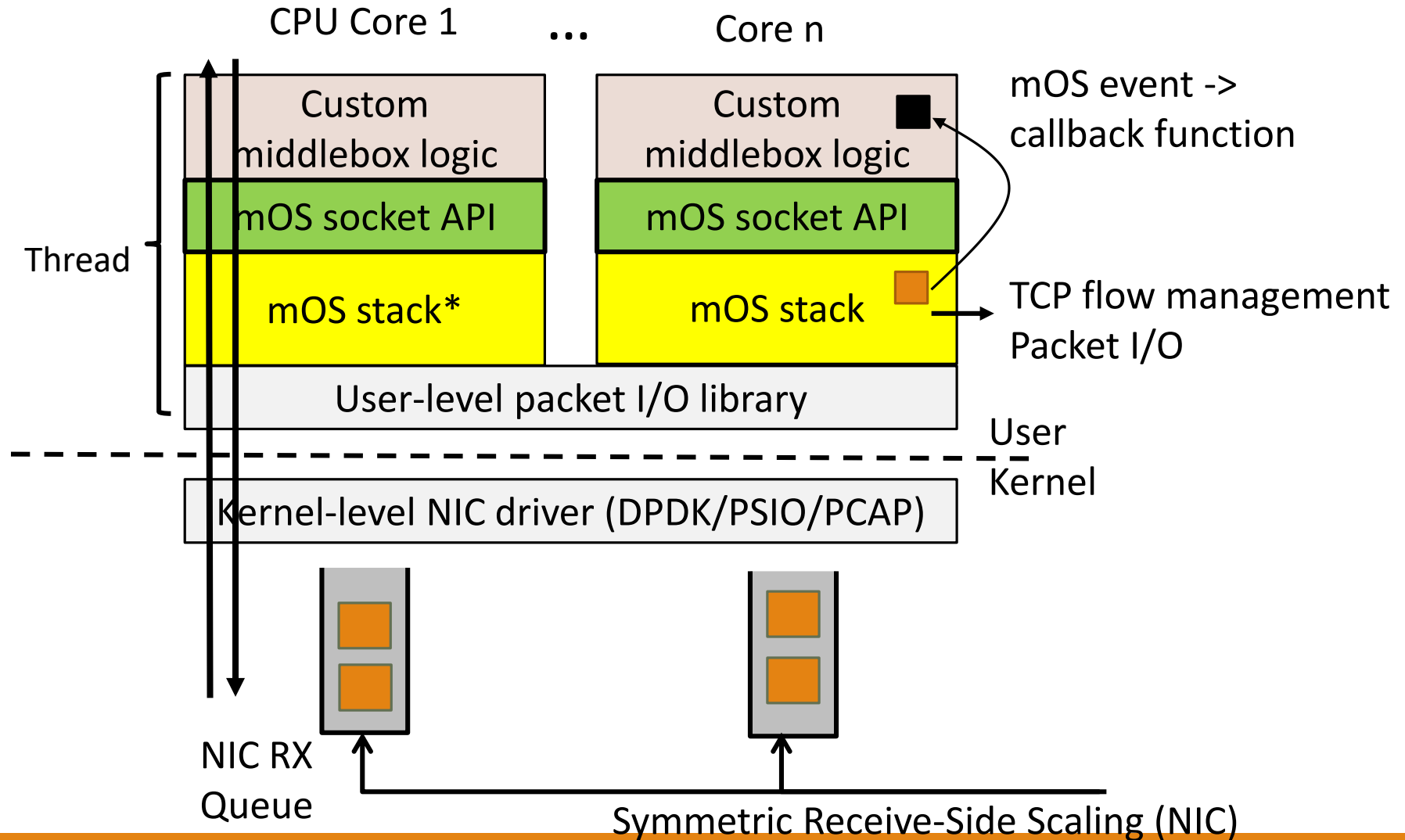
Represents the middlebox viewpoint on network traffic

- Monitors both TCP connections and IP packets
- Provides similar API to the Berkeley socket API

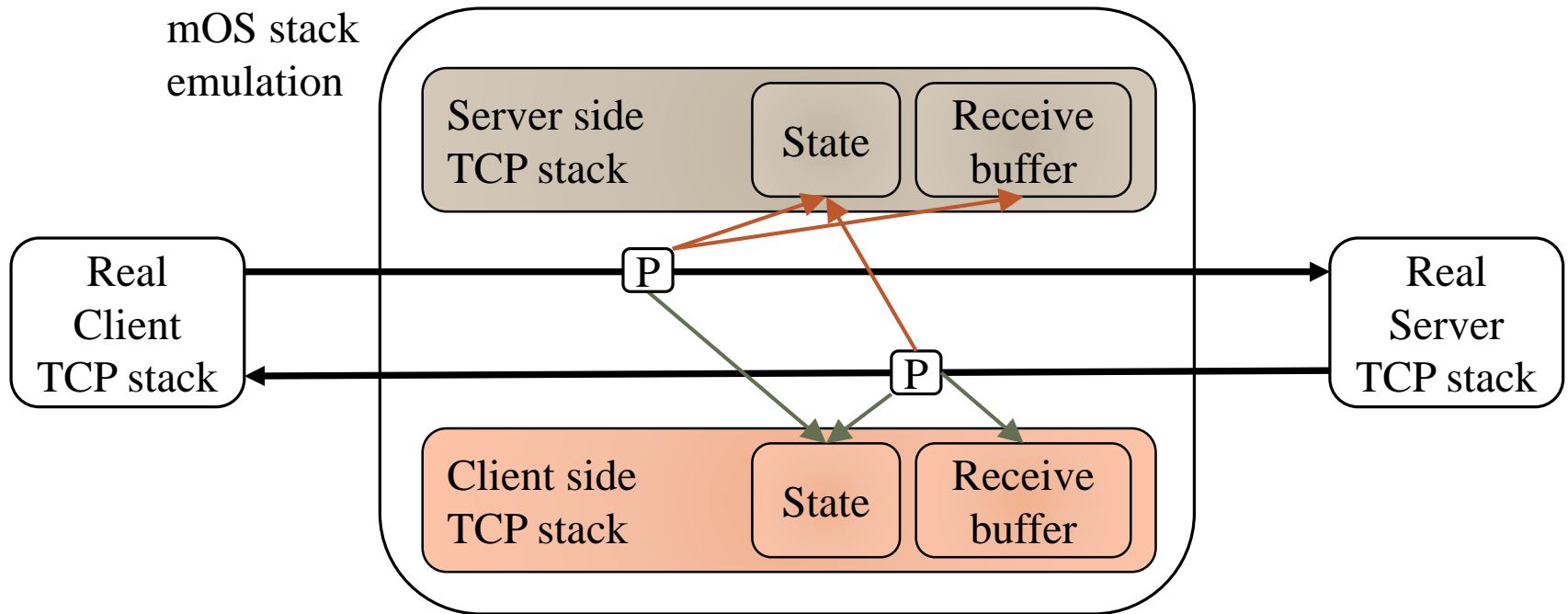


Separation of flow management from custom middlebox logic!

# Shared-Nothing Parallel Architecture



# mOS Flow Management



## **Dual** TCP stack management

- Track the TCP states of both client and server TCP stacks

Example: a client sends a SYN packet

- Client-side state changes from CLOSED to SYN\_SENT
- Server-side state changes from LISTEN to SYN\_RECEIVED

# mOS Event

---

Notable condition that merits middlebox processing

- Different from TCP socket events

## Built-in event (BE)

- Events that happen **naturally** in TCP processing
- e.g., packet arrival, connection start/teardown, retransmission, etc.

## User-defined event (UDE)

- User can define their own event
- UDE := base event + filter function
  - Raised when base event triggers and filter evaluates to TRUE
  - Nested event: base event can be either BE or UDE
  - e.g., HTTP request, 3 duplicate ACKs, malicious retransmission

Middlebox logic = a set of <event, event handler> tuples

# Sample Code: Initialization

```
static void
thread_init(mctx_t mctx)
{
    monitor_filter ft = {0};
    int msock; event_t http_event;

    msock = mtcp_socket(mctx, AF_INET, MOS_SOCKET_MONITOR_STREAM, 0);

    ft.stream_syn_filter = "dst net 216.58 and dst port 80";
    mtcp_bind_monitor_filter(mctx, msock, &ft);

    mtcp_register_callback(mctx, msock, MOS_ON_CONN_START, MOS_HK_SND, on_flow_start);

    http_event = mtcp_define_event(MOS_ON_CONN_NEW_DATA, chk_http_request);
    mtcp_register_callback(mctx, msock, http_event, MOS_HK_RCV, on_http_request);
}
```

Sets up a traffic filter in Berkeley packet filter (BPF) syntax

Defines a user-defined event that detects an HTTP request

Uses a built-in event that monitors each TCP connection start event

# UDE Filter Function

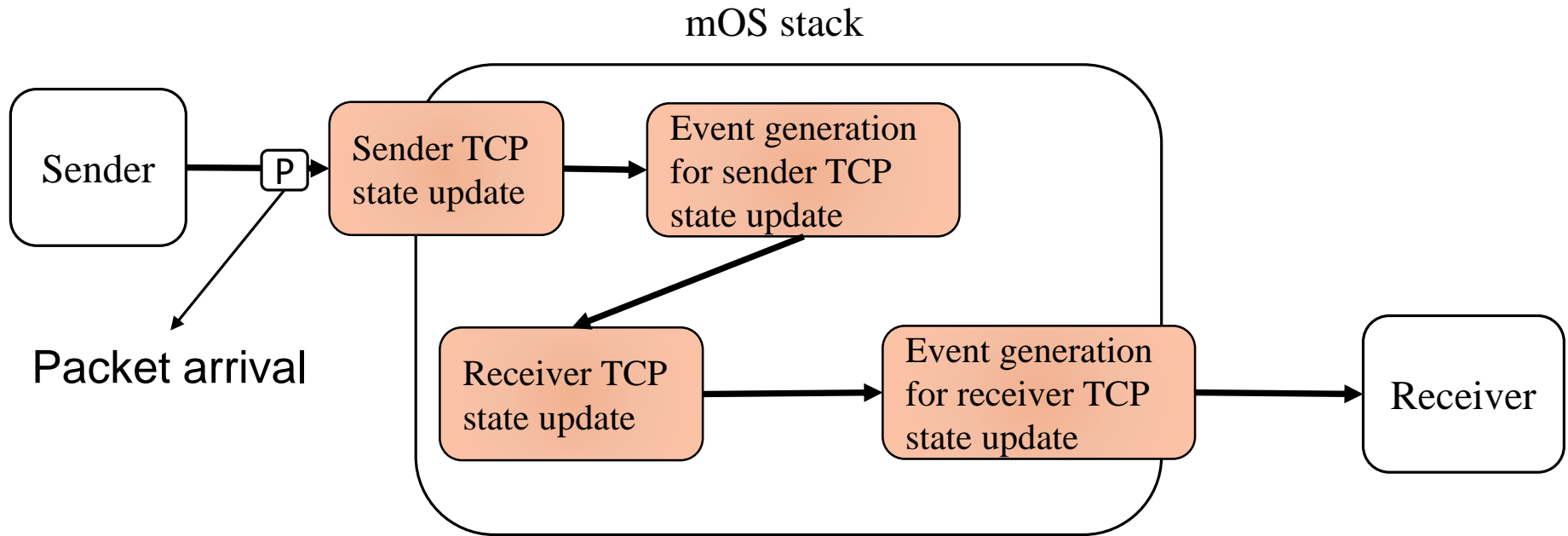
```
static bool chk_http_request(mctx_t m, int sock, int side, event_t event)
{
    struct httpbuf *p;
    u_char* temp; int r;

    if (side != MOS_SIDE_SVR) // monitor only server-side buffer
        return false;
    if ((p = mtcp_get_uctx(m, sock)) == NULL) {
        p = calloc(1, sizeof(struct httpbuf));
        mtcp_set_uctx(m, sock, p);
    }
    r = mtcp_peek(m, sock, side, p->buf + p->len, REQMAX - p->len - 1);
    p->len += r; p->buf[p->len] = 0;
    if ((temp = strstr(p->buf, "\n\n")) || (temp = strstr(p->buf, "\r\n\r\n"))) {
        p->reqlen = temp - p->buf;
        return true;
    }
    return false;
}
```

Called whenever the base event is triggered

If it returns TRUE, UDE callback function is called

# Event Generation Process



Reflects what a real middlebox sees and operates on

Based on the estimation of sender/receiver's TCP states

- E.g., packet arrival: sender's state has already been updated
- Infers the receiver stack update with a new packet

# Scalable Event Management

---

Each flow subscribes to a set of events

Each flow can change its own set of events over time

- Some flow **adds** a new event or delete an event
- Some flow **changes** the event handler for an event

Scalability problem

- How to manage event sets for 100+K concurrent flows?

Observation: the same event sets are shared by multiple flows

How to represent the event set for a flow?

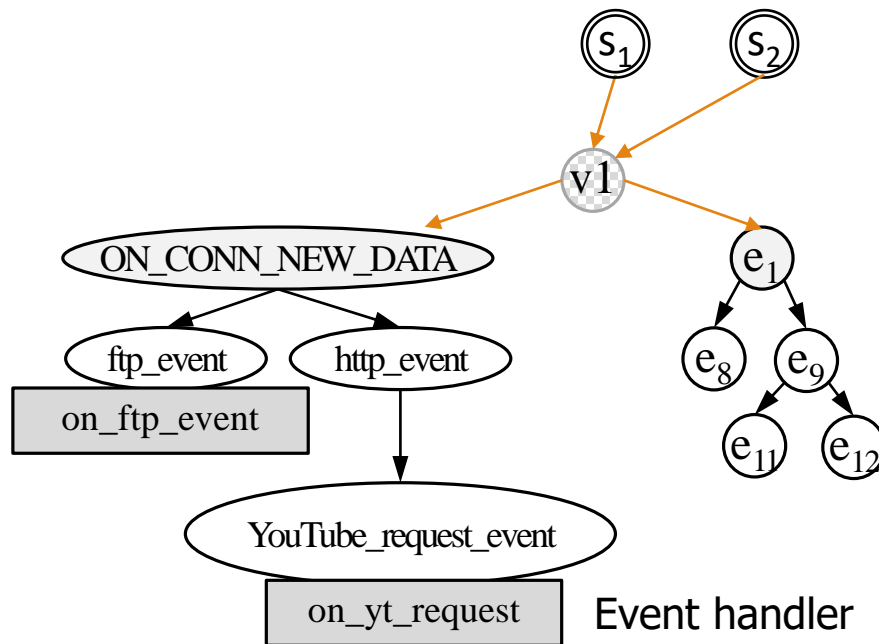
How to efficiently find the same event set?

- When a flow updates its set of events?

# Event Dependency Tree

Represents how a UDE is defined

Start from a built-in event as root

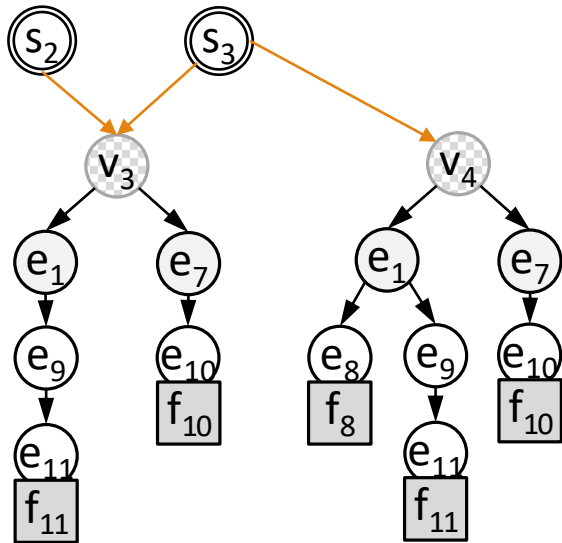


New flow

Points to a virtual root that has a set of dependency trees

Event handler

# Update on Event Dependency Tree

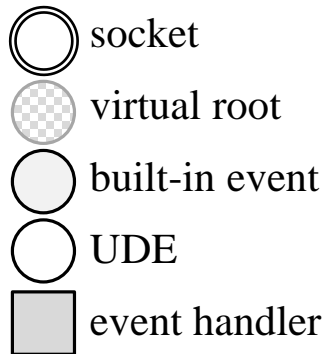


$s_3$  adds a new event  $\langle e_8, f_8 \rangle$  to  $v_3$

$v_4$  is created with a new event and  $s_3$  points to it

$s_2$  adds the same event  $\langle e_8, f_8 \rangle$  to  $v_3$

$v_4$  already exists, but how does  $s_2$  find  $v_4$ ?



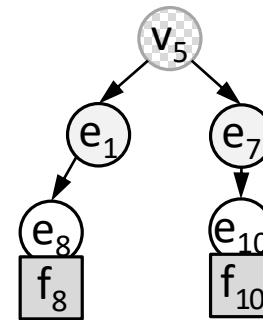
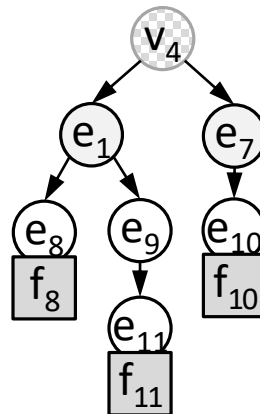
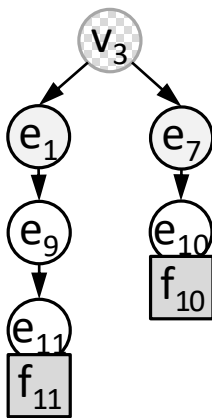
# Efficient Search for an Event Dependency Tree

Each event dependency tree has an ID

- $\text{id (virtual root)} = \text{XOR sum of hash (event + event handler)}$
- $\text{id (v3)} = \text{hash (e11 + f11)} \oplus \text{hash (e10 + f10)}$

New tree id after adding or deleting  $\langle e, f \rangle$  from  $t$

- $\text{id (t')} = \text{id (t)} \oplus \text{hash (e + f)}$
- Add  $\langle e8, f8 \rangle$  to  $v3$ ?
  - $\text{id(v4)} = \text{id(v3)} \oplus \text{hash (e8 + f8)}$
- Remove  $\langle e10, f10 \rangle$  from  $v4$ ?
  - $\text{id (v5)} = \text{id(v4)} \oplus \text{hash (e11 + f11)}$



# Fine-grained Resource Allocation

---

Not all middleboxes require full features

- Some middleboxes do not require flow reassembly
- Some middleboxes monitor only client-side data
- No more monitoring after handling certain events

Fine-control resource consumption

- Disable flow reassembly but keep only metadata
- Enable flow monitoring for one side
- Stop flow monitoring in the middle
- Per-flow manipulation with `setsockopt()`

```
// disabling receive buffers for both client and server stacks
int zero = 0;
if (!(config_monitor_side & MOS_SIDE_CLI))
    mtcp_setsockopt(mctx, sock, SOL_MONSOCKET, MOS_CLIBUF, &zero, sizeof(zero));
if (!(config_monitor_side & MOS_SIDE_SVR))
    mtcp_setsockopt(mctx, sock, SOL_MONSOCKET, MOS_SVRBUF, &zero, sizeof(zero));
```

# Current mOS stack API

---

## Socket creation and traffic filter

```
int      mtcp_socket(mctx_t mctx, int domain, int type, int protocol);  
  
int      mtcp_close(mctx_t mctx, int sock);  
  
int      mtcp_bind_monitor_filter(mctx_t mctx, int sock, monitor_filter_t ft);
```

## User-defined event management

```
event_t mtcp_define_event(event_t ev, FILTER filt);  
  
int      mtcp_register_callback(mctx_t mctx, int sock, event_t ev, int hook, CALLBACK cb);
```

## Per-flow user-level context management

```
void *   mtcp_get_uctx(mctx_t mctx, int sock);  
  
void     mtcp_set_uctx(mctx_t mctx, int sock, void *uctx);
```

## Flow data reading

```
ssize_t mtcp_peek(mctx_t mctx, int sock, int side, char *buf, size_t len);  
  
ssize_t mtcp_ppeek(mctx_t mctx, int sock, int side, char *buf, size_t count, off_t  
seq_off);
```

# Current mOS stack API

---

## Packet information retrieval and modification

```
int      mtcp_getlastpkt(mctx_t mctx, int sock, int side, struct pkt_info *pinfo);

int      mtcp_setlastpkt(mctx_t mctx, int sock, int side, off_t offset, byte *data, uint16_t
datalen, int option);
```

## Flow information retrieval and flow attribute modification

```
int      mtcp_getsockopt(mctx_t mctx, int sock, int l, int name, void *val, socklen_t *len);

int      mtcp_setsockopt(mctx_t mctx, int sock, int l, int name, void *val, socklen_t len);
```

## Retrieve end-node IP addresses

```
int      mtcp_getpeername(mctx_t mctx, int sock, struct sockaddr *addr, socklen_t *addrlen);
```

## Per-thread context management

```
mctx_t  mtcp_create_context(int cpu);

int      mtcp_destroy_context(mctx_t mctx);
```

## Initialization

```
int      mtcp_init(const char *mos_conf_fname);
```

# mOS Networking Stack Implementation

---

## Per-thread library TCP stack

- ~27K lines of C code (mTCP: ~11K lines)
- Based on mTCP user-level TCP stack [NSDI '14]
- Exploits parallelism on multicore systems and multiqueue NICs

## User-defined event implementation

- Designed to scale to arbitrary number of events
- Identical events are automatically shared by multiple flows

## Applications ported to mOS: ~9x code line reduction

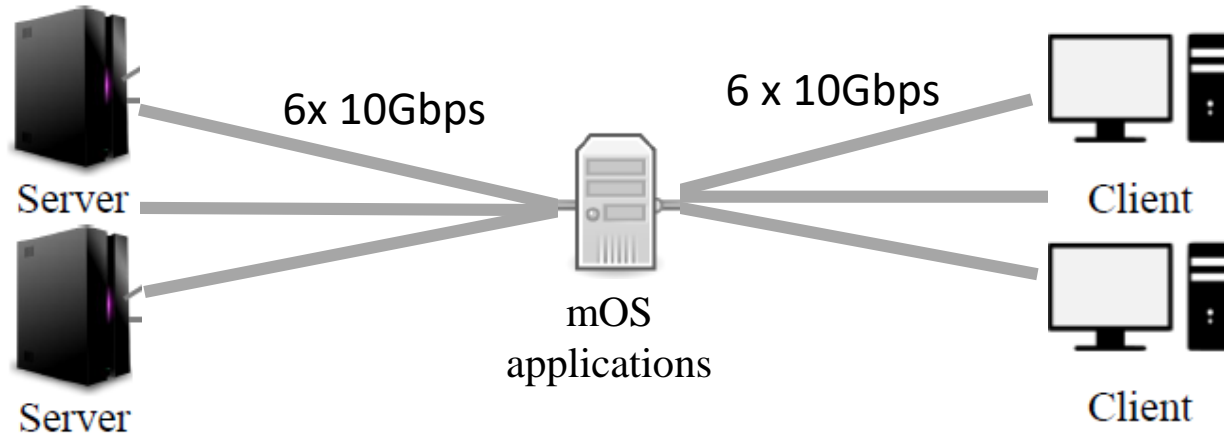
- Abacus: 4091 lines to 486 lines
- See the next page

# Real-world mOS Applications

---

- **mSnort** (modified 884 lines)
  - Snort with mOS flow management
- **mAbacus** (485 lines)
  - Secure cellular data accounting system
- **mHalfback proxy** (128 lines)
  - Low-latency proxy with proactive TCP retransmission
- **mPRADS** (modified 615 lines)
  - Pattern matching on flow-reassembled content
- **mnDPI** (modified 765 lines modification)
  - Pattern matching on flow-reassembled content
- **midstat**
  - Netstat for middlebox
- **mNAT**
  - High-performance NAT

# Evaluation: Experiment Setup



mOS applications in **inline** mode with mOS stream sockets

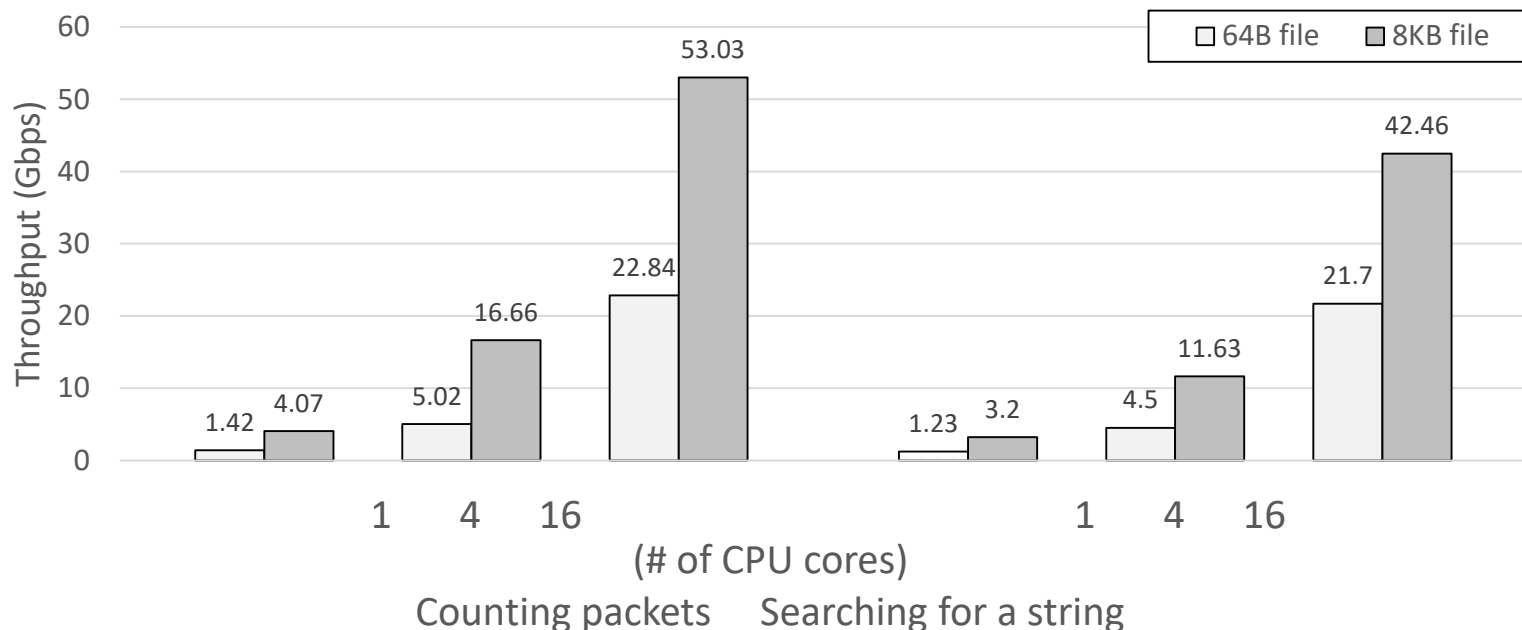
- Flow management and forwarding packets by their flows
- 2 x Intel E5-2690 (16 cores, 2.9 GHz), 20 MB L3 cache size,
- 132 GB RAM, 6 x 10 Gbps NICs

Six pairs of clients and servers: 60 Gbps max

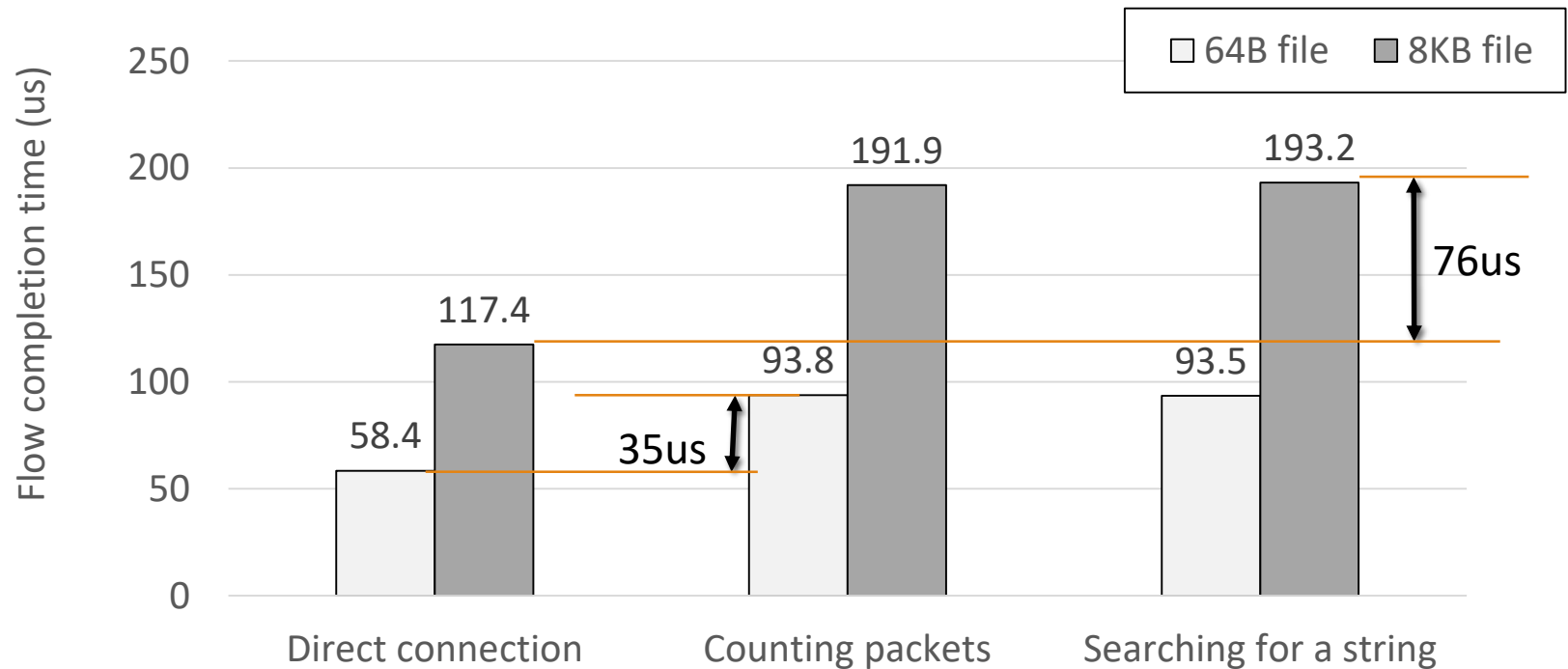
- Intel E3-1220 v3 (4 cores, 3.1 GHz), 8 MB L3 cache size
- 16 GB RAM, 1 x 10 Gbps NIC per machine

# Performance Scalability on Multicores

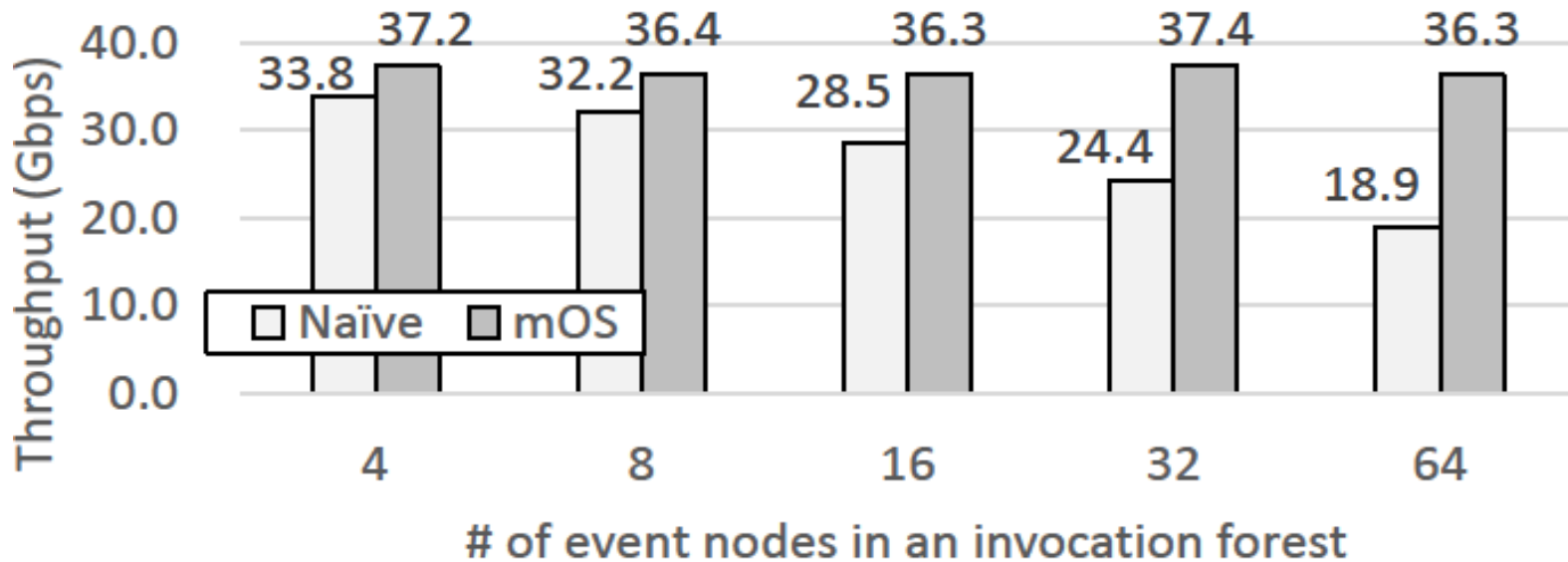
- File download traffic with 192,000 concurrent flows
  - Each flow downloads an X-byte content in one TCP connection
  - A new flow is spawned when a flow terminates
- Two simple applications
  - Counting packets per flow (packet arrival event)
  - Searching for a string in flow reassembled data (full flow reassembly & DPI)



# Latency Overhead by mOS Applications



# Event Management Performance



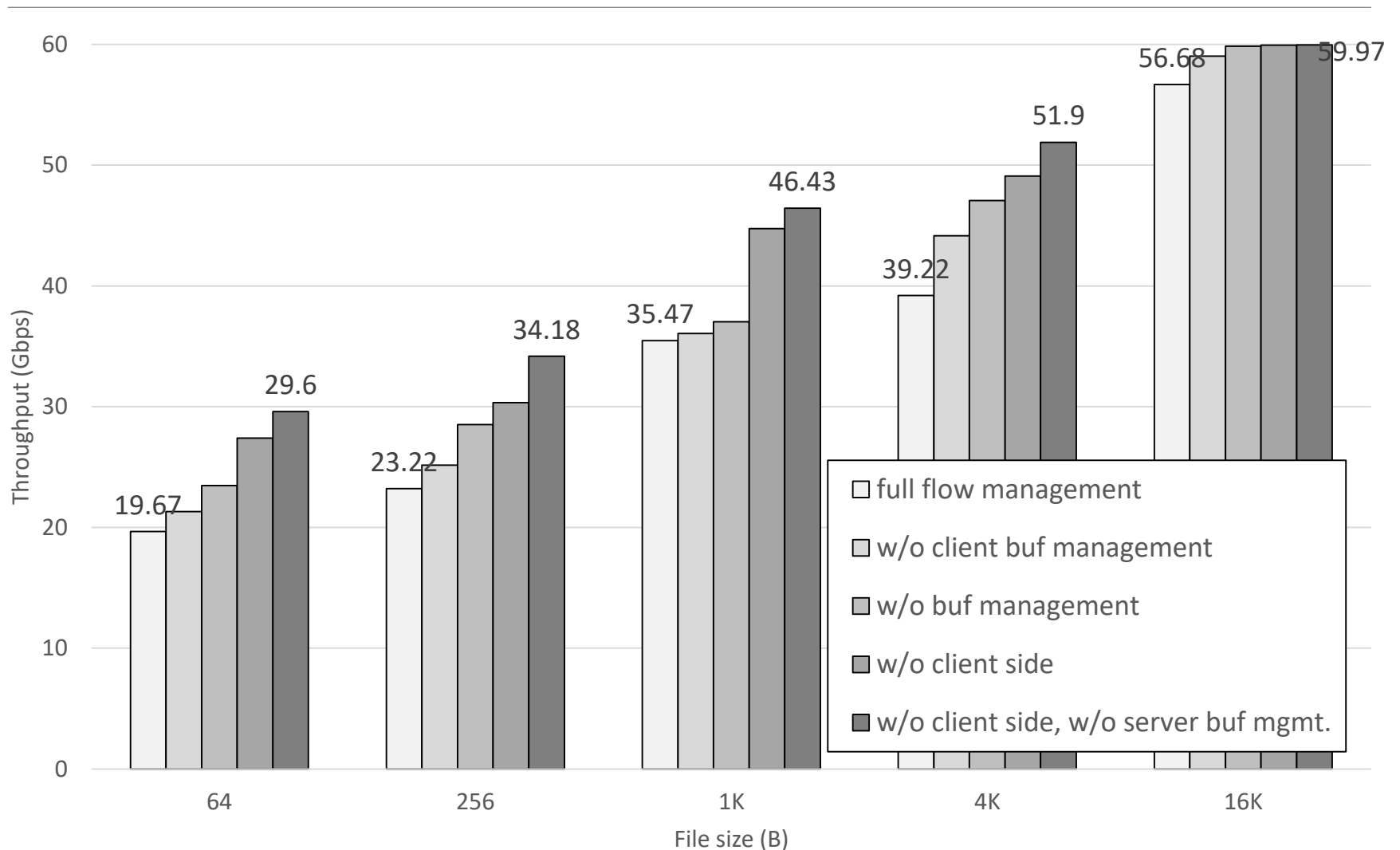
192,000 concurrent flows downloading large files

mOS application searches for a string, dynamically adds a new event

Increases the number of events per flow (4 to 64)

mOS improves the performance by 3.5 to 17.3 Gbps

# Performance Under Selective Resource Consumption



# Real Application Performance

---

Application	original + pcap	original + DPDK	mOS port
Snort-AC	0.57 Gbps	8.18 Gbps	9.17 Gbps
Snort-DFC	0.82 Gbps	14.42 Gbps	15.21 Gbps
nDPIReader	0.66 Gbps	28.92 Gbps	28.87 Gbps
PRADS	0.42 Gbps	2.03 Gbps	1.90 Gbps

- Workload: real LTE packet trace (~67 GB)
- 4.5x ~ 28.9x performance improvement
- Mostly due to multi-core aware packet processing (DPDK)
- mOS brings code modularity and correct flow management

# Conclusion

---

Current middlebox development suffers from

- Lack of modularity
- Lack of readability
- Lack of maintainability

Solution: reusable networking stack for middleboxes

mOS stack: abstraction for flow management

- Programming abstraction with socket-based API
- Event-driven middlebox processing
- Efficient resource usage with dynamic resource composition

# mOS Stack Is Open-Sourced

Public source code release of mOS stack/API at github

- <https://github.com/ndsl-kaist/mOS-networking-stack>

mOS online manual

- <http://mos.kaist.edu/guide/>



---

Thank you!

mOS project page

<http://mos.kaist.edu/>