mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes

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

KAIST EE
Most Middleboxes Deal with TCP Traffic

- TCP dominates the Internet
  - 95+% of traffic is TCP [1]

- Top 3 middleboxes in service providers rely on L4/L7 semantics

Virtual Appliances Deployed in Service Provider Data Centers [2]

<table>
<thead>
<tr>
<th>Service</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web application firewall</td>
<td>67%</td>
</tr>
<tr>
<td>Mail security gateway</td>
<td>63%</td>
</tr>
<tr>
<td>Web security gateway</td>
<td>47%</td>
</tr>
</tbody>
</table>

Example: Cellular Accounting System

- Custom middlebox application
  - No open source solution
Challenges in Building Flow-level Middleboxes

- The main logic for a cellular accounting system
  - No charge for TCP retransmission, only if payloads match.

For every IP packet, \( p \):

- \( p \)'s payload == original payload
  - Yes: skip accounting
  - No: TCP tunneling attack!

- \( p \) is retransmitted
  - Yes: charge for \( p \)
  - No: \( p \) is not retransmitted

Core logic itself is straightforward!
Challenges in Building Flow-level Middleboxes

- Requires handling complex flow-level states and events

- The accounting system requires:
  - Reassembly buffer that holds the original payload
  - Non-contiguous fragments that holds the original payload
  - Event notification on TCP retransmission
  - Storage for per-flow accounting metadata and statistics
## Challenges in Building Flow-level Middleboxes

- How to implement flow-processing features beneath its core logic?

<table>
<thead>
<tr>
<th>Approach</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Borrow code from open-source IDS (e.g., snort, suricata)</td>
<td>• 50K~100K code lines tightly coupled with their IDS logic</td>
</tr>
</tbody>
</table>
| Borrow code from open-source kernel (e.g., Linux/FreeBSD) | • Designed for TCP end host  
• Different from middlebox semantics                |
| Implement your own flow management code            | • Complex and error-prone  
• **Repeat** it for every custom middlebox |

- Implement your own flow management code
Difference from End-host TCP Applications

• Typical end-host TCP applications
  
  TCP application
  Berkeley Socket API
  TCP/IP stack ➔ Nice abstraction that separates TCP/IP stack from application

• Typical flow-processing middleboxes
  
  Middlebox application + Flow-processing logic
  Packet I/O stack ➔ Developers build own flow-processing logic from scratch (e.g., on top of PCAP, DPDK, PF_RING)

Our Goal
Build a reusable flow-processing networking stack for modular development of middleboxes
mOS Networking Stack

- A reusable stack for flow-processing middleboxes
  - Abstraction for sub-TCP layer middlebox operations

- Exposes programming abstractions
  - Monitoring sockets abstracting TCP flows
  - Flexible event system
  - Fine-grained resource usage

- Benefits
  - Clean, modular development of stateful middleboxes
  - Developers focus on core logic rather than flow management
  - Highly scalable on multi-10Gbps networks
Key Programming Abstractions in mOS

- For better reusability, mOS encourages decomposing a complex application into a set of <event, event handler> pairs
  - One can share a well-designed set of event definitions

- mOS provides two key programming abstractions:
  - mOS events for expressing custom flow-level conditions
  - mOS sockets for retrieving comprehensive flow-level features
Notable condition that merits middlebox processing

Built-in event (BE)
- Events that happen naturally in TCP processing
  - e.g., packet arrival, TCP connection start/teardown, retransmission

User-defined event (UDE)
- User can define their own event (= base event + filter function)

Key Abstraction: mOS Events

Built-in event
- New data arrival
- Packet arrival

User-defined event
- Filter (HTTP request)
- Filter (ACK packet)
- Filter (counter)

User-defined event
- HTTP request arrival
- ACK packet arrival
- 3 duplicate ACK arrival
Key Abstraction: mOS Monitoring Socket

- Abstracts a non-terminating **midpoint** of a ongoing connection
  - Simultaneously manages the flow states of both end-hosts
  - For every incoming flow, a new mOS monitoring socket is created

- To monitor fine-grained TCP-layer operations and metadata
  - e.g., abnormal packet retransmission, out-of-flow packet arrival, abrupt connection termination, employment of weird TCP/IP options

- Read flow-reassembled data or non-contiguous fragments

- Modify/drop the last packet that raised the event
mOS Flow Management

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

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

![Diagram of mOS stack emulation]

TCP client → SYN → SYN/ACK → ESTABLISHED → TCP server
mOS Flow Management

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

![Diagram showing mOS stack emulation with TCP states and transitions between SYN, SYN/ACK, DATA/ACK, and ESTABLISHED]

**TCP client**

**TCP server**
Scalable mOS Event Management

• Each flow can register/change its own set of events dynamically
  • Some flows may add or delete events
  • Some flows may change event handlers for registered events

• Scalability problem
  • How to efficiently manage event sets for 100K+ concurrent flows?
  • Naïve approach suffers from expensive copying of event sets

• Observation: the same event sets are shared by multiple flows
  • Reduces management overhead

Challenge
How to efficiently find/share the same event set?
Each socket points to an event invocation forest that records a set of flow events to wait on.

Data Structures for Event Management

Socket $S_1$

Event invocation forest $IF_1$

- `http_event`
- `ftp_event`
- `YouTube_event`

Event handlers:
- `OnFTPEvent()`
- `OnYouTubeRequest()`

Event objects:
- $e_1$, $e_2$, $e_3$, $e_4$, $e_5$, $e_6$, $e_7$
- $f_1$, $f_4$, $f_5$, $f_7$

Symbols:
- `socket`
- `built-in event`
- `UDE`
- `event handler`
Naïve way
1. \( s_1 \) registers a new event \(<e_3, f_3>\) \( \rightarrow \) IF\(_1\) is created
2. \( s_2 \) also registers the same event \(<e_3, f_3>\) \( \rightarrow \) IF\(_2\) is created

Problem
IF\(_1\) and IF\(_2\) are redundant!

Alternative
To reuse IF\(_1\) for \( s_2 \)
\( \rightarrow \) How does \( s_2 \) find IF\(_1\)?
Efficient Search for Dynamic Registration

- Each event invocation forest has an ID (searchable via hashtable)
  - \( id \) (invocation forest) = XOR sum of hash \((e + f)\)

- New invocation forest id after adding or deleting \(<e, f>\) from \(t\)
  - \( id \) (new forest) = \( id \) (old forest) \(\oplus\) hash \((e + f)\)
Efficient Search for Dynamic Registration

- Each event invocation forest has an ID (searchable via hashtable)
  - id (invocation forest) = XOR sum of hash (event + event handler)
- New invocation forest id after adding or deleting <e, f> from t
  - id (new forest) = id (old forest) ⊕ hash (e + f)

\[ s_1 \text{ registers a new event } <e_3, f_3> \]
\[ \text{id}(IF_0) \oplus h(e_3+f_3) = \text{id}(IF_1) \]
Efficient Search for Dynamic Registration

- Each event invocation forest has an ID (searchable via hashtable)
  - id (invocation forest) = XOR sum of hash (event + event handler)
- New invocation forest id after adding or deleting <e, f> from t
  - id (new forest) = id (old forest) ⊕ hash (e + f)

\[
\begin{align*}
\text{id}(\text{IF}_0) & \oplus h(e_3+f_3) = \text{id}(\text{IF}_0) \\
\text{id}(\text{IF}_1) & \oplus h(e_3+f_3) = \text{id}(\text{IF}_1)
\end{align*}
\]

\[s_2\] unregisters the event \(<e_3, f_3>\)
Efficient Search for Dynamic Registration

- Each event invocation forest has an ID (searchable via hashtable)
  - id (invocation forest) = XOR sum of hash (event + event handler)
- New invocation forest id after adding or deleting <e, f> from t
  - id (new forest) = id (old forest) ⊕ hash (e + f)

\[ s_1 \text{ unregisters the event } \langle e_3, f_3 \rangle \]
\[ \text{id}(IF_1) \oplus h(e_3+f_3) = \text{id}(IF_0) \]
Fine-grained Resource Management in mOS

• Not all middleboxes require full features
  • Some middleboxes do not require flow reassembly
Fine-grained Resource Management in mOS

- Not all middleboxes require full features
  - Some middleboxes do not require flow reassembly
  - Some middleboxes monitor only client-side data
Fine-grained Resource Management in mOS

- 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
mOS Stack Implementation

- Per-thread library TCP stack
  - ~26K lines of C code (mTCP [1] : ~11K lines)
- Shared nothing parallel architecture

[1] “mTCP: a highly scalable user-level TCP stack for multicore systems”, NSDI'14
Evaluation

1. Does mOS API support diverse middlebox applications?

2. Does mOS promise high performance?
mOS API Evaluation

• Does the API support diverse range of middleboxes?
  • Snort3 (strip ~10K lines)
    • Snort with mOS flow management
    • Replaces HTTP/TCP inspection module
  • nDPI
    • L7 protocol parsing over flow content
  • PRADS
    • Signature pattern matching on flow content

• Lessons learnt
  • mOS simplifies code
  • mOS patches vulnerabilities (nDPI/PRADS)
    • Detects signature that spans multiple segments
  • mOS does not degrade performance
    • Perform on par with respective vanilla (DPDK) versions

![Graph showing lines modified and total lines for Snort3, nDPI, and PRADS]
mOS API Evaluation (cont.)

- Does the API support diverse range of middleboxes?
  - Halfback proxy (128 lines)
    - Low latency proxy with proactive TCP retransmissions
  - Abacus (561 lines vs 4,091 lines)
    - Secure cellular data accounting system
  - Parallel NAT
    - High performance NAT
  - Midstat
    - netstat for middleboxes
  - L4 firewall
  - Etc.

- Applications ported to mOS: \(~9\times\) code line reduction
Performance Evaluation

• Does mOS provide high performance?

• mOS applications in **inline** mode
  • 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 192K 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)
Performance Scalability on Multicores

- File download traffic with 192K 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)

---

Performance linearly scales as # of cores are increased.
Latency overhead by mOS applications

- Direct connection
- Counting packets
- Searching for a string

Flow completion time (us)

- 64B file
- 8KB file

- 58.4
- 117.4
- 93.8
- 191.9
- 93.5
- 193.2

76us
Dynamic Event Registration Evaluation

- Monitor 192K concurrent flows
  - Flow size: 4KB
- Searching for a string in flow reassembled data
  - Dynamically register a new event when target string found
  - 50% client flows have target strings

![Throughput Graph](image-url)
Conclusion

- Software-based middleboxes have:
  - Modularity issues
  - Readability issues
  - Maintainability issues

- **mOS stack**: reusable networking stack for middleboxes
  - Programming abstraction with socket-based API
  - Event-driven middlebox processing
  - Efficient resource usage with dynamic resource composition

- mOS stack/API available @:
  [https://github.com/ndsl-kaist/mOS-networking-stack](https://github.com/ndsl-kaist/mOS-networking-stack)
Thank You

Questions?

http://mos.kaist.edu/
https://github.com/ndsl-kaist/mOS-networking-stack
Appendix
Extra Slides
Performance under Selective Resource Consumption

- full flow management
- w/o client buf management
- w/o buf management
- w/o client side
- w/o client side, w/o server buf mgmt.

<table>
<thead>
<tr>
<th>File size (B)</th>
<th>Throughput (Gbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>19.67</td>
</tr>
<tr>
<td>256</td>
<td>23.22</td>
</tr>
<tr>
<td>1K</td>
<td>29.6</td>
</tr>
<tr>
<td>4K</td>
<td>46.43</td>
</tr>
<tr>
<td>16K</td>
<td>59.97</td>
</tr>
</tbody>
</table>

38
Real applications performance

<table>
<thead>
<tr>
<th>Application</th>
<th>original + pcap</th>
<th>original + DPDK</th>
<th>mOS port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snort-AC</td>
<td>0.57 Gbps</td>
<td>8.18 Gbps</td>
<td>9.17 Gbps</td>
</tr>
<tr>
<td>Snort-DFC</td>
<td>0.82 Gbps</td>
<td>14.42 Gbps</td>
<td>15.21 Gbps</td>
</tr>
<tr>
<td>nDPIReader</td>
<td>0.66 Gbps</td>
<td>28.92 Gbps</td>
<td>28.87 Gbps</td>
</tr>
<tr>
<td>PRADS</td>
<td>0.42 Gbps</td>
<td>2.03 Gbps</td>
<td>1.90 Gbps</td>
</tr>
</tbody>
</table>

• Workload: real LTE packet trace (~67 GB)
• 4.5x ~ 28.9x performance improvement
• mOS brings code modularity & correct flow management
Events & Available Hooks

- Stream monitoring socket

<table>
<thead>
<tr>
<th>Built-in event</th>
<th>MOS_HK_SND</th>
<th>MOS_HK_RCV</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOS_ON_PKT_IN</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MOS_ON_CONN_START</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MOS_ON_CONN_END</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MOS_ON_TCP_STATE_CHANGE</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MOS_ON_REXMIT</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MOS_ON_CONN_NEW_DATA</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MOS_ON_ORPHAN</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- Raw monitoring socket

<table>
<thead>
<tr>
<th>Built-in event</th>
<th>MOS_HK_SND</th>
<th>MOS_HK_RCV</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOS_ON_PKT_IN</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Cellular Accounting with mOS Networking Stack

For every IP packet, $p$

- $p$ is retransmitted
  - yes
  - no
  - $p$’s payload == original payload
    - yes
      - skip accounting
    - no
      - account for $p$
- report abuse

4,639 LoC

Core Logic + Flow Mgmt

Event-action

561 LoC

Built-in events

- $e_{\text{REX}}$
- $e_{\text{NEW}}$

Filter

- $F_{\text{FAKE}}$

User-defined event

- $e_{\text{FAKE}}$

Event handler (action)

- $f_{\text{report}}$
- $f_{\text{acct}}$

Filter function

- $F_{\text{FAKE}}$

built-in event

- UDE

event handler

- MOS_ON_REXMIT
- MOS_ON_CONN_NEW_DATA
- IsFakeRexmit()
- UDE_FAKE_REXMIT
- ReportAbuse()
- AccountDataUsage()