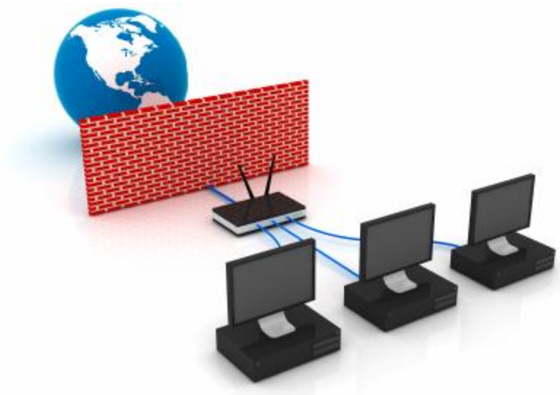
A Networking Stack for Modular Middlebox Development

Middlebox OS (mOS) Development Team

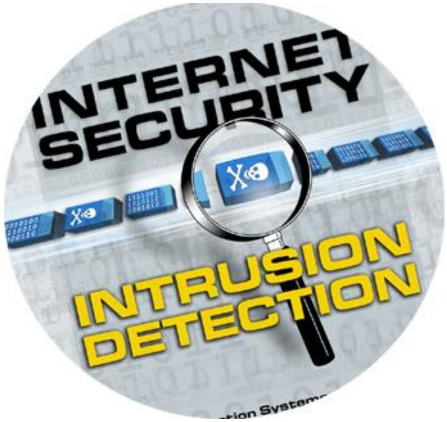
Asim Jamshed, Donghwi Kim, YoungGyoun Moon
Dongsu Han, KyoungSoo Park

Department of Electrical Engineering, KAIST

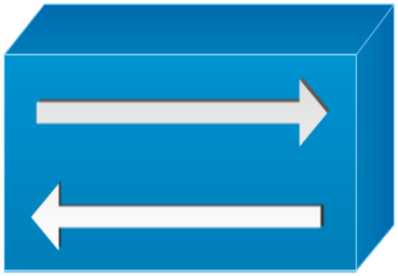
Network Middleboxes



Firewall

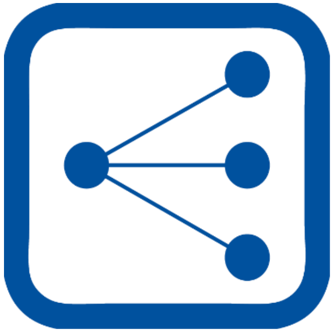


IDS/IPS



NAT

Web/SSL Proxy



Load Balancer

Middleboxes are Increasingly Popular

- Middleboxes are ubiquitous
  - # of middleboxes ~= # of routers [NSDI'12] (Enterprise)
  - Prevalent in cellular networks [SIGCOMM'11]

- They provide key functionalities in modern networks
  - Original Internet design lacks many such features

Toward Software Middleboxes

- Hardware middleboxes
  - Expensive
  - Proprietary
  - Hard to deploy new services

- Software middleboxes
  - Cost-effective
  - Accelerate time-to-market
  - Flexible
  - Many open-source projects

carrier-grade NAT

IDS

firewall

Network Address Translation

SNORT

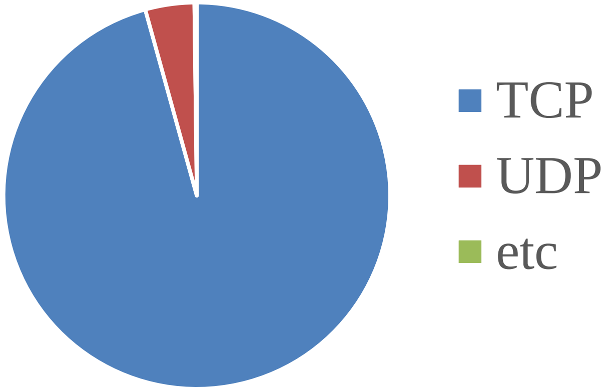
Iptables

Commodity x86 servers (white box)

Stateful Middleboxes Dominate the Internet

- In this talk, state = TCP/Application state
- 95+% of the Internet traffic is TCP [1]
- Most middleboxes deal with TCP traffic
  - Stateful firewalls
  - Protocol analyzers
  - Cellular data accounting
  - Intrusion detection/prevention systems
  - Network address translation

  …

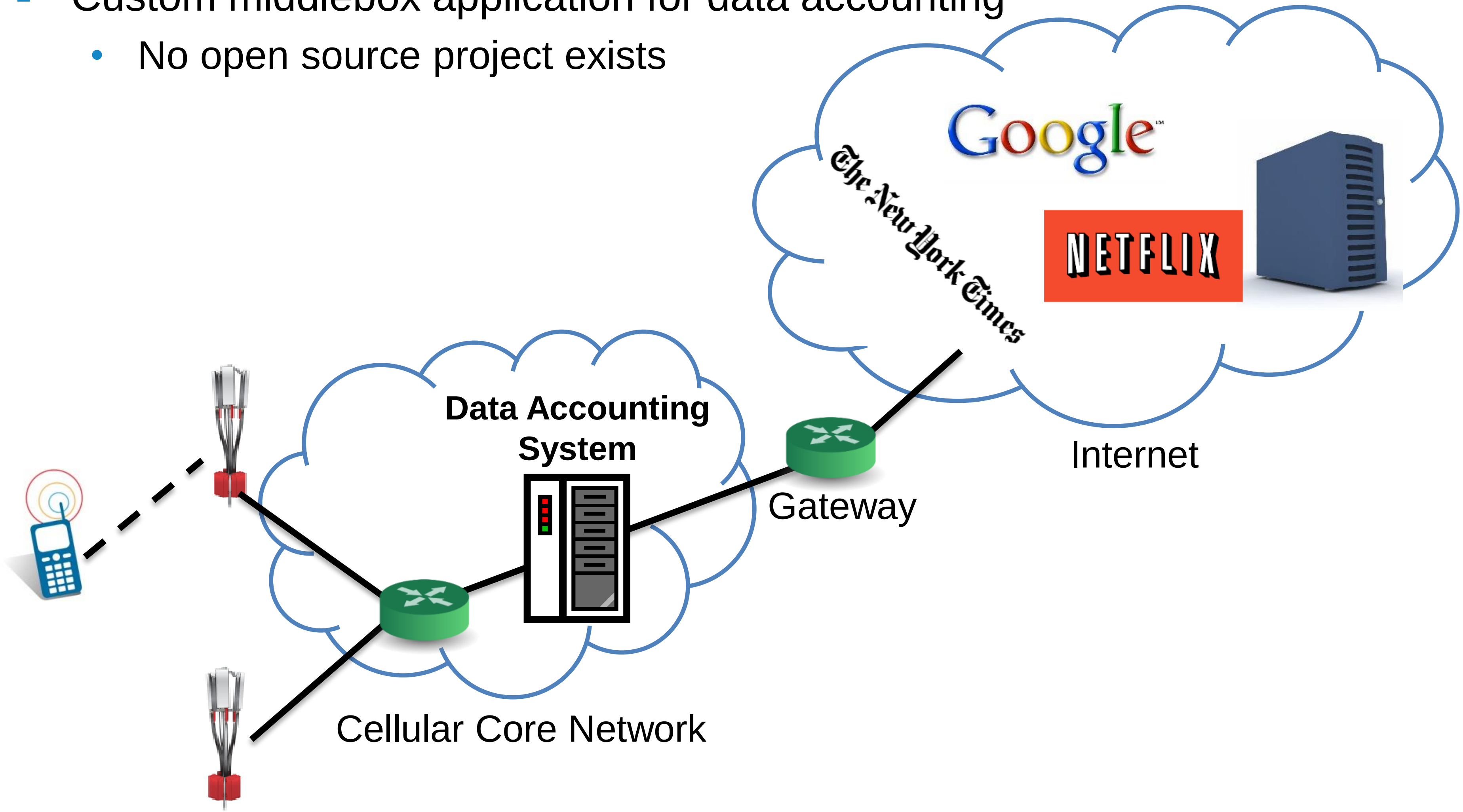


[1] Comparison of Caching Strategies in Modern Cellular Backhaul Networks, MobiSys 2013.

State management is complex and error-prone

Example: Cellular Data Accounting System

- Custom middlebox application for data accounting
  - No open source project exists



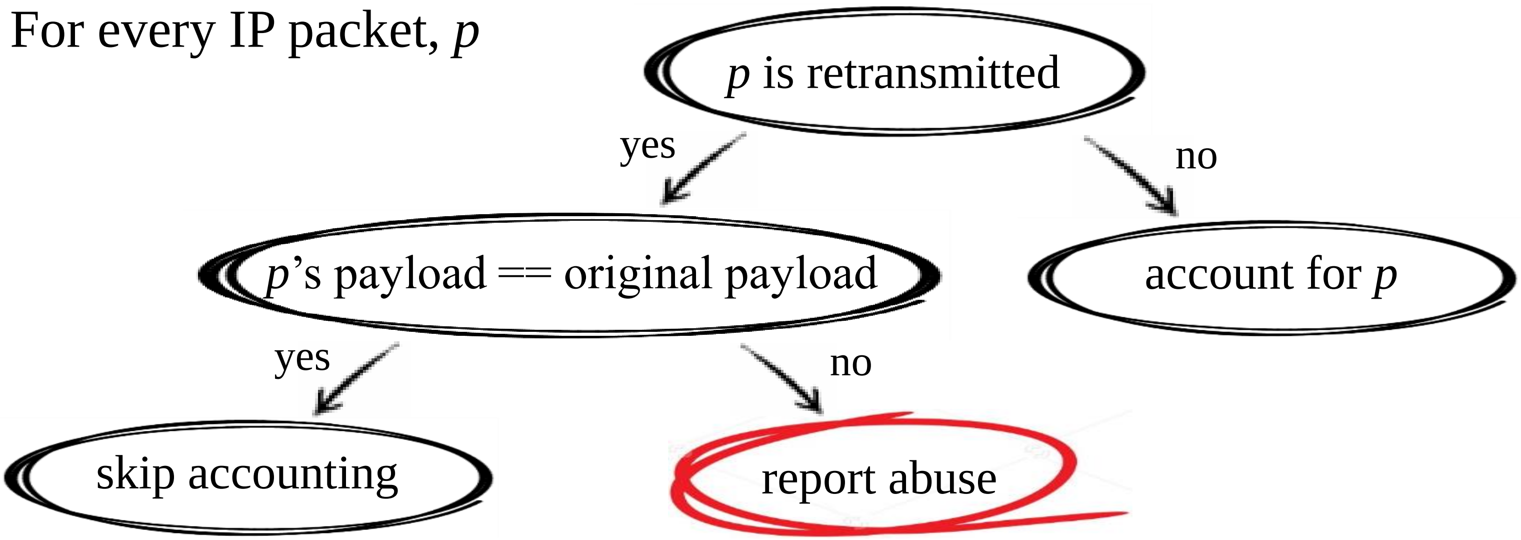
Develop a Cellular Data Accounting System

Requirements

1. Follow accounting policy in South Korea
   - "Selective" accounting does not charge for TCP retransmission packets
2. Detect TCP tunneling attack [NDSS '14]

seq# = 10	payload A		seq# = 10	payload B

For every IP packet, *p*



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)	• 50~100K code lines tightly coupled with their IDS logic
Borrow code from open-source kernel (e.g., Linux/FreeBSD)	• Designed for TCP end host • Different from middlebox semantics
Implement your own flow management	• ***Repeat*** it for every custom middlebox

Programming TCP Application

- Typical TCP applications

TCP application	User level
Berkeley Socket API	
TCP/IP stack	Kernel level

- Typical middleboxes?

  - Middlebox logic
  - Packet processing
  - Flow management
  - Spaghetti code?

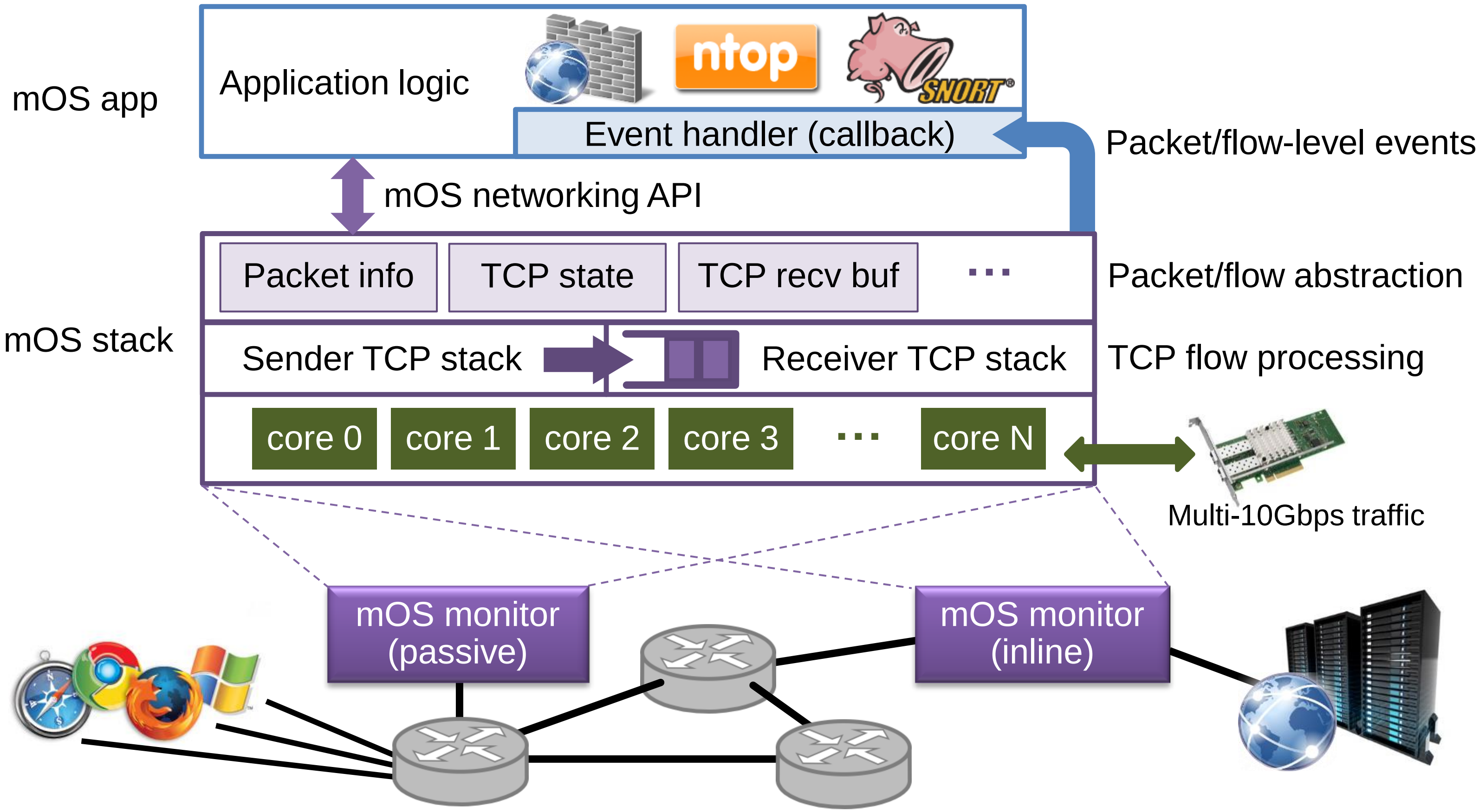
  No clear separation!

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

mOS Networking Stack

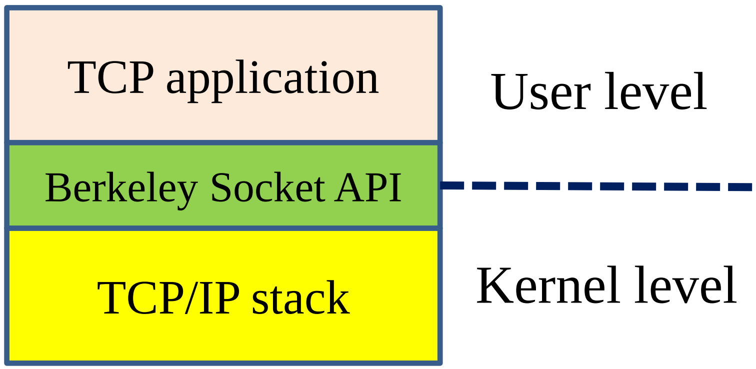
- Networking stack specialization for middleboxes
  - Abstraction for sub-TCP layer middlebox operations
- Key concepts
  - Separation of flow management from custom logic
  - Event-driven middlebox processing
  - Per-flow resource provisioning
- Benefits
  - Clean, modular development of stateful middleboxes
  - Developers focus on core logic rather than flow management
  - High performance flow management based on multi-core scalability

Operation Scenarios of mOS Applications

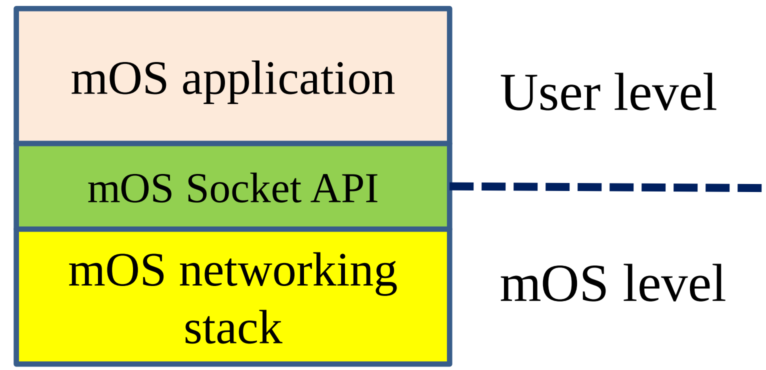


Programming Middlebox Application

- Typical TCP applications



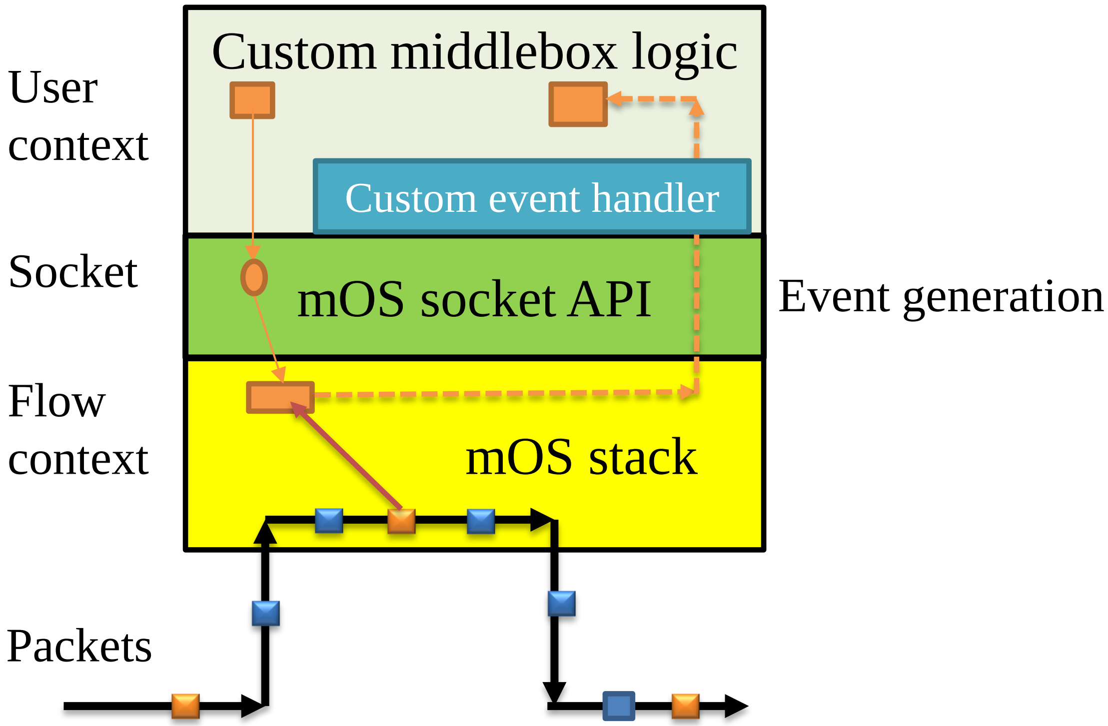
- mOS middlebox applications



- mOS socket API
  - Inspired by Berkeley socket API
  - Separates flow management from middlebox core logic
  - ***Never*** requires you to write flow management logic itself

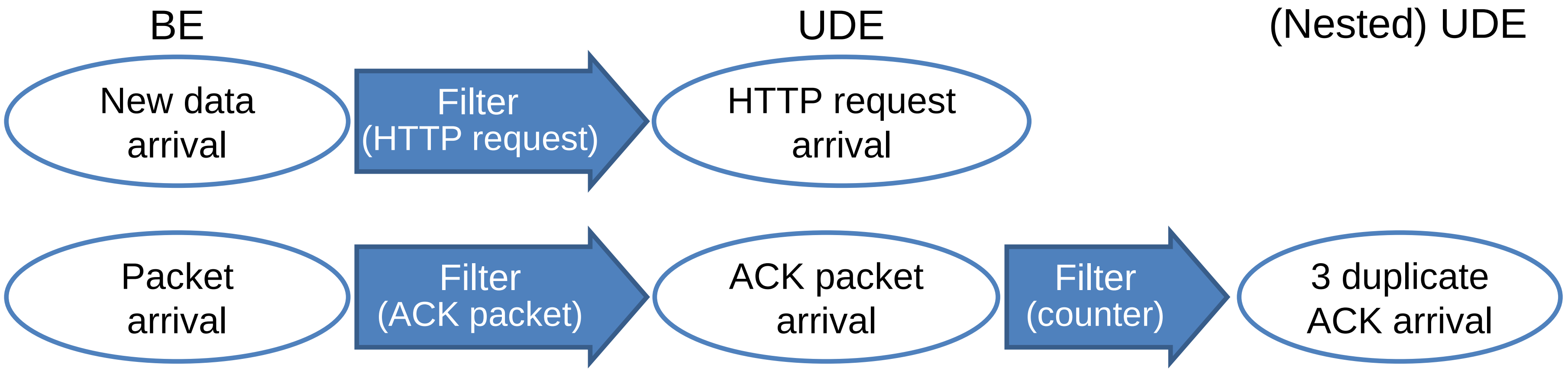
mOS Monitoring Socket Abstraction

- Stream monitoring socket
  - Abstraction for monitoring TCP connection
- Raw monitoring socket
  - Abstraction for monitoring IP packets



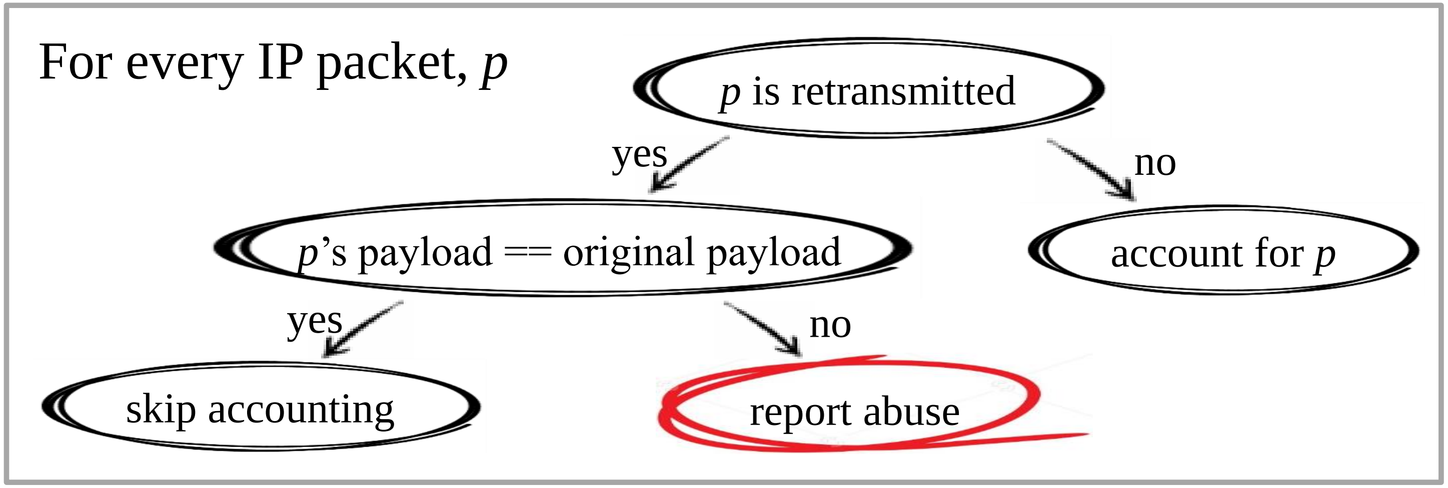
mOS Event

- Middlebox logic = a set of <event, event handler> tuples
- Built-in event (BE)
  - Events that happen naturally in TCP processing
    - e.g., packet arrival, TCP connection start/teardown, retransmission, etc.
- User-defined event (UDE)
  - User can define their own event (= base event + filter function)

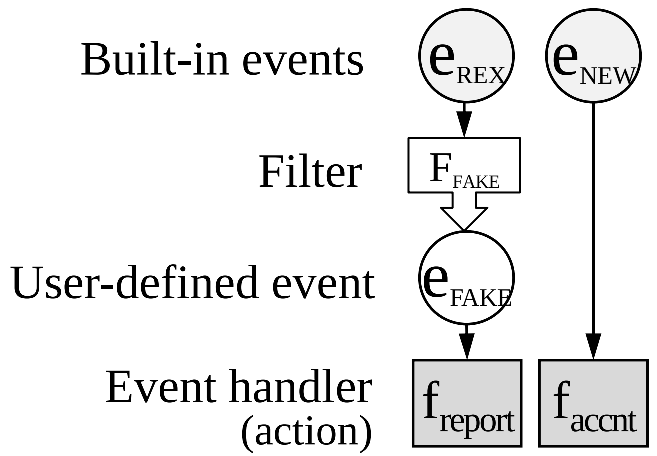


Cellular Data Accounting System with mOS

Core Logic



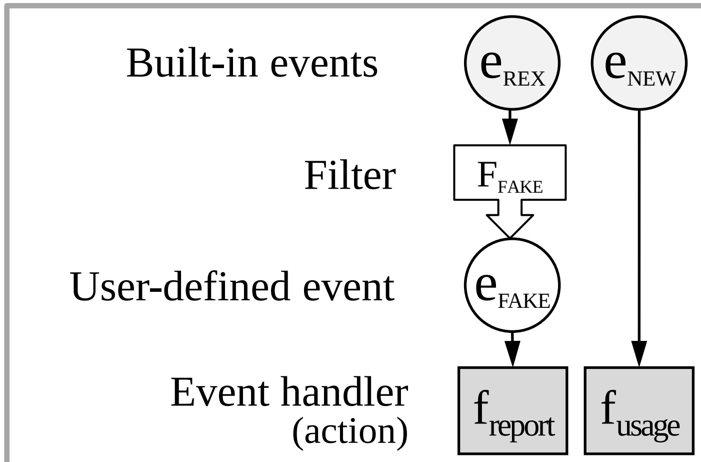
Event-action



e_{REX}	MOS_ON_REXMIT
e_{NEW}	MOS_ON_CONN_NEW_DATA
F_{FAKE}	IsFakeRexmit()
e_{FAKE}	UDE_FAKE_REXMIT
f_{report}	ReportAbuse()
f_{accnt}	AccountDataUsage()

Cellular Data Accounting System with mOS

Event-action



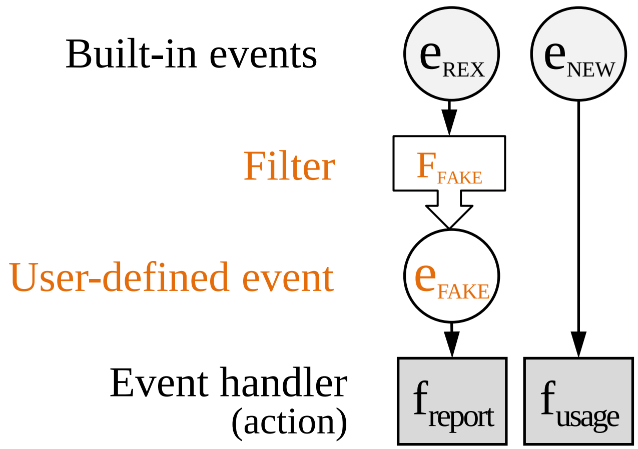
e_{REX}	MOS_ON_REXMIT
e_{NEW}	MOS_ON_CONN_NEW_DATA
F_{FAKE}	IsFakeRexmit()
e_{FAKE}	UDE_FAKE_REXMIT
f_{report}	ReportAbuse()
f_{usage}	AccountDataUsage()

Implementation

```
static void
thread_init(mctx_t mctx)
{
  int msock; event_t fake_rexmit_ev;

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

}
```

Cellular Data Accounting System with mOS

Event-action



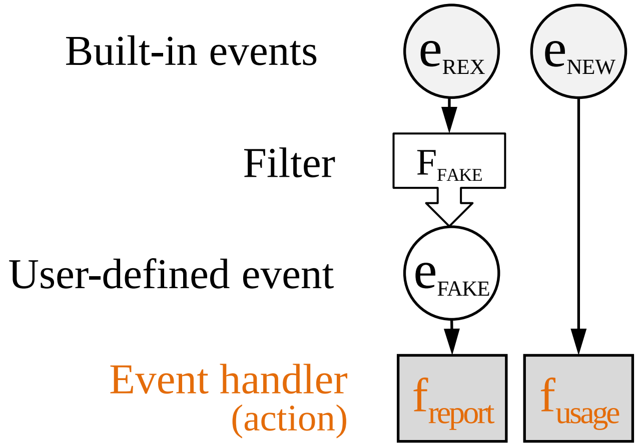
e_{REX}	MOS_ON_REXMIT
e_{NEW}	MOS_ON_CONN_NEW_DATA
F_{FAKE}	IsFakeRexmit()
e_{FAKE}	UDE_FAKE_REXMIT
f_{report}	ReportAbuse()
f_{usage}	AccountDataUsage()

Implementation

```
static void
thread_init(mctx_t mctx)
{
  int msock; event_t fake_rexmit_ev;

  msock = mtcp_socket(mctx, AF_INET, MOS_SOCK_MONITOR_STREAM, 0);
  fake_rexmit_ev = mtcp_define_event(MOS_ON_REXMIT, IsFakeRexmit);



}
```

Cellular Data Accounting System with mOS

Event-action



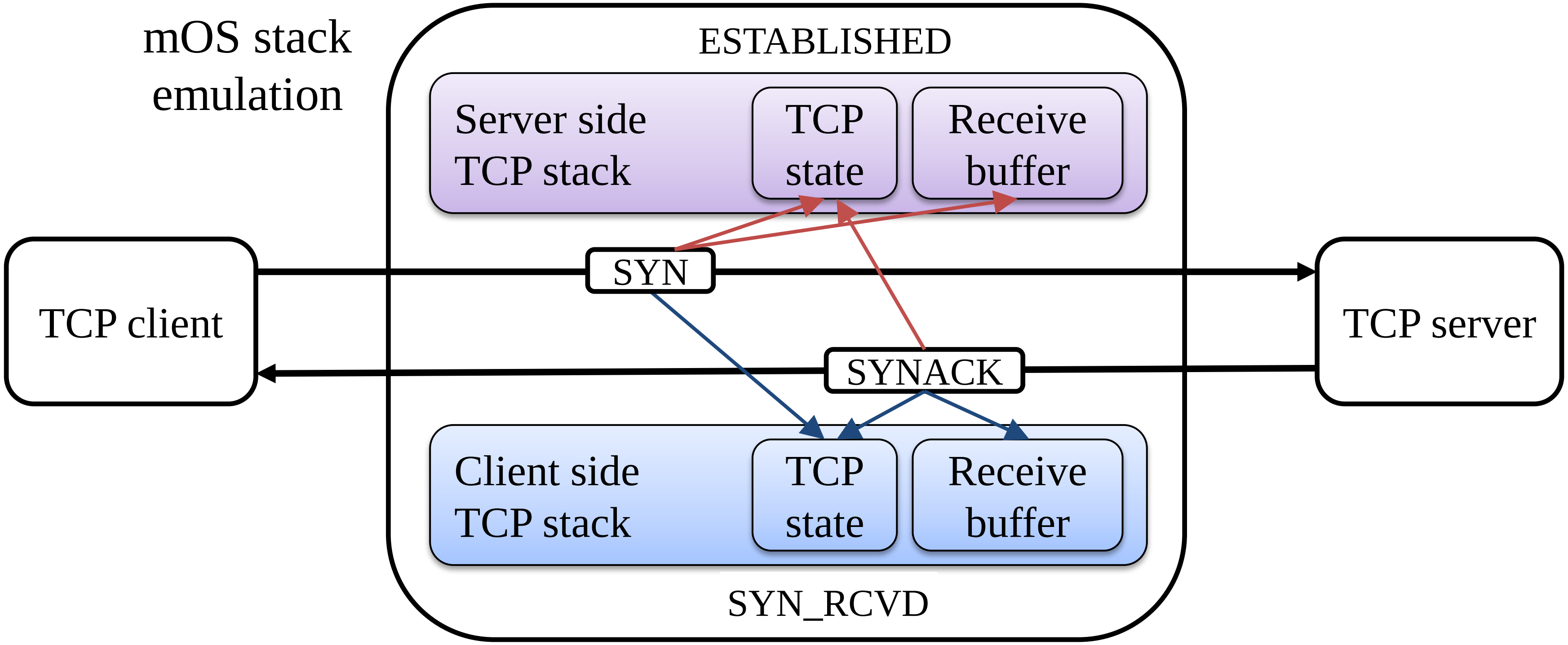
e_{REX}	MOS_ON_REXMIT
e_{NEW}	MOS_ON_CONN_NEW_DATA
F_{FAKE}	IsFakeRexmit()
e_{FAKE}	UDE_FAKE_REXMIT
f_{report}	ReportAbuse()
f_{usage}	AccountDataUsage()

Implementation

```
static void
thread_init(mctx_t mctx)
{
  int msock; event_t fake_rexmit_ev;

  msock = mtcp_socket(mctx, AF_INET, MOS_SOCK_MONITOR_STREAM, 0);
  fake_rexmit_ev = mtcp_define_event(MOS_ON_REXMIT, IsFakeRexmit);
  mtcp_register_callback(mctx, msock, fake_rexmit_ev, MOS_HK_SND,
                         ReportAbuse);
  mtcp_register_callback(mctx, msock, MOS_ON_CONN_NEW_DATA, MOS_HK_SND,
                         AccountDataUsage);
}
```

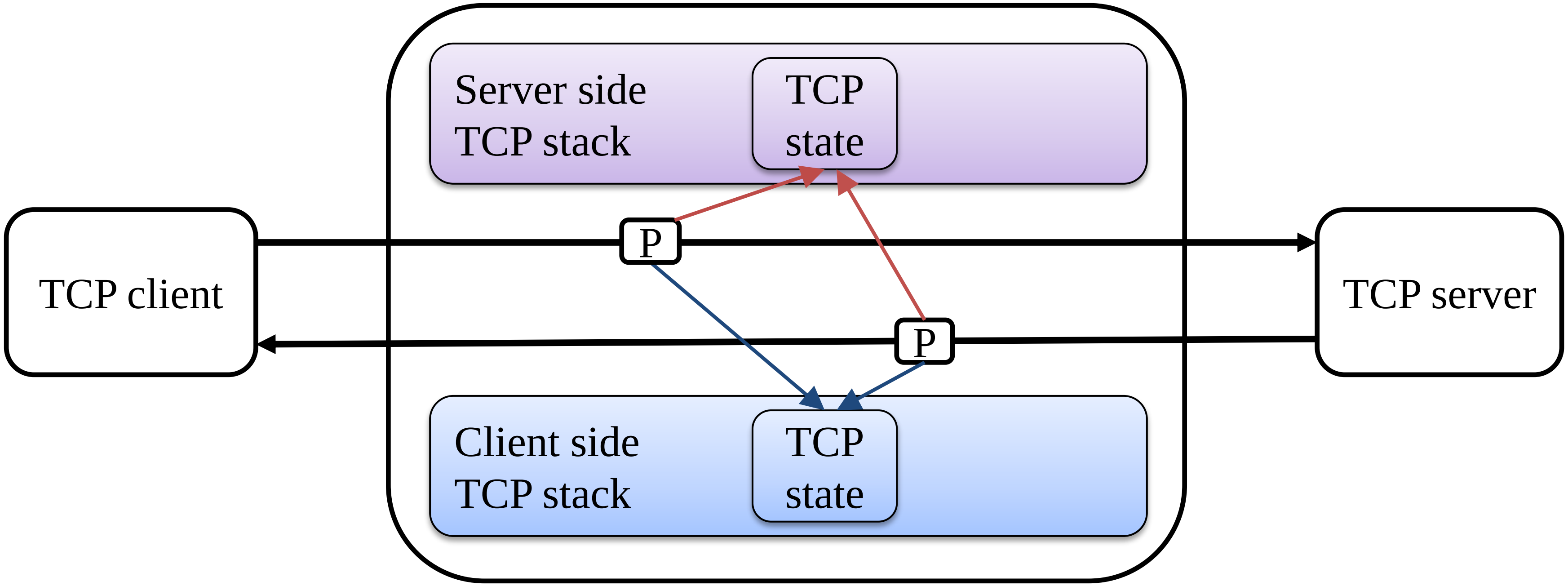
mOS Flow Management (Inline Mode)

- Dual TCP stack management
  - ***Infer*** the states of both client and server TCP stacks



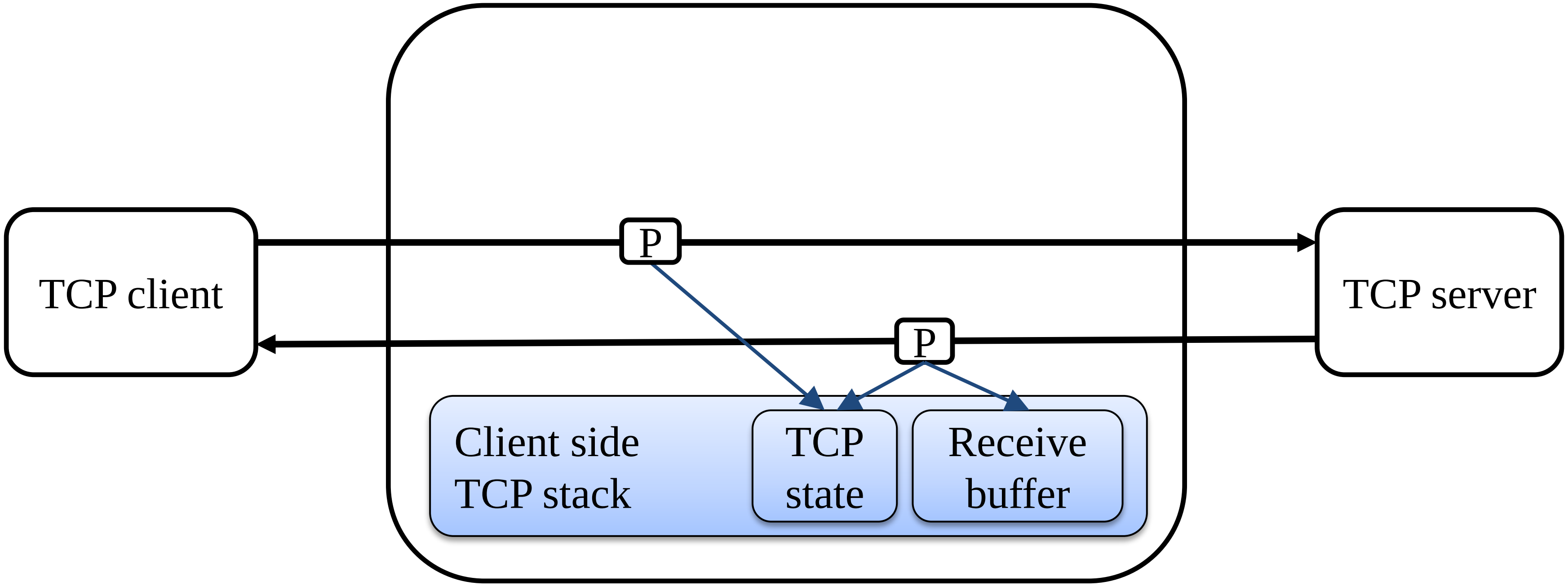
Fine-grained Resource Allocation

- Not all middleboxes require full features
  - Some middleboxes do not require flow reassembly



Fine-grained Resource Allocation

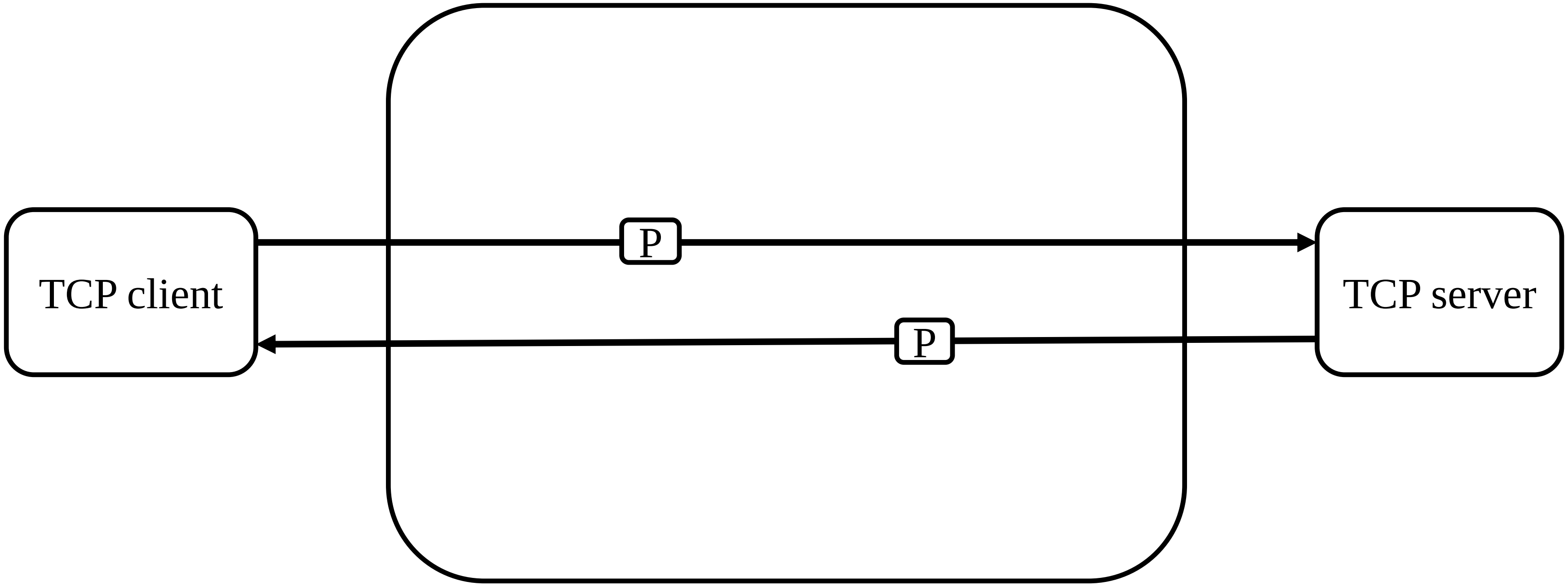
- Not all middleboxes require full features
  - Some middleboxes do not require flow reassembly
  - Some middleboxes monitor only client-side data



Fine-grained Resource Allocation

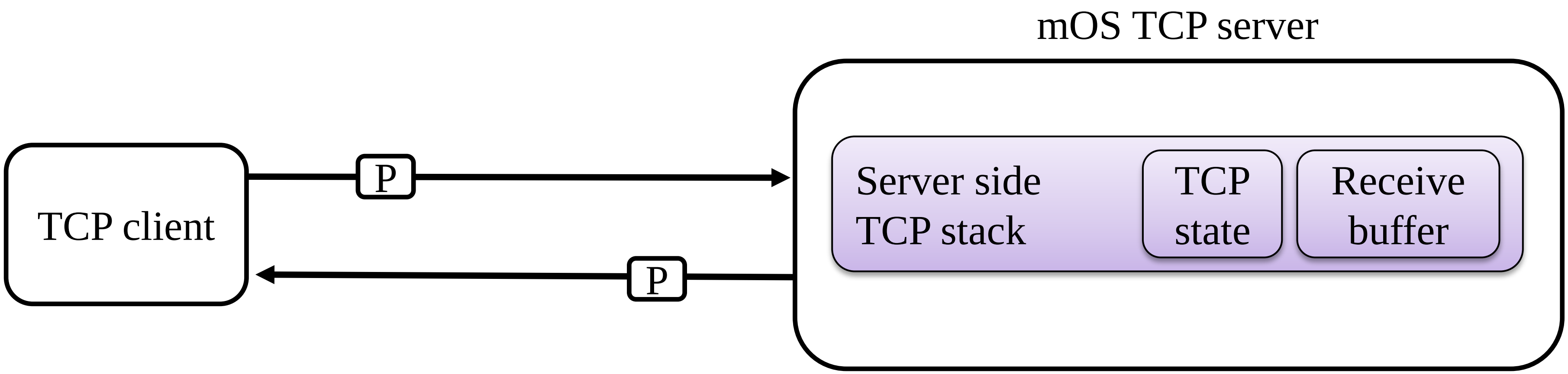
- Not all middleboxes require full features
  - Some middleboxes do not require flow reassembly
  - Some middleboxes monitor only client-side data
  - No more monitoring after handling certain events

Global or per-flow manipulation



mOS Networking Stack for End Hosts

- mOS networking stack also provides end host socket APIs
  - mOS networking stack can be configured as a end TCP stack



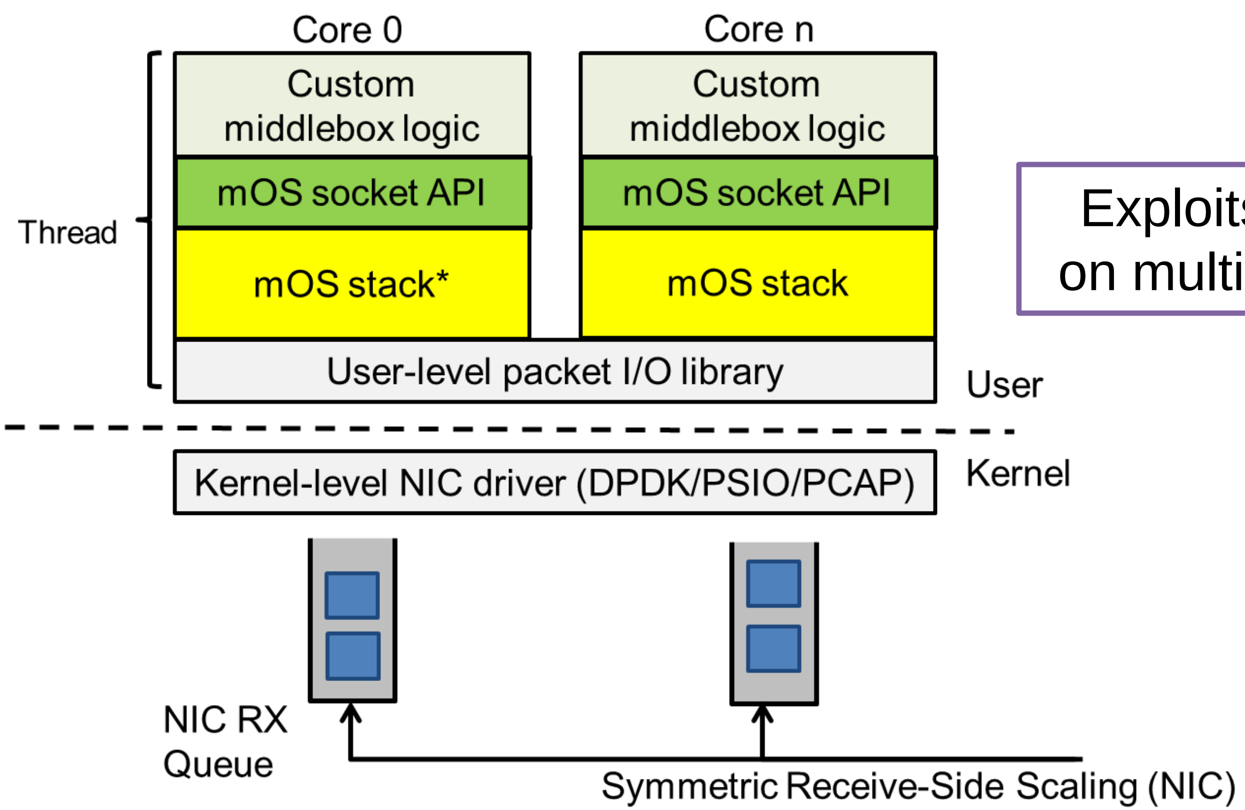
Current mOS stack API

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);
```
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);
```

17 functions are currently defined

mOS Networking Stack Implementation

- Per-thread library TCP stack
  - ~26K lines of C code (mTCP: ~11K lines)
  - Based on mTCP user level TCP stack [NSDI '14]



Exploits parallelism on multicore systems

mOS Networking Stack Implementation

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

- Applications ported to mOS: ~8x code line reduction

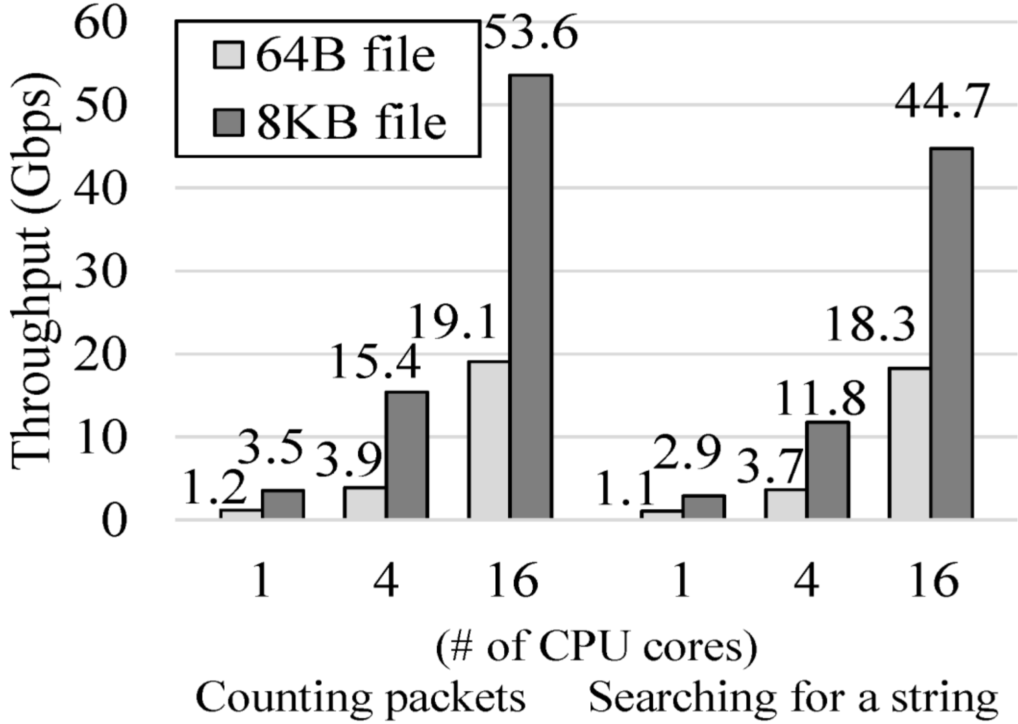
Application	Modified	SLOC	Output
Snort	2,104	79,889	HTTP/TCP inspection
nDPI	765	25,483	Stateful session management
PRADS	615	10,848	Stateful session management
Abacus	-	4,639 → 561	Detect out-of-order packet retransmission

Evaluation: Experiment Setup

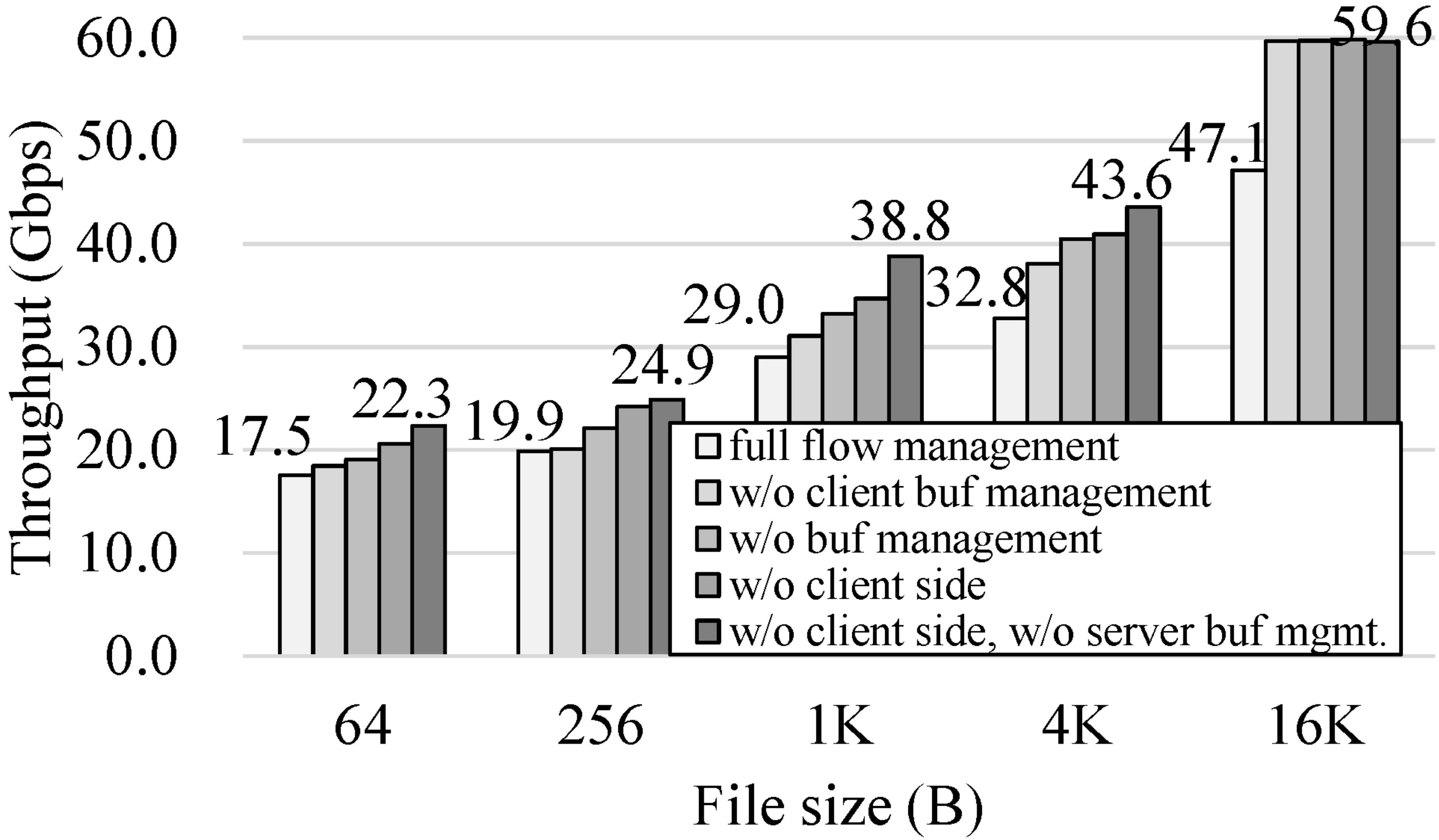
- Operating as **in-line** mode: clients ⇔ mOS applications ⇔ servers
- mOS applications with mOS stream sockets
  - Flow management and forwarding packets by their flows
  - 2 x Intel E5-2690 (16 cores, 2.9 GHz)
  - 20 MB L3 cache size, 132 GB RAM
  - 6 x 10 Gbps NICs
- Six pairs of clients and servers: 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 over # of CPU cores

- Concurrent number of flows: 192,000
  - Each flow downloads an 64B or 8KB 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)



Performance Under Selective Resource Consumption



Real Application Performance

- Workload: real LTE packet trace (~ 67 GB)

Application	original + pcap	original + DPDK	mOS port
Snort-AC	0.51 Gbps	8.43 Gbps	9.85 Gbps
Snort-DFC	0.78 Gbps	10.43 Gbps	12.51 Gbps
nDPIReader	0.66 Gbps	29.42 Gbps	28.34 Gbps
PRADS	0.42 Gbps	2.05 Gbps	2.02 Gbps

- 4.5x ~ 28.9x performance improvement
  - Mostly due to multi-core scalable packet processing (DPDK)
- mOS additionally brings code modularity and correct flow management

Conclusion

- Current middlebox development suffers from
  - Lack of modularity
  - Lack of readability
  - Lack of maintainability
- Key idea: reusable, common flow management 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

Thank You!

- mOS homepage: http://mos.kaist.edu/
  - Source code and guides are now available!
- Questions?