mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes

Muhammad Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park

School of Electrical Engineering, KAIST

Abstract

Stateful middleboxes, such as intrusion detection systems and application-level firewalls, have provided key functionalities in operating modern IP networks. However, designing an efficient middlebox is challenging due to the lack of networking stack abstraction for TCP flow processing. Thus, middlebox developers often write the complex flow management logic from scratch, which is not only prone to errors, but also wastes efforts for similar functionalities across applications.

This paper presents the design and implementation of mOS, a reusable networking stack for stateful flow processing in middlebox applications. Our API allows developers to focus on the core application logic instead of dealing with low-level packet/flow processing themselves. Under the hood, it implements an efficient event system that scales to monitoring millions of concurrent flow events. Our evaluation demonstrates that mOS enables modular development of stateful middleboxes, often significantly reducing development efforts represented by the source lines of code, while introducing little performance overhead in multi-10Gbps network environments.

1 Introduction

Network appliances or “middleboxes”, such as intrusion detection systems and application accelerators, are widely deployed in modern networks [59]. With the trend towards commodity server-based middleboxes [59] and network functions virtualization [38], these middlebox applications are commonly implemented in software. Middlebox development, however, still remains an onerous task. It often requires handling complex flow-level states and events at layer 4 or above, such as connection state management and flow reassembly. The key challenge is that middlebox developers have to build these low-level flow management features from scratch, due to lack of common abstractions and well-defined APIs. This is in stark contrast to end-host applications programming, where application programmers rely on a set of networking system calls, such as the Berkeley socket API, that hides the details.

Existing socket APIs focus on end-to-end semantics and transferring application (layer 7) data. Unfortunately, they are not flexible enough to monitor session state, packet loss or retransmission patterns at lower layers. In contrast, popular packet processing frameworks, such as Click [46], DPDK [4], PacketShader IOEngine [40], and netmap [57], provide useful features for packet-level I/O processing, but lack flow-level abstraction required for stateful middlebox applications. A huge semantic gap exists between the two commonly-used abstractions. Thus, the state-of-the art middlebox programming remains that each application implements low-level flow-processing features in addition to the application-specific logic. This practice prevents code reuse and makes it challenging to understand the details of implementation. For example, we find that two popular NIDS implementations, Snort and Suricata, are drastically different, although they expose similar flow management features [19, 58].

This work presents the design and implementation of mOS, a reusable networking stack and an API for modular development of flow-processing middlebox applications. The design of mOS is based upon two principles. First, the API should facilitate a clear separation between low-level packet/flow processing and application-specific logic. While tight integration of the two layers might benefit performance, it easily becomes a source of complexity and a maintenance nightmare. In contrast, a reusable middlebox networking stack allows developers to focus on core middlebox application logic. Second, the middlebox networking API should provide programming constructs that natively support user-definable *flow events* for custom middlebox operations. Most middlebox operations are triggered by a set of custom flow events—being able to express them via a well-defined API is the key to modular middlebox programming. For example, a middlebox application that detects malicious payload in retransmission should be able to easily express the condition for the event and provide a custom action as its event handler. Building middlebox applications as a synthesis of event processing significantly improves the code readability while hiding the details for tracking complex conditions.

mOS satisfies a number of practical demands for middlebox development. First, it exposes a *monitoring socket* abstraction to precisely express the viewpoint of a middlebox on an individual TCP connection flowing through it. Unlike an end-host stack, mOS simultaneously manages the flow states of both end-hosts, which allows developers to compose arbitrary conditions of a flow state on either side. Second, mOS provides scalable monitoring. In high-speed networks with hundreds of thousands of concurrent flows, monitoring individual flow events incurs high overhead. Our event system significantly reduces the memory fooprint and memory bandwidth requirement for dynamic event registration and deregistration. Third, the mOS API supports fine-grained resource management on a per-flow basis. Developers can dynamically enable/disable event generation for an active flow and turn off tracking of unnecessary features. Tight controlling of computing resources leads to high performance as it avoids redundant cycle wastes. Finally, the mOS implementation extends the mTCP [43] codebase to benefit from the scalable user-level TCP stack architecture that harnesses modern multicore systems.

We make the following contributions. First, we present a design and implementation of a reusable flow-processing networking stack for modular development of high-performance middleboxes. Second, we present key abstractions that hide the internals of complex middlebox flow management. We find that the mOS monitoring socket and its flexible event composition provides an elegant separation between the low-level flow management and custom application logic. Third, we demonstrate its benefits in a number of real-world middlebox applications including Snort, Halfback, and Abacus.

2 Motivation and Approach

In this section, we explain the motivation for a unified middlebox networking stack, and present our approaches to its development.

2.1 Towards a Unified Middlebox Stack

mOS targets middleboxes that require L4-L7 processing but typically cannot benefit from existing socket APIs, including NIDS/NIPSes [3, 6, 19, 41, 58], L7 protocol analyzers [7, 9], and stateful NATs [5, 10]. These middleboxes track L4 flow states without terminating the TCP connections, often perform deep-packet inspection on flow-reassembled data, or detect anomalous behavior in TCP packet retransmission.

Unfortunately, developing flow management features for every new middlebox is very tedious and highly error-prone. As a result, one can find a long list of bugs related to flow management even in popular middleboxes [30–32, 34–36]. What is worse, some middleboxes fail to implement critical flow management functions. For example, PRADS [12] and nDPI [9] perform pattern matching, but they do not implement flow reassembly and would miss the patterns that span over multiple packets. Similarly, Snort's HTTP parsing module had long been packet-based [61] and vulnerable to pattern evasion attacks. While Snort has its own sophisticated flow management module, it is tightly coupled with other internal data structures, making it difficult to extend or to reuse.

A reusable middlebox networking stack would significantly improve the situation, but no existing packet processing frameworks meet the requirements of general-purpose flow management. `Click` [27, 37, 46] encourages modular programming of packet processing but its abstraction level is restricted to layer 3 or lower layers. `iptables` [5] along with its `conntrack` module supports TCP connection tracking, but its operation is based on individual packets instead of flows. For example, it does not support flow reassembly nor allows monitoring fine-grained TCP state change or packet retransmission. `libnids` [8] provides flow reassembly and monitors both server and client sides concurrently. Unfortunately, its reassembly logic is not mature enough to properly handle multiple holes in a receive buffer [17], and it does not provide any control knob to adjust the level of flow management service. Moreover, the performance of `iptables` and `libnids` depends on the internal kernel data structures that are known to be heavyweight and slow [43, 57]. `Bro` [55] provides flow management and events similar to our work, but its built-in events are often too coarse-grained to catch arbitrary conditions for middleboxes other than NIDS. While tailoring the event system to custom needs is possible through Bro's plugin framework [60], writing a plugin requires deep understanding of internal data structures and core stack implementation. In addition, `Bro` scripting and the implementation of its flow management are not designed for high performance, making it challenging to support multi-10G network environments.

2.2 Requirements and Approach

By analyzing various networking features of flow-processing middleboxes, we identify four key requirements for a reusable networking stack.

R1: *Middleboxes must be able to combine information from multiple layers using the stack.* For example, an NIDS must process each individual packet, but also be able to reconstruct bytestreams from TCP flows for precise analysis. A NAT translates the address of each IP packet, but it should also monitor the TCP connection setup and teardown to block unsolicited packets.

R2: *The networking stack must keep track of L4 states of both end-points, while adjusting to the level of state management service that applications require.* Tracking per-flow L4 state embodies multiple levels of services. Some applications require full TCP processing including reassembling bi-directional bytestreams; others only need basic session management, such as sequence number tracking while a few require single-side monitoring without bytestream management (*e.g.* TCP/UDP port blockers). Thus, we must dynamically adapt to application needs.

R3: The networking stack must provide intuitive abstactions to cater middlebox developers' diverse needs in a modular manner. It must provide flow-level abstractions that enable developers to easily manipulate flows and packets that belong to the flow. It should enable separation of its services from the application logic and allow developers to create higher-level abstraction that goes beyond the basic framework. Unfortunately, current middlebox applications often do not decouple the two. For instance, Snort heavily relies on its customized `stream`[1] preprocessing module for TCP processing, but its application logic (the detection engine) is tightly interlaced with the module. A developer needs to understand the Snort-specific `stream` programming constructs before she can make any updates on flow management attributes inside the detection engine.

R4: *The networking stack must deliver high performance.* Many middleboxes require throughputs of 10+ Gbps handling hundreds of thousands of concurrent TCP flows even on commodity hardware [2, 41, 42, 44, 62].

Our main approach is to provide a well-defined set of APIs and a unified networking stack that hides the implementation details of TCP flow management from custom application logic. mOS consists of four components designed to meet all requirements:

- **mOS networking API** is designed to provide packet- and flow-level abstractions. It hides the details of all TCP-level processing, but exposes information from multiple layers **(R1, R2, and R3)**, encouraging developers to focus on the core logic and write a modular code that is easily reusable.
- **Unified flow management:** mOS delivers a composable and scalable flow management library **(R2 and R4)**. Users can adjust flow parameters (related to buffer management *etc.*) dynamically at run time. Moreover, it provides a multi-core aware, high performing stack.
- **User-defined events:** Applications built on mOS are flexible since the framework provides not only a set of built-in events but also offers support for registering user-defined events **(R1 and R3)**.
- **User-level stack:** mOS derives its high performance from mTCP [43] that is a parallelizable userspace TCP/IP stack **(R4)**.

3 mOS Programming Abstractions

In this section, we provide the design and the usage of the mOS networking API and explain how it simplifies stateful middlebox development.

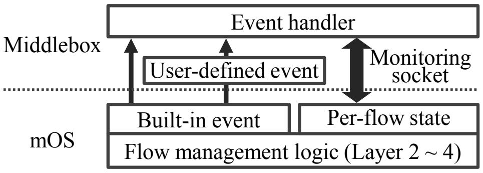


Figure 1: Interaction between mOS and its application

3.1 Monitoring Socket

The key abstraction that mOS exposes is a *monitoring socket*, which abstracts a middlebox's tap-point on a passing TCP flow or IP packets. Conceptually, it is similar to a Berkeley socket, but they differ in the operating semantics. First, a stream monitoring socket [2] represents a *non-terminating midpoint* of an active TCP connection. With a stream monitoring socket, developers write only high-level actions for an individual connection while underlying networking stack automatically tracks low-level TCP flow states of *both* client and server [3]. Second, a monitoring socket can monitor fine-grained TCP-layer operations while a Berkeley socket carries out coarse-grained, application-layer operations. For example, a monitoring socket can detect TCP or packet-level events such as abnormal packet retransmission, packet arrival order, abrupt connection termination, employment of weird TCP/IP options, etc., while it simultaneously supports reading flow-reassembled data from server or client.

Using the monitoring socket and its API functions (listed in Appendix A), one can write custom flow actions in a modular manner. First, a developer creates a 'passive' monitoring socket (similar to a listening socket) and binds it to a traffic scope, specified in a Berkeley packet filter (BPF) syntax. Only those flows/packets that fall into the scope are monitored. Note that there is no notion of "accepting" a connection since a middlebox does not engage in a connection as an explicit endpoint. Instead, one can express when custom action should be executed by setting up flow events as described in Section 3.2. All one needs is to provide the event handlers that perform a custom middlebox logic, since the networking stack automatically detects and raises the events by managing the flow contexts. When an event handler is invoked, it is passed an 'active' monitoring socket that represents the flow triggering the event. Through the socket, one can probe further on the flow state or retrieve and modify the last packet that raised the event. Figure 2 shows a code example that initializes a typical application with the mOS API.

3.2 Modular Programming with Events

The mOS API encourages modular middlebox programming by decomposing a complex application into a set of independent <event, event handler> pairs. It supports two classes of events: built-in and user-defined events.

[1] The `stream` module spans about 9,800 lines of code in Snort-3.0.0a1 version.

[2] Similarly, a *raw* monitoring socket represents IP packets.

[3] We call a connection initiator as client and its receiver as server.

Event	Description
MOS_ON_PKT_IN	In-flow TCP packet arrival
MOS_ON_CONN_START	Connection initiation (the first SYN packet)
MOS_ON_REXMIT	TCP packet retransmission
MOS_ON_TCP_STATE_CHANGE	TCP state transition
MOS_ON_CONN_END	Connection termination
MOS_ON_CONN_NEW_DATA	Availability of new flow-reassembled data
MOS_ON_ORPHAN	Out-of-flow (or non-TCP) packet arrival
MOS_ON_ERROR	Error report (e.g., receive buffer full)

Table 1: mOS built-in events for stream monitoring sockets. Raw monitoring sockets can use only MOS_ON_PKT_IN raised for every incoming packet.

Built-in events represent pre-defined conditions of notable flow state that are automatically generated in the process of TCP flow management in mOS. Developers can create their own user-defined events (UDEs) by extending existing built-in or user-defined events.

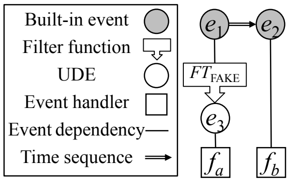
Built-in event: Built-in events are used to monitor common L4 events in an active TCP connection, such as start/termination of a connection, packet retransmission, or availability of new flow-reassembled data. With a state transition event, one can even detect any state change in a TCP state transition diagram. Table 1 lists eight built-in events that we have drawn from common functionalities of existing flow-processing middleboxes. These pre-defined events are useful in many applications that require basic flow state tracking. For example, developers can easily write a stateful NAT (or firewall) with only packet arrival and connection start/teardown events without explicitly tracking sequence/acknowledgment numbers or TCP state transitions. Also, gathering flow-level statistics at a network vantage point can be trivially implemented in only a few lines of code as shown in Appendix B.

User-defined event: For many non-trivial middlebox applications, built-in events are insufficient since they often require analyzing L7 content or composing multiple conditions into an event. For example, an event that detects 3 duplicate ACKs or that monitors an HTTP request does not have built-in support.

UDEs allow developers to systematically express such desired conditions. A UDE is defined as a base event and a boolean filter function that specifies the event condition. When the base event for an UDE is raised, mOS fires the UDE only if the filter function is evaluated to true. mOS also supports a multi-event filter function that can dynamically determine an event type or raise multiple events simultaneously. This feature is useful when it has to determine the event type or trigger multiple related events based on the same input data.

```
static void
mOSAppInit(mctx_t m)
{
  monitor_filter_t ft = {0};
  int s; event_t hev;

  // creates a passive monitoring socket with its scope
  s = mtcp_socket(m, AF_INET, MOS_SOCK_MONITOR_STREAM, 0);
  ft.stream_syn_filter = "dst net 216.58 and dst port 80";
  mtcp_bind_monitor_filter(m, s, &ft);

  // sets up an event handler for MOS_ON_REXMIT
  mtcp_register_callback(m,s,MOS_ON_REXMIT,MOS_HK_RCV,OnRexmitPkt);

  // defines a user-defined event that detects an HTTP request
  hev = mtcp_define_event(MOS_ON_CONN_NEW_DATA, IsHTTPRequest, NULL);

  // sets up an event handler for hev
  mtcp_register_callback(m, s, hev, MOS_HK_RCV, OnHTTPRequest);
}
```

Figure 2: Initialization code of a typical mOS application. Due to space limit, we omit error handling in this paper.



e_1	MOS_ON_REXMIT
e_2	MOS_ON_CONN_NEW_DATA
e_3	UDE_FAKE_REXMIT
FT_{FAKE}	IsFakeRexmit()
f_a	HandleFakeRexmitPacket()
f_b	AccountDataUsage()

Figure 3: Abacus event-action diagram

UDEs bring three benefits to event-driven middlebox development. First, new types of events can be created in a flexible manner because the filter function can evaluate arbitrary conditions of interest. A good filter function, however, should run fast without producing unnecessary side effect. Second, UDEs provide easy extensibility. One can create new events by extending any existing ones, including another UDE. For example, a developer can define a UDE that detects a YouTube [23] video request by extending a generic HTTP request UDE (e.g., using hev in Figure 2 as a base event). Third, it encourages code reuse. One can share a well-designed set of event definitions as a UDE library, and 3rd party developers can implement their own event handlers. For example, an open-source NIDS can declare all corner cases in flow management as a UDE library while 3rd party can provide custom actions to address each case.

3.3 Programming a Custom Middlebox

We show how one can build a custom flow-processing middlebox application using the mOS API and events. We pick Abacus [39] as an example here since it represents the needs of a real-world custom middlebox. Abacus is a cellular data accounting system that detects a “free-riding” attack by TCP-level tunneling. It has been reported that some cellular ISPs do not account for TCP retransmission packets [39], and this attack enables free-riding on cellular data by appending fake TCP headers that look like packet retransmission. The attack detection requires comparing the payload of original and retransmitted packets either by buffering a sliding window or sampling some bytes [39].

Writing Abacus with the mOS API is straightforward. First, we draw an event-action diagram that captures its main operations as shown in Figure 3. It represents the normal data as a built-in event (e_2, new data event) and registers a per-flow accounting function (f_b) for the event.

To detect the free-riding attack, we extend a built-in event (retransmission) to define a fake retransmission event (e_3). The filter function (FT_{FAKE}), as shown in Figure 4, determines whether the current packet retransmission is legal. In the code, `mtcp_getlastpkt()` retrieves the metadata (`pkt_info` structure) and the payload of the packet that triggers the retransmission event. `mctx_t` represents the thread context that the mOS stack is bound to and `sock` identifies the active flow that the packet belongs to. Then, the code uses `mtcp_ppeek()` to fetch the original flow-reassembled data at a specified sequence number offset and compares it with the payload if the sequence number ranges match. In case of partial retransmission, it calls `mtcp_getsockopt()` to retrieve non-contiguous TCP data fragments (`frags`) from a right flow buffer and compares the payload of the overlapping regions. If any part is different from the original content, it returns `true` and e_3 is triggered, or otherwise it returns `false`. When e_3 is triggered, f_a is executed to report an attack, and stops any subsequent event processing for the flow.

While Abacus is a conceptually simple middlebox, writing its flow management from scratch would require lots of programming effort. Depending on a middlebox, at least thousands to tens of thousands of code lines are required to implement basic flow management and various corner cases. The mOS API and its stack significantly save this effort while it allows the developer to write only the high-level actions in terms of events. Drawing an event-action diagram corresponds well to the application design process. Also, it better supports modular programming since the developer only needs to define their own UDEs and convert filters and event handlers into functions.

4 mOS Design and Implementation

In this section, we present the internals of mOS. At a high-level, mOS takes a stream of packets from a network, classifies and processes them by the flow, and triggers matching flow events. Inside event handlers, the application runs custom logic. mOS supports TCP flow state management for end-hosts, scalable event monitoring, extended flow reassembly, and fine-grained resource management. It is implemented by extending mTCP [43]. In total, it amounts to 27K lines of C code, which includes 11K lines of the mTCP code.

4.1 Stateful TCP Context Management

Automatic management of TCP contexts is the core functionality of mOS. For flow management, mOS keeps track of the following L4 states of both end-points: (1) TCP connection parameters for tracking initiation, state transition, and termination of each connection, (2) a payload reassembly buffer and a list of fragmented packets for detecting new payload arrival and packet retransmission. We further explain the payload reassembly buffer in Section 4.3.

```
static bool
IsFakeRexmit(mctx_t mctx, int sock, int side, event_t event,
             struct filter_arg *arg)
{
  struct pkt_info pi;
  char buf[MSS];
  struct tcp_ring_fragment frags[MAX_FRAG_NUM];
  int nfrags = MAX_FRAG_NUM;
  int i, size, boff, poff;

  // retrieve the current packet information
  mtcp_getlastpkt(mctx, sock, side, &pi);

  // for full retransmission, compare the entire payload
  if (mtcp_ppeek(mctx, sock, side, buf,
                 pi.payloadlen, pi.offset) == pi.payloadlen)
    return memcmp(buf, pi.payload, pi.payloadlen);

  // for partial retransmission, compare the overlapping region
  // retrieve the data fragments and traverse them
  mtcp_getsockopt(mctx, sock, SOL_MONSOCKET, (side == MOS_SIDE_CLI) ?
          MOS_FRAGINFO_CLIBUF : MOS_FRAGINFO_SVRBUF, frags, &nfrags);

  for (i = 0; i < nfrags; i++) {
    if ((size = CalculateOverlapLen(&pi, &(frags[i]), &boff, &poff)))
      if (memcmp(buf + boff, pi.payload + poff, size))
        return true;    // payload mismatch detected
  }
  return false;
}
```

Figure 4: A filter function that detects fake retransmission. CalculateOverlapLen() retrieves the size of sequence number overlap of the current packet and each fragment in a receive buffer.

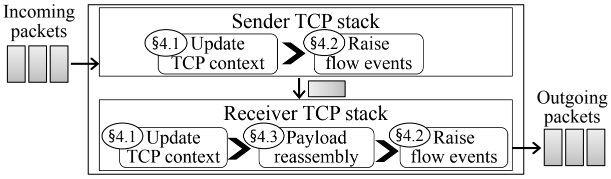
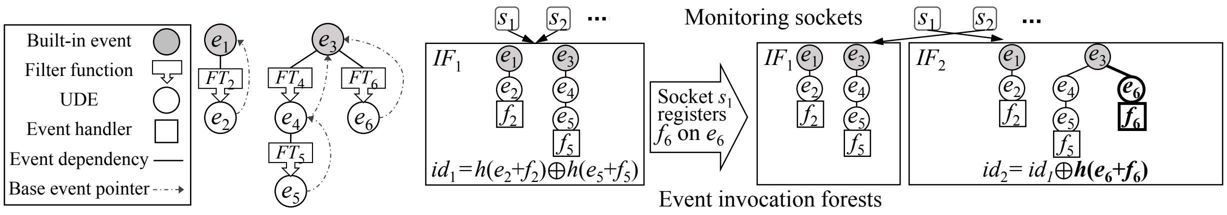


Figure 5: Packet processing steps in mOS

We design the TCP context update in mOS to closely reflect the real TCP state machine by emulating the states of both end-hosts. Tracking the flow states of both end-hosts is required as each side may take on a different TCP state. Figure 5 illustrates our model. When a packet arrives, mOS first updates its TCP context for the packet sender [4] and records all flow events that must be triggered. Note that event handlers are executed as a batch after the TCP context update. This is because intermixing them can produce an inconsistent state as some event handler may modify or drop the packet. Also, processing sender-side events before the receiver side's is necessary to strictly enforce the temporal order of the events. After sender-side stack update, mOS repeats the same process (update and trigger events) for the receiver-side stack. Any events relating to packet payload (new data or retransmission) are triggered in the context of a receiver since application-level data is read by the receiver. In addition, packet modification (or drop) is allowed only in the sender-side event handlers as mOS meticulously follows the middlebox semantics. The only exception is the retransmission event, which is processed just before receiver context update. This is to give

[4] Note that both server and client can be a packet sender.



(a) Global event dependency forest (d-forest)

(b) Dynamic event registration upon event invocation forests (i-forests)

Figure 6: mOS's event management. s_1 and s_2 originally share the same event invocation forest (i-forest), IF_1. If s_1 registers an event handler, f_6, then IF_2 is created. If s_2 repeats the same registration later, then s_2 simply moves its i-forest pointer to IF_2.

its event handler a chance to modify or drop the packet if the retransmission turns out to be malicious. While a receiver-side event handler cannot modify the packet, it can still reset a connection in case it detects malicious intrusion attempts in reassembled payload.

4.2 Scalable Event Management

mOS applications operate by registering for flow events and providing custom actions for them. For high performance, it is critical to have scalable event management with respect to the number of flows and user-defined events. However, a naïve implementation introduces a serious scalability challenge because mOS allows events to be registered and de-registered on a per-flow basis for fine-grained control. A busy middlebox that handles 200K concurrent flows and 1,000 UDEs per flow [5] amounts to managing a billion events [6], which would require large amount of memory and consume huge memory bandwidth. mOS provides an efficient implementation of internal data structures and algorithms designed to address the scalability challenge.

Data structures: Note UDEs form a tree hierarchy with its root being one of the eight built-in events. Thus, mOS maintains all custom flow event definitions in a global event dependency forest (d-forest) with eight dependency trees (d-trees). Figure 6(a) shows an example of a d-forest. Each node in a d-tree represents a UDE as a base event (e.g., parent node) and its filter function. Separate from the d-forest, mOS maintains, for each monitoring socket, its event invocation forest (i-forest) that records a set of flow events to wait on. Similar to the d-forest, an i-forest consists of event invocation trees (i-trees) where each i-tree maintains the registered flow events derived from its root built-in event. Only those events with an event handler are being monitored for the socket.

Addressing scalability challenge: If each socket maintains a separate i-forest, it would take up a large memory footprint and cause performance degradation due to redundant memory copying and releasing of the same i-forest. mOS addresses the problem by *sharing the same i-forest* with different flows. Our observation is that flows of the same traffic class (e.g., Web traffic) are likely to process the same set of events, and it is highly unlikely for all sockets to have a completely different i-forest. This implies that we can reduce the memory footprint by sharing the same i-forest.

When an active socket (e.g., individual TCP connection) is created, it inherits the i-forest from its passive monitoring socket (e.g., listening socket) by keeping a pointer to it. When an event is registered or de-registered for an active socket, mOS first checks if the resulting i-forest already exists in the system. If it exists, the active socket simply shares the pointer to the i-forest. Otherwise, mOS creates a new i-forest and adds it to the system. In either case, the socket adjusts the reference count of both previous and new i-forests, and removes the i-forest whose reference count becomes zero.

The key challenge lies in how to efficiently figure out if the same i-forest already exists in the system. A naïve implementation would require traversing every event node in all i-forests, which does not scale. Instead, we present an O(1) solution here. First, we devise a novel i-forest identification scheme that produces a unique id given an i-forest. We represent the id of an i-forest with m i-trees as: $t_1 \oplus t_2 \oplus ... \oplus t_m$, where t_k indicates the id of the k-th i-tree and $\oplus$ is a bitwise exclusive-or (`xor`) operation. Likewise, the id of an i-tree with n leaf event nodes is defined as: $h(e_1+f_1) \oplus h(e_2+f_2) \oplus ... \oplus h(e_n+f_n)$, where h is a one-way hash function, $+$ is simple memory concatenation, and e_i and f_i are the ids of the i-th event and its event handler, respectively. Note that the ids of a distinct event and its event handler are generated as unique in the system, which makes a distinct i-forest have a unique id with a high probability with a proper choice of the hash function. We include the id of an event handler in hash calculation since some event can be registered multiple times with a different handler. Calculating the new id of an i-forest after adding or removing an event becomes trivial due to the `xor` operation. Simply, new-id $=$ old-id $\oplus h(e+f)$ where e and f are the ids of the event and its event handler that need to be registered or deregistered. The fast id operation enables efficient lookup in the invocation forest hash table.

[5] Not unreasonable for an NIDS with thousands of attack signatures.

[6] 200K flows x 5,000 event nodes, assuming a UDE is derived from four ancestor base events on average.

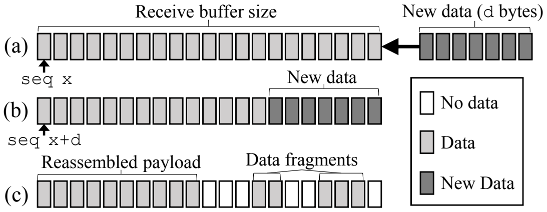


Figure 7: (a) When the receive buffer is full, mOS raises MOS_ON_ERROR to notify the application. (b) At overwriting, the internal read pointer is adjusted. (c) mtcp_getsockopt() exports data fragments caused by out-of-order packets.

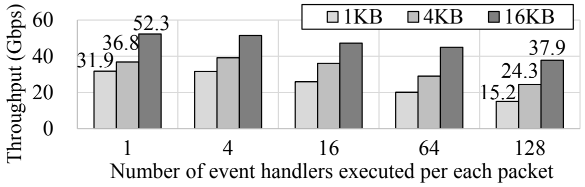


Figure 8: mOS performance over the number of events triggered per each packet. Performance measured with 192K concurrent connections fetching HTTP objects using a 60Gbps link. Event filters/handlers do minimal operation.

Deterministic ordering of event handler execution: A single packet arrival can trigger multiple events for a flow. Thus, the order of event handler execution must be predefined for deterministic operation. For this, we assign a fixed priority for all built-in events. First, packet arrival events (MOS_ON_PKT_IN and MOS_ON_ORPHAN) are processed because they convey the L3 semantics. Then, MOS_ON_CONN_START is triggered followed by MOS_ON_TCP_STATE_CHANGE and MOS_ON_CONN_NEW_DATA. Finally, MOS_ON_CONN_END is scheduled to give a chance to other events to handle the flow data before connection termination. Note, all built-in events are handled after TCP context update with an exception of MOS_ON_REXMIT, a special event triggered just before receive-side TCP context update.

All derived events inherit the priority of their root built-in event. mOS first records all built-in events that are triggered, and 'executes' each invocation tree in a forest by the priority order of the root built-in events. 'Executing' an invocation tree means traversing the tree in the breadth-first search order and executing each node by evaluating its event filter and running its event handler. For example, events in F_2 in Figure 6(b) are traversed in the order of $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4 \rightarrow e_6 \rightarrow e_5$.

4.3 Robust Payload Reassembly

Many middleboxes require L7 content scanning for detecting potential attack or application-specific patterns. mOS supports such applications with robust payload reassembly that handles a number of sophisticated cases.

Basic operation: mOS exposes mtcp_peek() and mtcp_ppeek() to the application for reading L7 data in a flow. Similar to recv(), mtcp_peek() allows the application to read the entire bytestream from an end-host. Internally, mOS maintains and adjusts a current read pointer for each flow as the application reads the data. mtcp_ppeek() is useful for retrieving flow data or fragments at an arbitrary sequence number.

Reassembly buffer outrun: Since TCP flow control applies between end-hosts, the receive buffer managed by mOS can become full while new packets continue to arrive (see Figure 7 (a)). Silent content overwriting is undesirable since the application may not notice the buffer overflow. Instead, mOS raises an error event to explicitly notify the application about the buffer overflow. The application can either drain the buffer by reading the data or enlarge the buffer. Otherwise, mOS overwrites the buffer with the new data, and adjusts the internal read pointer(see Figure 7 (b)). To notify the application about the overwriting, we make mtcp_peek() fail right after overwriting. Subsequent function calls continue to read the data from the new position. Notifying the application about buffer overflow and overwriting allows the developer to write correct operations even at corner cases.

Out-of-order packet arrival: Unlike the end-host TCP stack, some middlebox applications must read partially-assembled data, especially when detecting attack scenarios with out-of-order or retransmitted packets. mOS provides data fragment metadata by mtcp_getsockopt(), and the application can retrieve payload of data fragment by mtcp_ppeek() with a specific sequence number to read (see Figure 7 (c)).

Overlapping payload arrival: Another issue lies in how to handle a retransmitted packet whose payload overlaps with the previous content. mOS allows to express a flexible policy on content overlap. Given that the update policy differs by the end-host operating systems [53], mOS supports both policies (e.g., overwriting with the retransmitted payload or not) that can be configured on a per-flow basis. Or a developer can register for a retransmission event and implement any custom policy of her choice.

4.4 Fine-grained Resource Management

A middlebox must handle a large number of concurrent flows with limited resources. mOS is designed to adapt its resource consumption to the computing needs as follows.

Fine-grained control over reassembly: With many concurrent flows, the memory footprint and memory bandwidth consumption required for flow reassembly can be significant. This is detrimental to those applications that do not require flow reassembly. To support such applications, mOS allows disabling or resizing/limiting the TCP receive buffer at run-time on a per-flow basis. For example, middleboxes that rely on IP whitelisting modules (*e.g.* Snort's IP reputation preprocessor [18]) can use this feature to dynamically disable buffers for those flows that arrive from whitelisted IP regions.

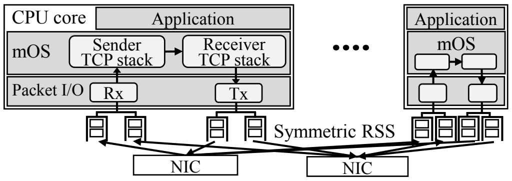


Figure 9: mOS application threading model

Uni-directional monitoring: Some middleboxes may want to monitor only the client-side requests or others deployed before server farms may be interested only in the ingress traffic. In such a case, developers can turn off TCP state management of one side. Disabling the TCP stack of one side would ignore raising events, stack update, and flow reassembly.

Dynamic event management: The number of registered events affects the overall performance, as shown in Figure 8. When a large number of UDEs are naïvely set up for all flows, it degrades the performance due to frequent filter invocations. To address this, the mOS API supports dynamic cancellation of registered events. For example, one can stop scanning the flow data for attack signatures beyond a certain byte limit [51]. Alternatively, one can register for a new event or switch the event filter depending on the characteristics of each flow. Such a selective application of flow events provides flexibility to the developers, while minimizing the overall resource consumption.

Threading model: mOS adopts the shared-nothing parallel-processing architecture that effectively harnesses modern multi-core CPUs. Figure 9 shows the threading model of mOS. At start, it spawns n independent threads, each of which is pinned to a CPU core and handles its share of TCP flows using symmetric receive-side scaling (S-RSS) [63]. S-RSS maps all packets in the same TCP connection to the same RX queue in a network interface card (NIC), by making the Toeplitz hash function [47] produce the same value even if the source and destination IP/port pairs on a packet are swapped. This enables line-rate delivery of packets to each thread as packet classification is done in NIC hardware. Also, it allows each mOS thread to handle entire packets in a connection without sharing flow contexts with other threads, which avoids expensive inter-core locks and cache interference. We adopt flow-based load balancing as it is reported to achieve a reasonably good performance with real traffic [63].

mOS reads multiple incoming packets as a batch but processes each packet by the run-to-completion model [28]. mOS currently supports Intel DPDK [4] and netmap [57] as scalable packet I/O, and supports the pcap library [20] for debugging and testing purposes. Unlike mTCP, the application runs event handlers in the same context of the mOS thread. This ensures fast event processing without context switching.

Appl	Modified	SLOC	Output
Snort	2,104	79,889	Stateful 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 tunneling

Table 2: Summary of mOS application updates. Snort's SLOC represents the code lines that are affected by our porting.

5 Evaluation

This section evaluates mOS by answering three key questions: (1) Does the mOS API support diverse use cases of middlebox applications? (2) Does mOS provide high performance? (3) Do mOS-based applications perform correct operations without introducing non-negligible overhead?

5.1 mOS API Evaluation

For over two years of mOS development, we have built a number of middlebox applications using the mOS API. These include simple applications such as a stateful NAT, middlebox-netstat, and a stateful firewall as well as porting real middlebox applications, such as Snort [58], nDPI library [9], and PRADS [12] to use our API. Using these case studies, we demonstrate that the mOS API supports diverse applications and enables modular development by allowing developers to focus on the core application logic. As shown in Table 2, it requires only 2%-12% of code modification to adapt to the mOS framework. Moreover, our porting experience shows that mOS applications have clear separation of the main logic from the flow management modules. We add a prefix 'm', to the name of mOS-ported application (*e.g.,* Snort → mSnort).

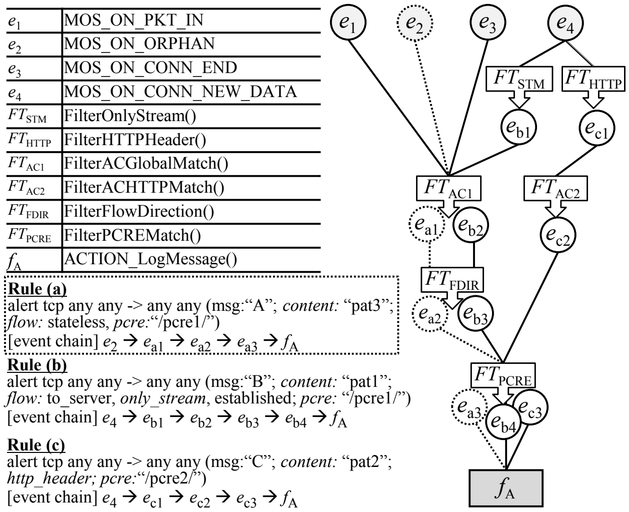
mSnort3: We demonstrate that the mOS API helps modularize a complex middlebox application by porting Snort3 [16] to using mOS. A typical signature-based NIDS maintains a set of attack signatures (or rules) and examines whether a flow contains the attack patterns. The signatures consist of a large number of rule options that express various attack patterns (e.g., content, pcre), payload type (*e.g.,* http_header) and conditions (e.g., only_stream and to_server). To enhance the modularity of Snort, we leverage the monitoring socket abstraction and express the signatures using event-action pairs.

To transform the signatures into event-action pairs, we express each rule option type as a UDE filter, and synthesize each rule as a chain of UDEs. We use three synthetic rules shown in Figure 10 as an example. For example, the `http_header` rule option in rule (c) corresponds to the filter function FT_{HTTP} that triggers an intermediate event e_{c1}. e_{c1} checks FT_{AC2} for string pattern matching and triggers e_{c2}, which in turn runs PCRE pattern matching (FT_{PCRE}), triggers e_{c3} and finally executes its event handler (f_A). Note, Snort scans the traffic against these rules multiple times: (a) each time a packet arrives, (b) whenever enough flow data is reassembled, and (c) whenever a flow finishes. (b) and (c) are required to detect attack patterns that spread over multiple packets in a flow. These naturally correspond to the four mOS built-in events (e_1 to e_4) in Figure 10.

One challenge in the process lies in how one represents the `content` rule option. The `content` rule option specifies a string pattern in a payload, but for efficient pattern matching, it is critical that the payload should be scanned once to find all the string patterns specified by multiple rules. To reflect this need, we use a multi-event filter function that performs Snort's Aho-Corasick algorithm [24]. Given a message, it scans the payload only once and raises distinct events for different string patterns.

We have implemented 17 rule options (out of 45 options[7]) that are most frequently used in Snort rules. Our implementation covers HTTP attack signatures as well as general TCP content attack patterns. The actual implementation required writing a Snort rule parser that converts each rule into a series of UDEs with filter functions so that mSnort3 can run with an arbitrary set of attack signatures. In total, we have modified 2,104 lines out of 79,889 lines of code which replaces the Snort's `stream5` and `http-inspect` modules that provide flow management and HTTP attack detection, respectively.

mSnort3 benefits from mOS in a number of ways. First, we find that each UDE is independent from the internal implementation of flow management, which makes the code easy to read and maintain. Also, the same filter function is reused to define different intermediate UDEs since it can be used to extend a different base event. This makes the rule-option evaluator easier to write, understand, and extend. In contrast, Snort's current rule-option evaluator, which combines the evaluation results of multiple rule options in a rule, is a very complex recursive function that spans over 500 lines of code. Second, since the attack signatures are evaluated in a modular manner, one can easily add new rule options or remove existing ones without understanding the rest of the code. In contrast, such modification is highly challenging with existing `stream5` and `http-inspect` modules since other Snort's modules heavily depend on internal structures and implementation of the two. Third, rules that share the same prefix in the event chain would benefit from sharing the result of event evaluation. Say, two rules are represented as $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4$, and $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_5 \rightarrow e_6$. mOS ensures to evaluate up to e_3 only once and shares the result between the two rules. Fourth, mSnort3 can now leverage more fine-grained monitoring features of mOS such as setting a different buffer size per flow or selective buffer management. These features are difficult to implement in the existing code of Snort3.



e_1	MOS_ON_PKT_IN
e_2	MOS_ON_ORPHAN
e_3	MOS_ON_CONN_END
e_4	MOS_ON_CONN_NEW_DATA
FT_{STM}	FilterOnlyStream()
FT_{HTTP}	FilterHTTPHeader()
FT_{AC1}	FilterACGlobalMatch()
FT_{AC2}	FilterACHTTPMatch()
FT_{FDIR}	FilterFlowDirection()
FT_{PCRE}	FilterPCREMatch()
f_A	ACTION_LogMessage()

Rule (a)
alert tcp any any -> any any (msg:“A”; *content:* “pat3”;
flow: stateless, *pcre:*“/pcre1/”)
[event chain] $e_2 \rightarrow e_{a1} \rightarrow e_{a2} \rightarrow e_{a3} \rightarrow f_A$

Rule (b)
alert tcp any any -> any any (msg:“B”; *content:* “pat1”;
flow: to_server, *only_stream*, established; *pcre:* “/pcre1/”)
[event chain] $e_4 \rightarrow e_{b1} \rightarrow e_{b2} \rightarrow e_{b3} \rightarrow e_{b4} \rightarrow f_A$

Rule (c)
alert tcp any any -> any any (msg:“C”; *content:* “pat2”;
http_header; pcre:“/pcre2/”)
[event chain] $e_4 \rightarrow e_{c1} \rightarrow e_{c2} \rightarrow e_{c3} \rightarrow f_A$

Figure 10: mSnort event-action diagram of sample rules. Keywords in a rule (e.g., ‘content’ and ‘pcre’) are translated into a chain of multiple UDEs. String search (‘content’ option) is performed first. ‘only_stream’ inspects only flow-reassembled data. The ‘msg’ string is printed out at f_A if the input data meets all rule options.

mAbacus: The original Abacus code is based on Monbot [63], a home-grown flow monitor for analyzing the content-level redundancy. Abacus reuses the packet I/O and flow management modules of Monbot, while disabling irrelevant features like content hashing. Although the core logic of Abacus is simple, its original implementation requires understanding low-level flow management and modifying 1,880 lines (out of 4,808 lines) of the code. We write mAbacus from a clean slate and in a top-down approach. mAbacus exploits the monitoring socket abstraction to monitor TCP packet retransmission, flow creation/termination and payload arrival events for accounting purpose. Compared to the original version, mAbacus brings two extra benefits. First, it correctly detects retransmission over fragmented segments from out-of-order packets in a receive buffer. Second, one can disable a receive buffer of any side in a single line of code , while original Abacus requires commenting out 100+ lines of its flow processing code manually. The new implementation requires only 561 lines of code. This demonstrates that mOS hides the details of TCP flow management and allows application developers to focus on their own logic.

mHalfback: Halfback [50] is a transport-layer scheme designed for optimizing the flow completion time (FCT). It relies on two techniques: (i) skipping the TCP slow start phase to pace up transmission rate at start, and (ii) performing proactive retransmission for fast packet loss recovery. Inspired by this, we design mHalfback, a middlebox application that performs proactive retransmission over TCP flows. mHalfback has no pacing phase, since a middlebox cannot force a TCP sender to skip the slow start phase. Instead, mHalfback provides fast recovery of any packet

[7] Remaining options are mostly unrelated to HTTP/TCP protocols.

loss, so that it transparently reduces the FCT without any modification of end-host stacks. The main logic of mHalfback is as follows: (i) when a TCP data packet arrives, mHalfback holds a copy of the packet for future retransmission. (ii) when a TCP ACK packet comes from the receiver, mHalfback will retransmit data packets (up to a certain threshold). mHalfback calls `mtcp_getlastpkt()` to hold the packet, and `mtcp_sendpkt()` for proactive retransmission. When the volume of per-flow data exceeds a retransmission threshold (e.g., [50] uses 141 KB), it deregisters the packet arrival event for the flow, so that any packet beyond the threshold would not be retransmitted. Likewise, mHalfback stops monitoring a flow when its connection is closed. Connection closure is easily detected with the state change built-in event. With the help of mOS monitoring socket, mHalfback implementation requires only 128 lines of code.

mnDPI library: nDPI [9] is an open-source DPI library that detects more than 150 application protocols. It scans each packet payload against a set of known string patterns or detects the protocol by TCP/IP headers. Unfortunately, it neither performs flow reassembly nor properly handles out-of-order packets. We have ported nDPI (`libndpi`) to use the mOS API by replacing their packet I/O and applying UDEs derived from 4 built-in events as in mSnort3. Our porting enables all applications that use `libndpi` to detect the patterns over flow-reassembled data. This requires adding 765 lines of code to the existing 25,483 lines of code.

mPRADS: PRADS [12] is a passive fingerprinting tool that detects the types of OSes and server/client applications based on their network traffic. It relies on PCRE [13] pattern matching on TCP packets for this purpose. Like nDPI, PRADS does not perform flow reassembly. Furthermore, despite its comprehensive pattern set, the implementation is somewhat ad-hoc since it inspects only the first 10 (which is an arbitrarily set threshold) packets of a flow. mPRADS employs 24 UDEs on `MOS_ON_CONN_NEW_DATA` to detect different L7 protocols in separate event handlers. This detects the patterns regardless of where the pattern appears in a TCP connection. We modify only 615 lines out of 10,848 lines to update mPRADS.

5.2 mOS Performance

We now evaluate the performance of mOS, including the flow management and event system.

Experiment setup: We evaluate mOS applications on a machine with dual Intel E5-2690 v0 CPUs (2.90GHz, 16 cores in total), 64 GB of RAM, and 3 dual-port Intel 82599 10G NICs. For flow generation, we use six pairs of clients and servers (12 machines) where each pair communicates via a 10G link through an mOS application. Each client/server machine is equipped with one Intel Xeon E3-1220 v3 CPU (4 core, 3.10 GHz), 16 GB of RAM, and a Intel 10G NIC. All machines run Linux kernel version 3.13 and use Intel's DPDK v16.04.

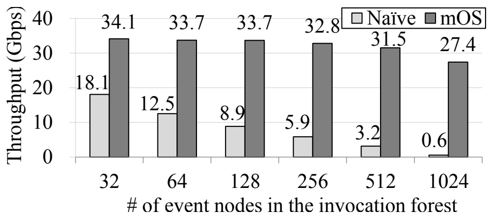


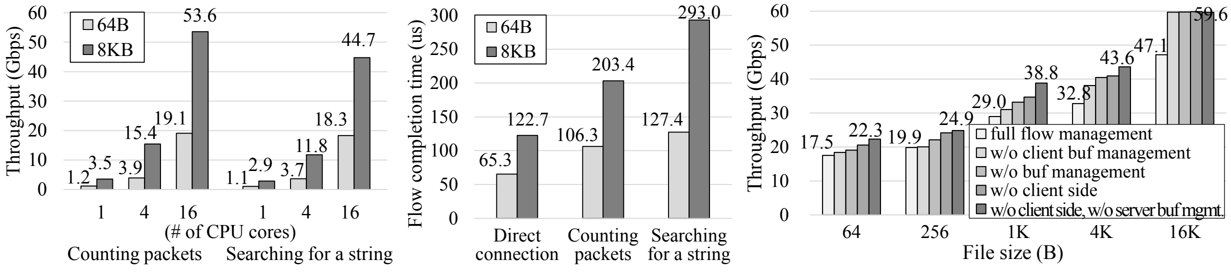
Figure 12: Performance at dynamic event registration

Synthetic workload: The client spawns many concurrent flows that download HTTP objects of the same size from a server. When an object download completes, the client fetches another object so that the number of concurrent connections stays the same. Both the client and server are implemented with mTCP [43] for high performance, and all six pairs can generate up to 192K concurrent flows.

Microbenchmarks: Figure 11(a) shows the performance of mOS applications when the clients fetch HTTP objects (64 bytes or 8 KB) with 192K concurrent connections. `packetcount` counts the number of packets per each flow (by using a `MOS_ON_PKT_IN` event handler) while `stringsearch` performs keyword string search over flow-reassembled data (by using a `MOS_ON_CONN_NEW_DATA` event handler). We observe that the performance almost linearly scales over the number of CPU cores, and both applications achieve high throughputs despite a large number of concurrent flows when multiple CPU cores are employed. At 16 cores, `packetcount` produces 19.1 Gbps and 53.6 Gbps for 64 bytes and 8 KB objects, respectively while `stringsearch` achieves 18.3 Gbps and 44.7 Gbps for 64 bytes and 8 KB, respectively. Figure 11(b) shows the flow completion time for the two mOS applications. mOS applications add 41 $\sim$ 62 us of delay to that of a direct connection (without any middlebox) for 64-byte objects and 81 $\sim$ 170 us of latency for 8 KB objects. We believe the latency stretch is reasonable even when a middlebox operates in the middle.

Figure 11(c) compares the performances of application `packetcount` under various levels of resource consumption. Depending on the file size, selective resource configuration improves the performance by up to 25% to 34% compared with the full flow management of both sides. Not surprisingly, disabling the entire state update of one side provides the biggest performance boost, but skipping flow buffering also brings non-trivial performance improvement. This confirms that tailoring resource consumption to the needs of a specific middlebox application produces significant performance benefit.

Efficient i-forest management: Figure 12 compares the performance of `stringsearch` as it dynamically registers for an event. Clients download 4KB objects with 192K concurrent flows. When the application finds the



(a) mOS multicore scalability (b) Flow completion time (c) Performance under selective resource consumption

Figure 11: Microbenchmark experiments with a standalone middlebox.

target string, it registers for an extra event with the socket to inspect the data further. We have the server inject the target string for 50% of object downloads, and vary the number of initial registered events per socket from 32 to 1024. A naïve implementation would take a snapshot of the invocation forest and copy the entire forest to dynamically register for an event. In contrast, our algorithm looks up the same invocation forest and spawns a new one only if it does not exist. Our algorithm outperforms the naïve implementation by 15.4 to 26.8 Gbps depending on the number of event nodes, which confirms that fast identification of the same forest greatly enhances the performance at dynamic event registration.

Memory footprint analysis: Each TCP context takes up 456 bytes of metadata and starts with a 8 KB flow buffer. So, 192K concurrent flows would consume about 3.2 GB with bidirectional flow reassembly. For i-forest management, an event node takes up 128 bytes. Thus, monitoring 4,000 events per connection (e.g., Snort) would consume n * 512 KB, if the application ends up generating n distinct i-forests for updating events for k times during its operation (typically, $k >> n$). In contrast, a naïve implementation would consume k * 512 KB, which would use 98 GB if all 192K flows update their i-forest dynamically ($k = 192K$).

Handling abnormal traffic: We evaluate the behavior of mOS when it sees a large number of abnormal flows. We use the same test environment, but have the server transmit the packets out of order (across the range of sender's transmission window) at wire rate. We confirm that (a) mOS correctly buffers the bytestream in the right order, and (b) its application shows little performance degradation from when clients and servers directly communicate.

5.3 mOS Application Evaluation

We verify the correctness of mOS-based middleboxes and evaluate their performance with real traffic traces.

Correct flow reassembly: We test the correctness of mnDPI, mPRADS, and mAbacus under lightweight load with 24K concurrent flows, downloading 64KB HTTP objects. We randomly inject 10 different patterns where each pattern crosses over 2 to 25 packets in different flows and see if mnDPIReader (a simple DPI application using libndpi) and mPRADS detect them. We repeat the test for 100 times, and confirm that mnDPI and mPRADS successfully detect the flows while their original versions miss them. We also inject 1K fake retransmission flows and find that mAbacus successfully detects all such attacks.

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
Abacus	-	-	28.48 Gbps

Table 3: Performance of original and mOS-ported applications under a real traffic trace. Averaged over five runs.

Performance under real traffic: We measure the performance of original applications and their mOS ports with a real network packet trace. The trace is obtained from a large cellular ISP in South Korea and records 89 million TCP packets whose total size is 67.7 GB [63]. It contains 2.26 million TCP connections and the average packet size is 760 bytes. We replay the traffic at the speed of 30 Gbps to gauge the maximum performance of mOS-ported applications. We use the same machines as in Section 5.2.

Table 3 compares the performances of Snort, nDPIReader, PRADS, and Abacus. Snort-DFC uses a more efficient multi-string matching algorithm, DFC [33], instead of the Aho-Corasick algorithm (Snort-AC). Original applications (except Abacus) use the pcap library by default, which acts as the main performance barrier. Porting them to use the DPDK library greatly improves the performance by a factor of 4.9 to 44.6 due to scalable packet I/O. mOS-based applications deliver comparable performances to those of DPDK-ports while mOS ports provide code modularity and correct operation in pattern matching. This confirms that mOS does not incur undesirable performance overhead over DPDK-ported applications.

mSnort is actually slightly faster than Snort+DPDK. This is mainly due to the improved efficiency of mOS's flow management over Snort's `stream5`. Our profiling finds that the `stream5` module incurs more memory accesses per packet on average. Compared to others, PRADS shows much lower performance because it naïvely performs expensive PCRE pattern matching on the traffic.

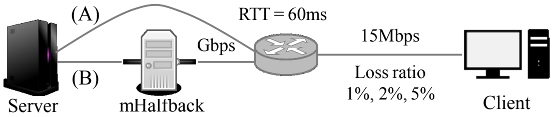


Figure 13: Configuration environment for mHalfback evaluation

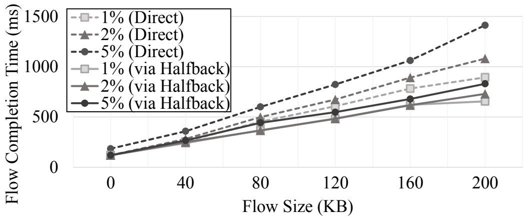


Figure 14: Average flow completion time by packet loss ratio

The flow management overhead in mnDPIReader and mPRADS is small, representing only about 1 to 4% of performance change.

Performance under packet loss: We evaluate mHalfback by injecting Web traffic into a lossy network link. We test with the same topology used in the original paper [50] as shown in Figure 13. Figure 14 compares average FCT of a direct connection (Figure 13(A)) and of a connection via Halfback proxy (Figure 13(B)) under various packet loss rates. We find that mHalfback significantly reduces the FCT with the help of fast loss recovery. When testing with flows that download 100 KB under 5% packet loss, mHalfback brings 20% to 41% FCT reduction. While this is lower than 58% FCT reduction reported by the original paper, it is promising as it does not require any modification on the server.

6 Related Work

We discuss previous works that are related to mOS.

Flow management support: `libnids` [8] is a flow-level middlebox library that can be used for building middlebox applications but it does not provide comprehensive flow reassembly features [17, 22]. Bro [55] provides an event-driven scripting language for network monitoring. mOS differs from Bro in its programming model. mOS is designed to write broader range of network applications and provides an API that allows more fine-grained control over live connections. A mOS middlebox developer can *dynamically* register new events per flow at any stage of a connection life cycle, a flow's TCP receive buffer management can be disabled at run-time, and monitoring of any side (client or server) of the flow can be disabled dynamically. Bro does not offer such features.

Modular middlebox development: Click [46] provides a modular packet processing platform, which allows development of complex packet forwarding applications by chaining *elements*. Click has been a popular platform in research community to implement L3 network function prototypes [26, 29, 45, 49, 52]. mOS, on the other hand, provides comprehensive, flexible flow-level abstractions that allow mapping a custom flow-level *event* to a corresponding *action*, and is suitable for building L4-L7 monitoring applications. CliMB [48] provides a modular TCP layer composed of Click elements, but its TCP-based elements are designed only for end-host stacks; whereas mOS facilitates programming middleboxes.

xOMB [25] is a middlebox architecture that uses programmable pipelines that simplify the development of inline middleboxes. Although xOMB shares the goal of simplifying development of flow-level middleboxes, its system is focused on an L7 proxy, which uses BSD sockets to initialize and terminate connections. mOS focuses on exposing flow-level states and events for general-purpose monitoring applications without the ability to create new connections. CoMb [59] aims to provide efficient resource utilization by consolidating common processing actions across multiple middleboxes; while mOS cuts down engineering effort by combining common flow-processing tasks on a single machine dataplane.

Scalable network programming libraries: Several high-speed packet I/O frameworks have been proposed [4, 14, 40, 57]. However, extending these frameworks to support development of stateful middlebox applications requires significant software engineering effort. mOS uses mTCP [43], a scalable user-level multicore TCP stack for stateful middleboxes. The performance scalability of mOS comes from mTCP's per-thread socket abstraction, shared-nothing parallel architecture, and scalable packet I/O. IX [28] and Arrakis [56] present new networking stack designs by separating the kernel control planes from data planes. However, both models only provide endpoint networking stacks. There are a few on-going efforts that provide fast-path networking stack solutions [1, 11, 15, 21] for L2/L3 forwarding data planes. We believe mOS is the first fast-path networking stack which provides comprehensive flow-level monitoring capabilities for L4-L7 stateful middleboxes. Modnet [54] provides modular TCP stack customization for demanding applications, but only for end host stack.

7 Conclusion

Modular programming of stateful middleboxes has long been challenging due to complex low-level protocol management. This works addresses the challenge with a general-purpose, reusable networking stack for stateful middleboxes. mOS provides clear separation of interface and implementation in building complex stateful middleboxes. Its flow management module provides accurate tracking of the end-host states, enabling the developer to interact with the system with a well-defined set of APIs. Its flow event system flexibly expresses per-flow conditions for custom actions. The mOS source code is available at `https://github.com/ndsl-kaist/mos-networking-stack`.

8 Acknowledgments

We would like to thank our shepherd Minlan Yu and anonymous reviewers of NSDI'17 for their insightful comments on the paper. We also thank Shinae Woo and Keon Jang for their contributions on the early design of the mOS networking stack. This work was supported in part by the ICT R&D Program of MSIP/IITP, Korea, under Grants B0126-16-1078, B0101-16-1368 [Development of an NFV-inspired networked switch and an operating system for multi-middlebox services],[2016-0-00563, Research on Adaptive Machine Learning Technology Development for Intelligent Autonomous Digital Companion], and by the National Research Council of Science & Technology (NST) grant by the Korea government (MSIP) No. CRC-15-05-ETRI.

References

[1] 6WINDGate Software: Network Optimization Software. http://www.6wind.com/products/6windgate/.

[2] Accelerating Snort with PF_RING DNA – ntop. http://www.ntop.org/pf_ring/accelerating-snort-with-pf_ring-dna/.

[3] Argus - NSMWiki. http://nsmwiki.org/index.php?title=Argus.

[4] Intel DPDK: Data Plane Development Kit. http://dpdk.org/.

[5] IPTABLES manpage. http://ipset.netfilter.org/iptables.man.html.

[6] Kismet Wireless Network Detector. https://www.kismetwireless.net/.

[7] L7-filter: Application Layer Packet Classifier for Linux. http://l7-filter.sourceforge.net/.

[8] Libnids. http://libnids.sourceforge.net/.

[9] nDPI | ntop. http://www.ntop.org/products/ndpi/.

[10] netfilter/iptables project homepage. http://www.netfilter.org/.

[11] OpenFastPath: Technical Overview. http://www.openfastpath.org/index.php/service/technicaloverview/.

[12] Passive Real-time Asset Detection System. http://gamelinux.github.io/prads/.

[13] PCRE - Perl Compatible Regular Expressions. http://www.pcre.org/.

[14] PF_RING ZC (Zero Copy). http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/.

[15] Project Proposals/TLDK. https://wiki.fd.io/view/Project_Proposals/TLDK.

[16] Snort 3.0. https://www.snort.org/snort3.

[17] Snort: Re: Is there a snort/libnids alternative. http://seclists.org/snort/2012/q4/396.

[18] Snort: Reputation Preprocessor. http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node17.html#SECTION003220000000000000000.

[19] Suricata Open Source IDS/IPS/NSM engine. http://suricata-ids.org/.

[20] Tcpdump & Libpcap. http://www.tcpdump.org/.

[21] The Fast Data Project (FD.io). https://fd.io/.

[22] wireshark - using "follow tcp stream"; code in my c project - Stack Overflow. https://ask.wireshark.org/questions/39531/using-follow-tcp-stream-code-in-my-c-project.

[23] YouTube. https://www.youtube.com/.

[24] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, June 1975.

[25] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: extensible open middleboxes with commodity servers. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2012.

[26] B. Anwer, T. Benson, N. Feamster, and D. Levin. Programming Slick Network Functions. In *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, 2015.

[27] T. Barbette, C. Soldani, and L. Mathy. Fast Userspace Packet Processing. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2015.

[28] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[29] A. Bremler-Barr, Y. Harchol, and D. Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[30] Bug #1238: Possible evasion in stream-reassemble.c. Suricata-3.11 ChangeLog. https://github.com/inliniac/suricata/blob/master/ChangeLog.

[31] Bug #1557: Stream: retransmission not detected. Suricata-3.11 ChangeLog. https://github.com/inliniac/suricata/blob/master/ChangeLog.

[32] Bug #1684: eve: stream payload has wrong direction in IPS mode. Suricata-3.11 ChangeLog. https://github.com/inliniac/suricata/blob/master/ChangeLog.

[33] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han. DFC: Accelerating Pattern Matching for Network Applications. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[34] Common Vulnerabilities and Exposures. CVE - CVE-2003-0209: DSA-297-1 Snort – integer overflow. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0209.

[35] Common Vulnerabilities and Exposures. CVE - CVE-2004-2652: Snort 2.3.0 – DecodeTCP Vulnerabilities. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0209.

[36] Common Vulnerabilities and Exposures. CVE - CVE-2009-3641: Snort 2.8.5 IPv6 Remote DoS. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3641.

[37] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009.

[38] ETSI. Network Functions Virtualization – Introductory White Paper. https://portal.etsi.org/NFV/NFV_White_Paper.pdf, 2012.

[39] Y. Go, E. Jeong, J. Won, Y. Kim, D. F. Kune, and K. Park. Gaining Control of Cellular Traffic Accounting by Spurious TCP Retransmission. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[40] S. Han, K. Jang, K. Park, and S. B. Moon. PacketShader: a GPU-Accelerated Software Router. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2010.

[41] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: a Highly-scalable Software-based Intrusion Detection System. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[42] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: cheap SSL acceleration with commodity processors. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[43] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[44] H. Jiang, G. Zhang, G. Xie, K. Salamatian, and L. Mathy. Scalable High-Performance Parallel Design for Network Intrusion Detection Systems on Many-Core Processors. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2013.

[45] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless Network Functions. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2015.

[46] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[47] H. Krawczyk. LFSR-based Hashing and Authentication. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 1994.

[48] R. Laufer, M. Gallo, D. Perino, and A. Nandugudi. CliMB: Enabling Network Function Composition with Click Middleboxes. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2016.

[49] B. Li, K. Tan, L. Luo, R. Luo, Y. Peng, N. Xu, Y. Xiong, P. Chen, Y. Xiong, and P. Cheng. ClickNP: Highly Flexible and High-performance Network Processing with Reconfigurable Hardware. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[50] Q. Li, M. Dong, and P. B. Godfrey. Halfback: Running Short Flows Quickly and Safely. In *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2015.

[51] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider. Enriching Network Security Analysis with Time Travel. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2008.

[52] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[53] Novak, Judy and Sturges, Steve. Target-Based TCP Stream Reassembly. *Sourcefire Inc.*, pages 1–23, 2007.

[54] S. Pathak and V. S. Pai. ModNet: A Modular Approach to Network Stack Extension. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[55] V. Paxson. Bro: A System for Detecting Network Intruders in Real-time. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 1998.

[56] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2014.

[57] L. Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[58] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX Systems Administration Conference (LISA)*, 1999.

[59] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[60] The Bro Project. Writing Bro Plugins. `https://www.bro.org/sphinx/devel/plugins.html`.

[61] The Snort Project. Snort Users Manual 2.9.5. pages 59–73, 2013.

[62] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[63] S. Woo, E. Jeong, S. Park, J. Lee, S. Ihm, and K. Park. Comparison of Caching Strategies in Modern Cellular Backhaul Networks. In *Proceeding of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2013.

Appendix A

/ monitoring socket creation/closure, scope setup (see Figure 2) */*

`int mtcp_socket(mctx_t mctx, int domain, int type, int protocol);`

— Create a socket. The socket can be either regular TCP socket (type = MOS_SOCK_STREAM),

— a TCP connection monitoring socket (*type* = MOS_SOCK_MONITOR_STREAM) or a raw packet monitoring socket (*type* = MOS_SOCK_MONITOR_RAW).

`int mtcp_bind_monitor_filter(mctx_t mctx, int sock, monitor_filter_t ft);`

— Bind a monitoring socket(*sock*) to a traffic filter (*ft*). *ft* limits the traffic monitoring scope in a BPF syntax.

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

— Retrieve the peer address information of a socket (*sock*).

— Depending on the size of *addrlen*, one can get either server-side or both server and client-side address information.

`int mtcp_close(mctx_t mctx, int sock);`

— Close a socket. Closing a monitoring socket does not terminate the connection but unregisters all flow events for the socket.

/ event manpulation (see Figure 2 and Section 3.2) */*

`event_t mtcp_define_event(event_t ev, filter_t filt, struct filter_arg *arg);`

— Define a new event with a base event (*ev*) and a filter function (*filt*) with a filter argument (*arg*).

`event_t mtcp_alloc_event(event_t parent_event);` /

`int mtcp_raise_event(mctx_t mctx, event_t child_event);`

— Define a follow-up event for a filter that can trigger multiple (child) events.

— Raise a child event for a multi-event filter.

`int mtcp_register_callback(mctx_t mctx, int sock, event_t ev, int hook, callback_t cb);`

— Register (or unregister) an event handler (or a callback function) (*cb*) for an event (*ev*) in the context of a monitoring socket (*sock*).

— *hook* specifies when the event should be fired. It can be fired after updating packet sender's TCP context (MOS_HK_SND)

— or after updating packet receiver's TCP context (MOS_HK_RCV) or MOS_NULL, which does not care.

/ current packet information and modification (see Figure 4) */*

`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);`

— mtcp_getlastpkt() retrieves the information of the last packet of a flow (*sock* and *side*).

— mtcp_setlastpkt() updates the last packet with *data* at *offset* bytes from an anchor for *datalen* bytes.

— *option* is the anchor for *offset*. It can be one of MOS_ETH_HDR, MOS_IP_HDR, MOS_TCP_HDR or MOS_TCP_PAYLOAD.

`int mtcp_sendpkt(mctx_t mctx, int sock, const struct pkt_info *pkt);`

— Send a self-constructed TCP packet (*pkt*) for a given flow (*sock*).

/ flow-reassembled buffer reading (see Figure 4 and Section 4.3) */*

`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);`

— Read the data in a TCP receive buffer of *sock*. *side* specifies either client or server side.

— mtcp_ppeek() is identical to mtcp_peek() except that it reads the data from a specific offset(*seq_off*) from the initial sequence number.

/ TCP flow monitoring and manipulation (see Figures 4 & 7) */*

`int mtcp_getsockopt(mctx_t mctx, int sock, int level, int optname, void *optval, socklen_t *optlen);` /

`int mtcp_setsockopt(mctx_t mctx, int sock, int level, int optname, void *optval, socklen_t optlen);`

— Retrieve (or set) socket-level attributes.

/ per-flow user-level metadata management */*

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

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

— Retrieve (or store) a user-specified pointer (*uctx*) associated with a socket (*sock*).

/ initialization routines */*

`mctx_t mtcp_create_context(int cpu);` /

`int mtcp_destroy_context(mctx_t mctx);`

— Create (or destroy) a mOS context and associate it with a *cpu* id.

`int mtcp_init(const char *mos_conf_fname);`

— Initialize mOS with the attributes in a configuration file (*mos_conf_fname*). Called one time per process.

Table 4: The current mOS networking API. More detail is found in mOS manual pages: `http://mos.kaist.edu/index_man.html`.

Appendix B

```c
// count # of packets in each TCP flow
static void // callback for MOS_ON_PKT_IN
OnFlowPkt(mctx_t m, int sock, int side, event_t event,
          struct filter_arg *arg)
{
   if (side == MOS_SIDE_CLI)
      g_pktcnt[sock]++;
}

// count # of packet retransmissions in each TCP flow
static void // callback for MOS_ON_REXMIT
OnRexmitPkt(mctx_t m, int sock, int side, event_t event,
            struct filter_arg *arg)
{
   g_rexmit_cnt[sock]++;
}

// count # of client-initated TCP connection teardown
static void // callback for MOS_ON_TCP_STATE_CHANGE
OnTCPStateChange(mctx_t m, int sock, int side, event_t event,
                 struct filter_arg *arg)
{
   if (side == MOS_SIDE_CLI) {
      int state; socklent_t len = sizeof(state);
      mtcp_getsockopt(m, sock, SOL_MONSOCKET,
                      MOS_TCP_STATE_CLI, &state, &len);
      if (state == TCP_FIN_WAIT_1)
         g_cli_term++;
   }
}

// print the statistics and reset counters
// count total # of completed flows
static void // callback for MOS_ON_TCP_CONN_END
OnFlowEnd(mctx_t m, int sock, int side, event_t event,
          struct filter_arg *arg)
{
   if (sock != MOS_SIDE_CLI) return;

   TRACE_LOG("TCP flow (sock=%d) had %d packets, rexmit: %d\n",
             sock, g_pktcnt[sock], g_rexmit_cnt[sock]);
   g_pktcnt[sock] = 0; g_rexmit_cnt[sock] = 0;
   g_total_flows++;
}
```

Figure 15: Code examples with mOS built-in event handlers. With only 2∼5 lines of code, one can gather various flow-level statistics in a middlebox.