US 20190303562A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2019/0303562 A1**

MASPUTRA et al. (43) **Pub. Date: Oct. 3, 2019**

(54) **METHODS AND APPARATUS FOR SECURE OPERATION OF USER SPACE COMMUNICATION STACKS**

(71) Applicant: **Apple Inc.**, Cupertino, CA (US)

(72) Inventors: **CAHYA ADIANSYAH MASPUTRA**, San Jose, CA (US); **SANDEEP NAIR**, San Joe, CA (US); **THOMAS FRANCIS PAULY**, Cupertino, CA (US); **DIETER WILFRED SIEGMUND**, Cuptertino, CA (US); **WEI SHEN**, Cupertino, CA (US); **OLIVIER MARDINIAN**, Cupertino, CA (US); **DARRIN JEWELL**, Woodside, CA (US)

(21) Appl. No.: **16/368,396**

(22) Filed: **Mar. 28, 2019**

Related U.S. Application Data

(60) Provisional application No. 62/649,509, filed on Mar. 28, 2018.

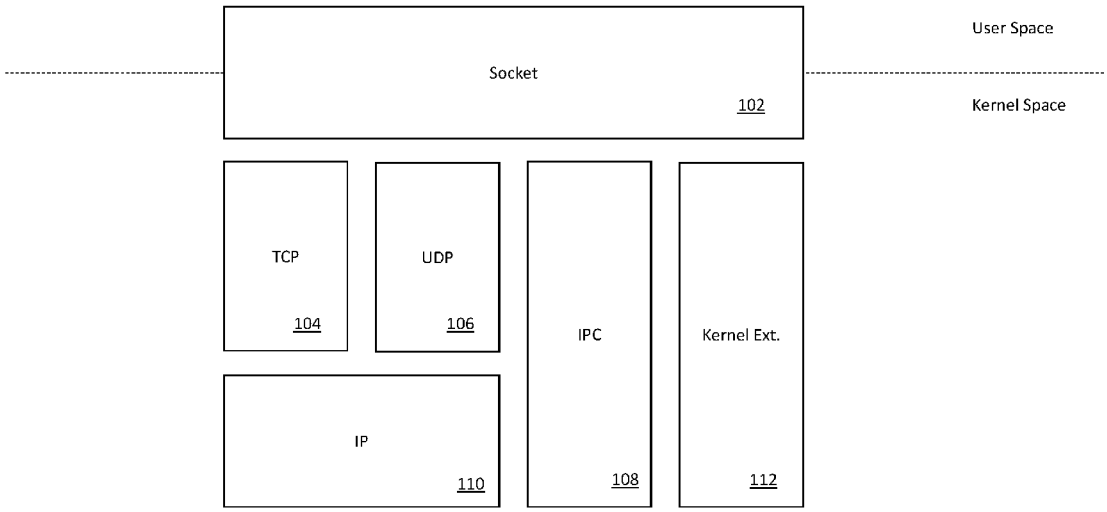
Publication Classification

(51) **Int. Cl.**
G06F 21/52 (2006.01)
G06F 16/23 (2006.01)

(52) **U.S. Cl.**
CPC ***G06F 21/52*** (2013.01); *G06F 2221/034* (2013.01); ***G06F 16/2365*** (2019.01)

(57) **ABSTRACT**

Methods and apparatus for efficient data transfer within a user space network stack. Unlike prior art monolithic networking stacks, the exemplary networking stack architecture described hereinafter includes various components that span multiple domains (both in-kernel, and non-kernel). For example, unlike traditional "socket" based communication, disclosed embodiments can transfer data directly between the kernel and user space domains. Direct transfer reduces the per-byte and per-packet costs relative to socket based communication. A user space networking stack is disclosed that enables extensible, cross-platform-capable, user space control of the networking protocol stack functionality. The user space networking stack facilitates tighter integration between the protocol layers (including TLS) and the application or daemon. Exemplary systems can support multiple networking protocol stack instances (including an in-kernel traditional network stack).



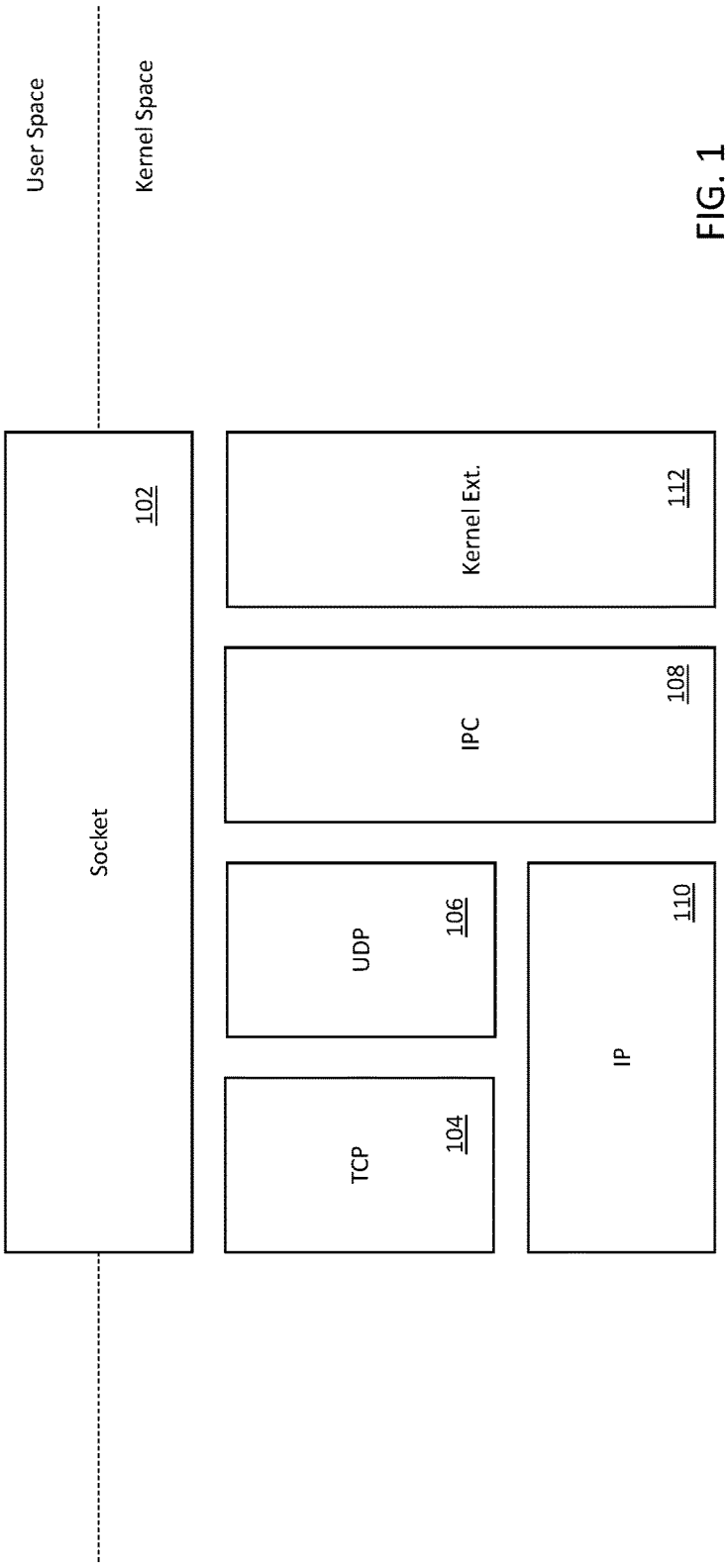

User Space
Kernel Space
Socket
102
TCP
104
UDP
106
IP
110
IPC
108
Kernel Ext.
112

FIG. 1

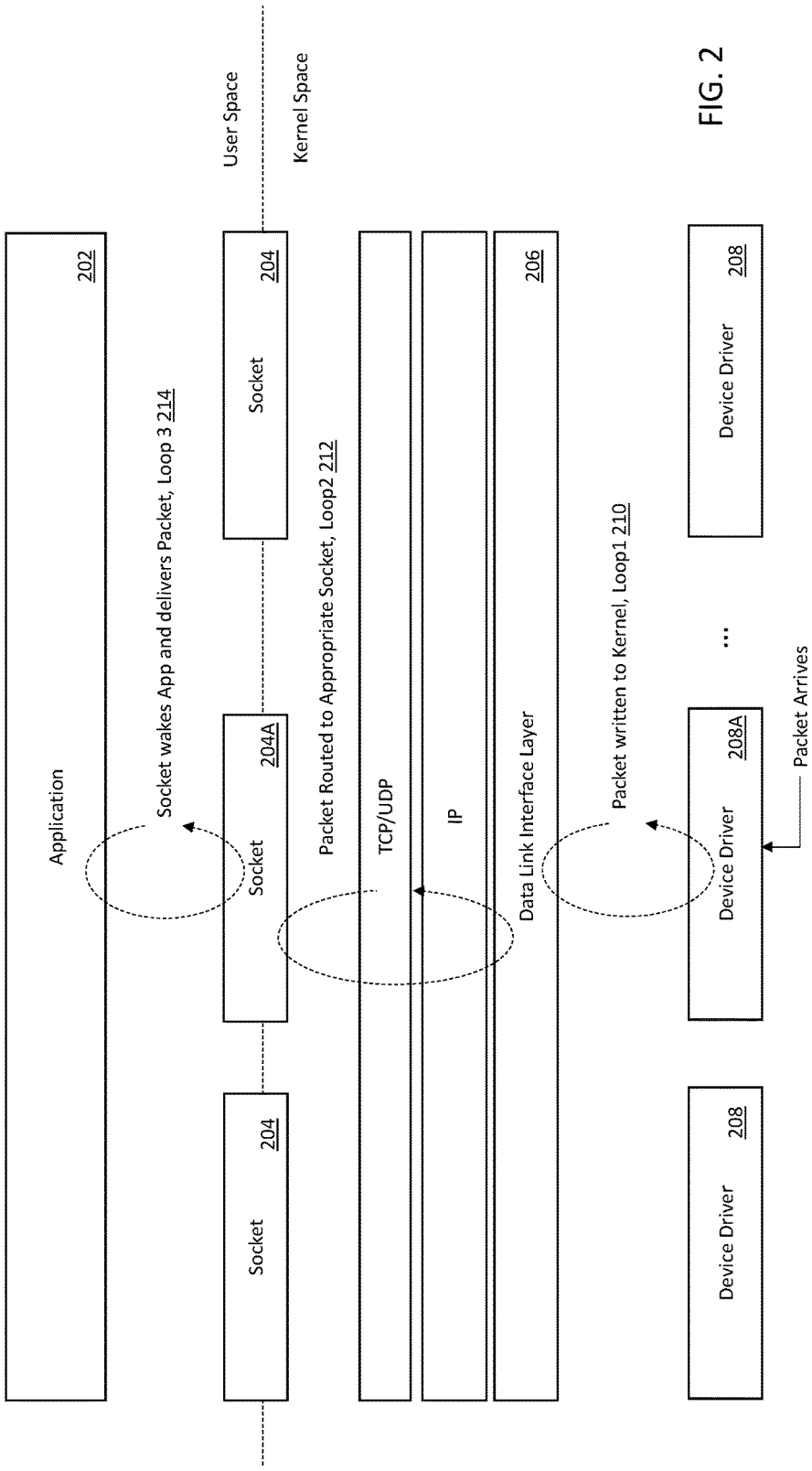

User Space
Kernel Space
Application
202
Socket
204
Socket
204A
Socket
204
Socket wakes App and delivers Packet, Loop 3 214
Packet Routed to Appropriate Socket, Loop2 212
TCP/UDP
IP
Data Link Interface Layer
206
Packet written to Kernel, Loop1 210
Device Driver
208
Device Driver
208A
Device Driver
208
...
Packet Arrives

FIG. 2

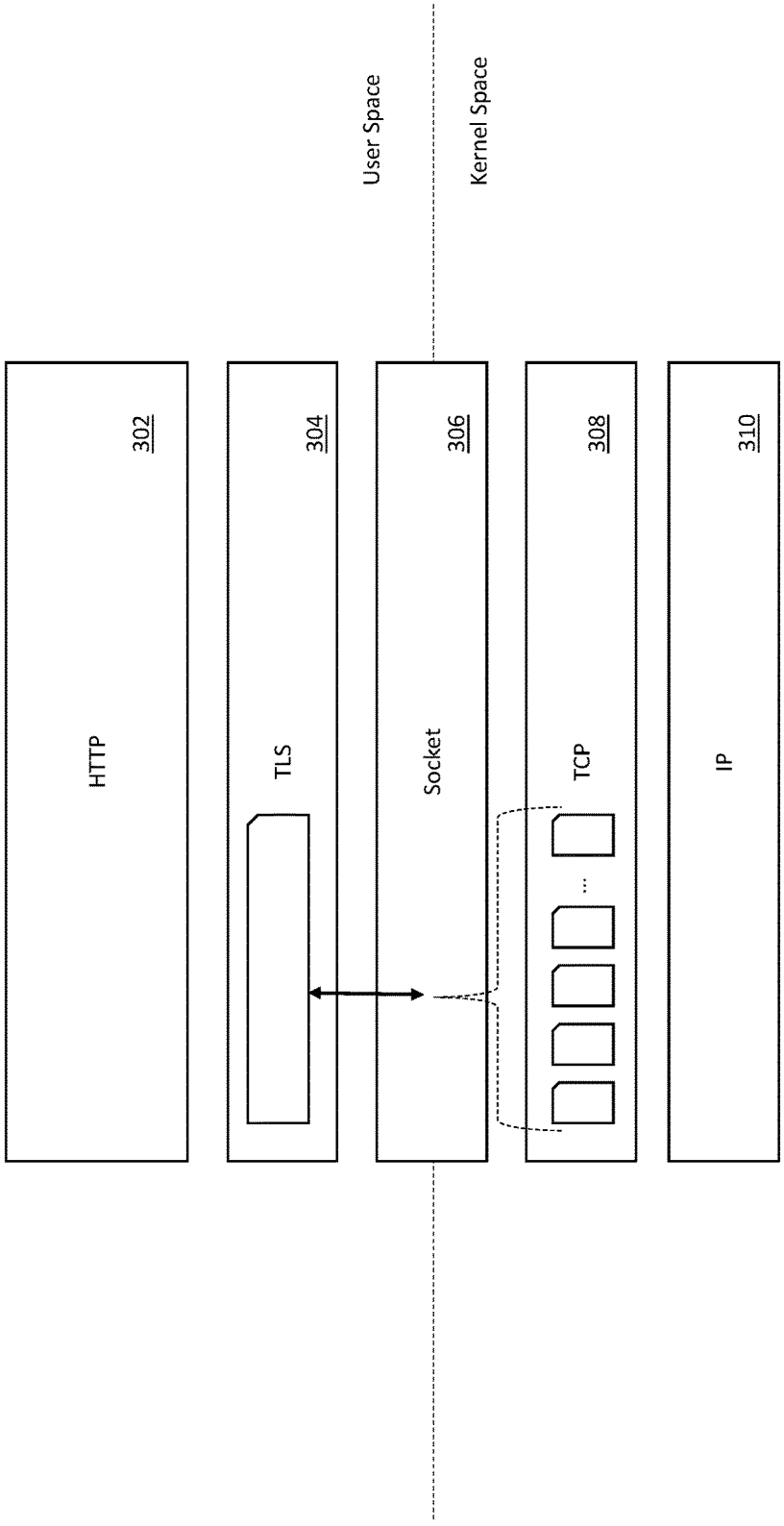

User Space
Kernel Space
HTTP
302
TLS
304
Socket
306
TCP
308
IP
310

FIG. 3

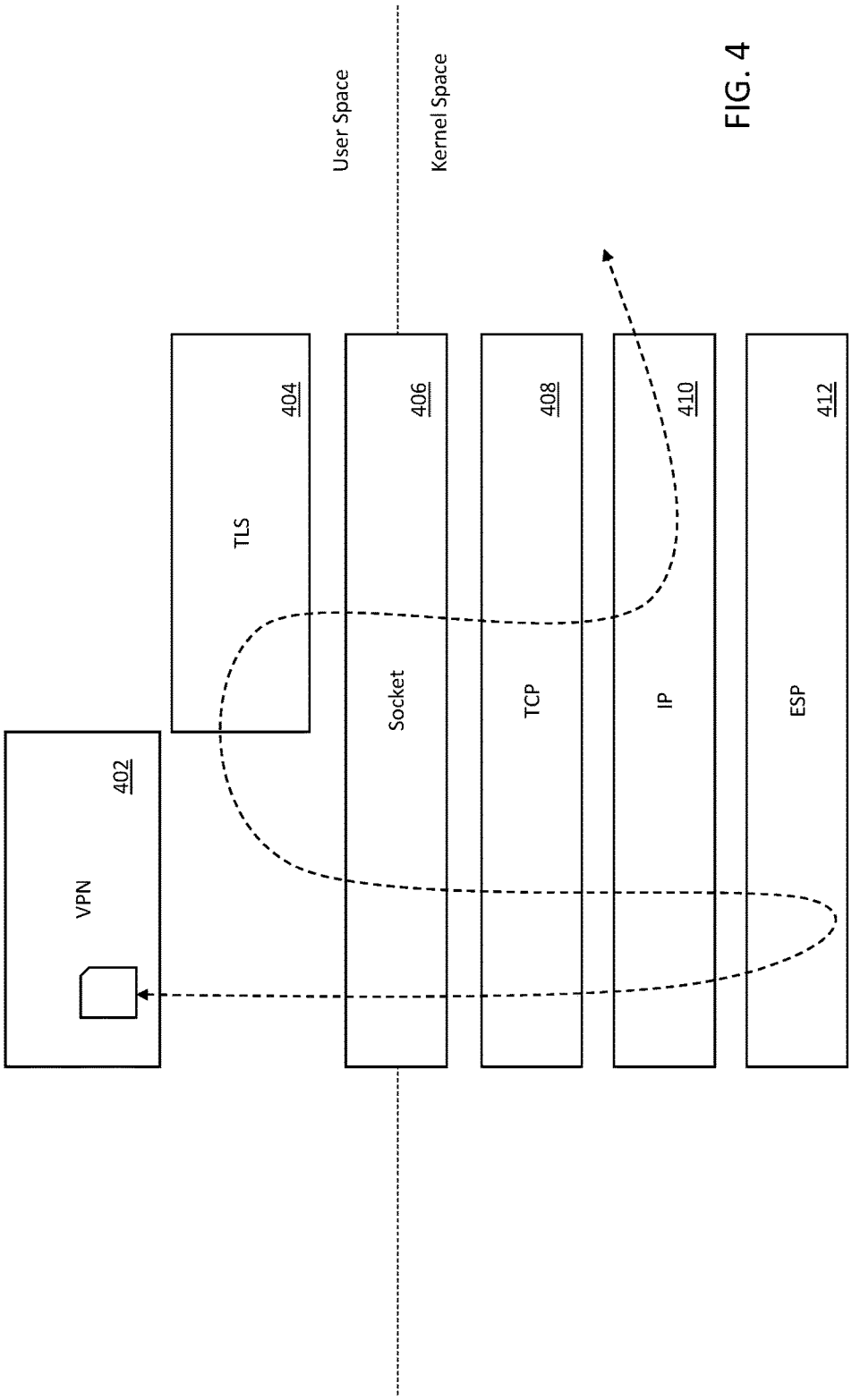

User Space
Kernel Space
VPN
402
TLS
404
Socket
406
TCP
408
IP
410
ESP
412

FIG. 4

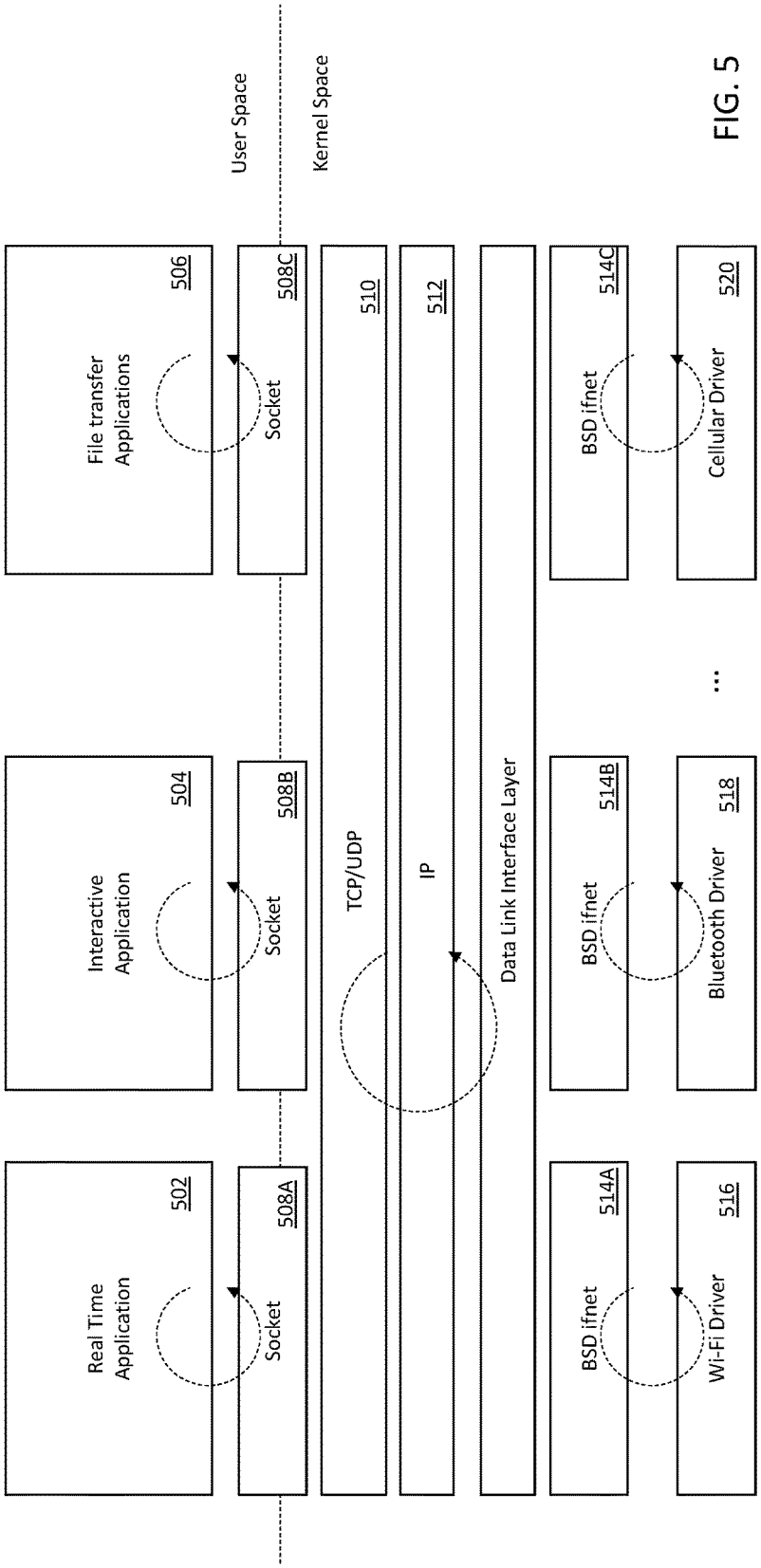

User Space
Kernel Space
Real Time Application
502
Interactive Application
504
File transfer Applications
506
Socket
508A
Socket
508B
Socket
508C
TCP/UDP
510
IP
512
Data Link Interface Layer
BSD ifnet
514A
BSD ifnet
514B
BSD ifnet
514C
Wi-Fi Driver
516
Bluetooth Driver
518
Cellular Driver
520
...

FIG. 5

FIG. 6

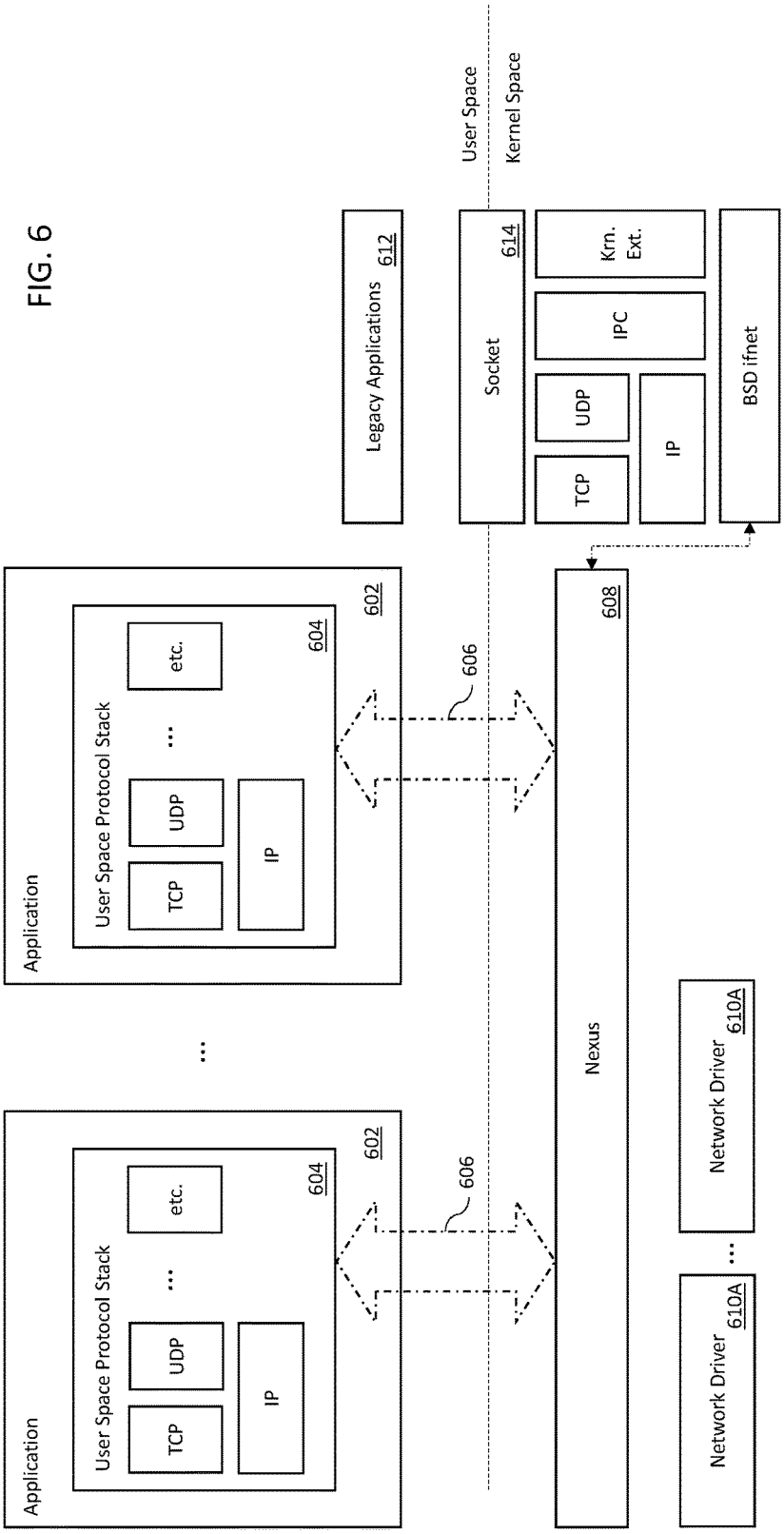
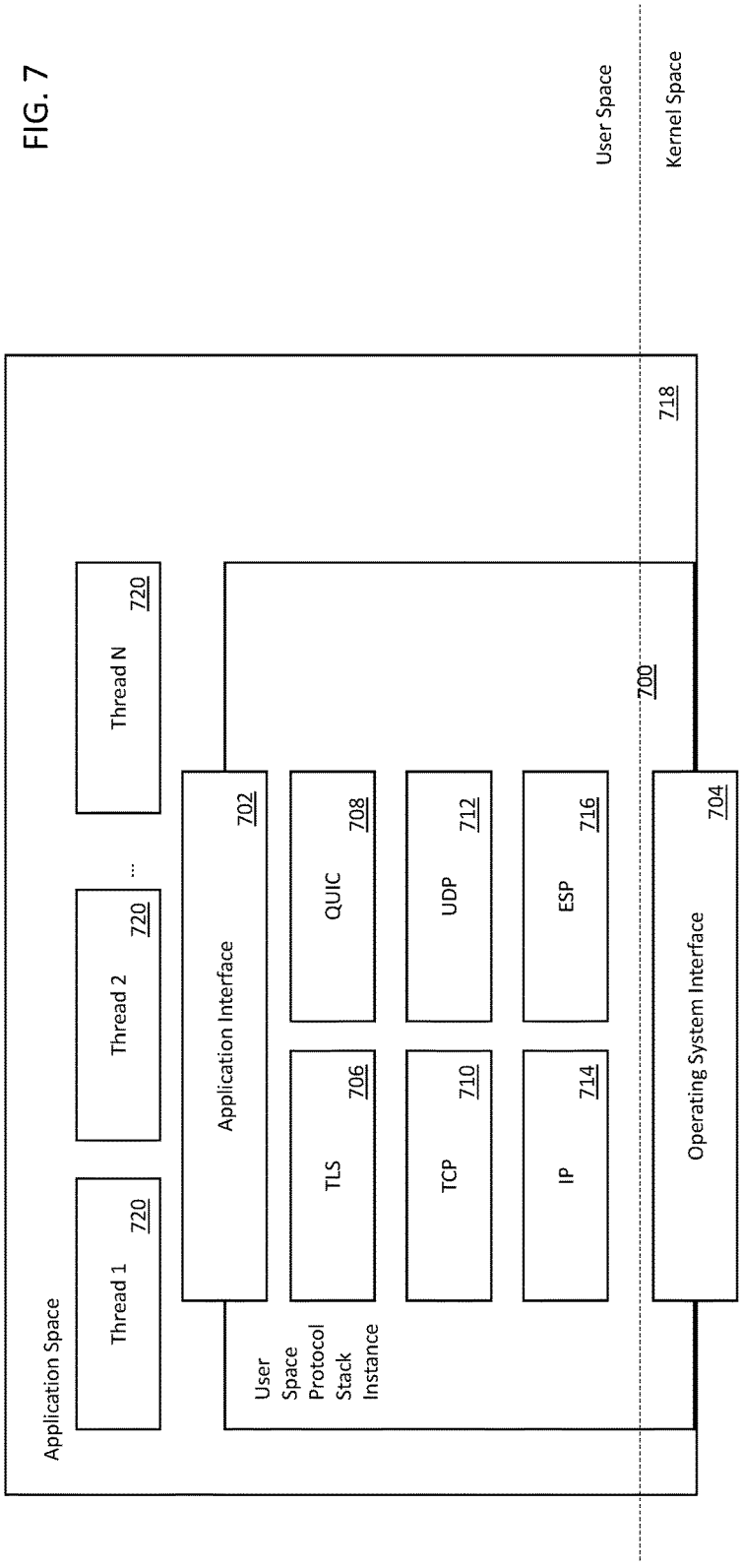

User Space
Kernel Space
Legacy Applications 612
Socket 614
TCP
UDP
IP
IPC
Krn. Ext.
BSD ifnet
Application
User Space Protocol Stack
TCP
UDP
IP
etc.
602
604
606
Nexus 608
Network Driver 610A

FIG. 7


User Space
Kernel Space
718
700
Application Space
Thread 1 720
Thread 2 720
...
Thread N 720
User Space Protocol Stack Instance
Application Interface 702
TLS 706
QUIC 708
TCP 710
UDP 712
IP 714
ESP 716
Operating System Interface 704

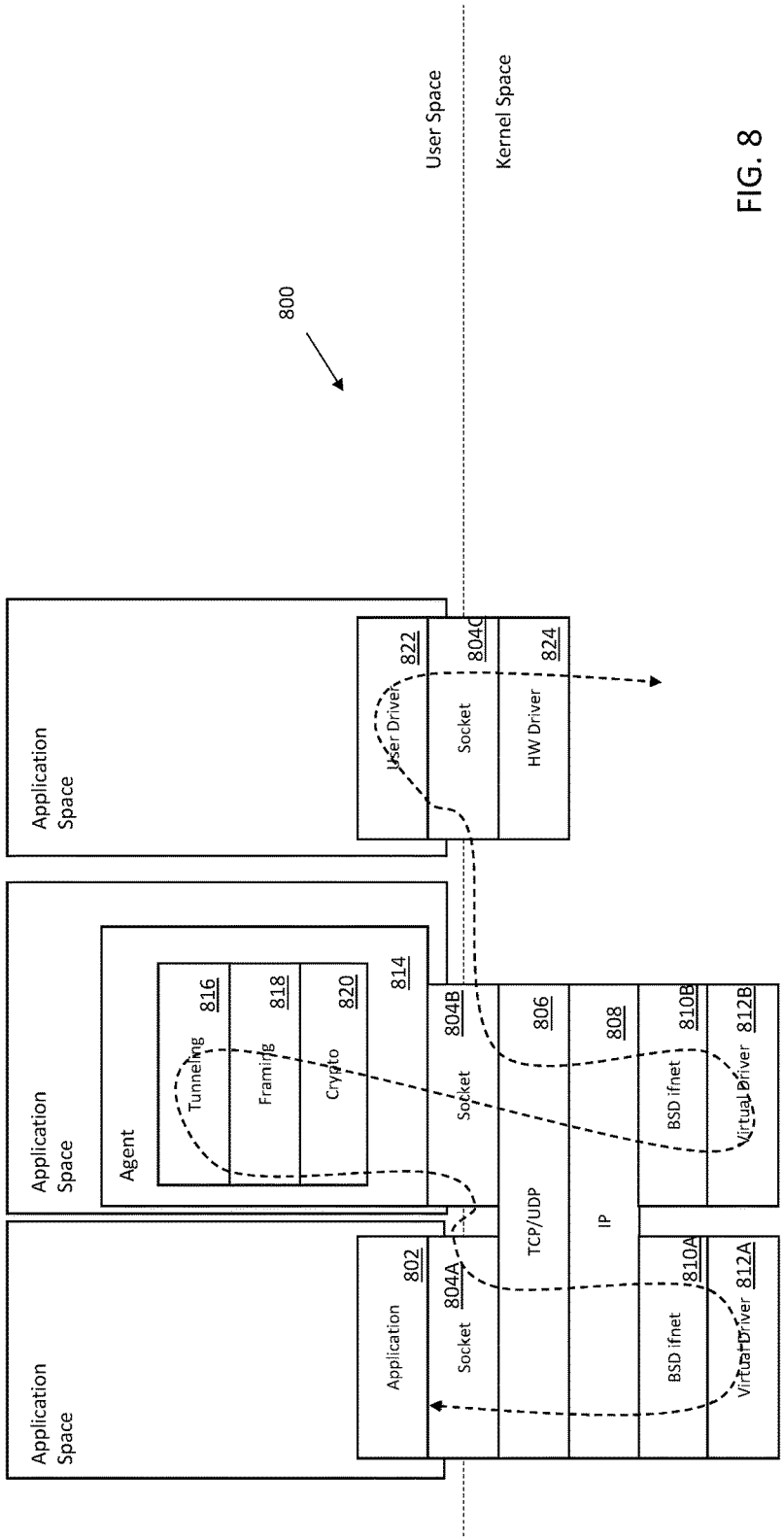

800
User Space
Kernel Space
Application Space
Application
802
Socket
804A
Application Space
Agent
814
Tunneling
816
Framing
818
Crypto
820
Socket
804B
Application Space
User Driver
822
Socket
804C
HW Driver
824
TCP/UDP
806
IP
808
BSD ifnet
810A
Virtual Driver
812A
BSD ifnet
810B
Virtual Driver
812B

FIG. 8

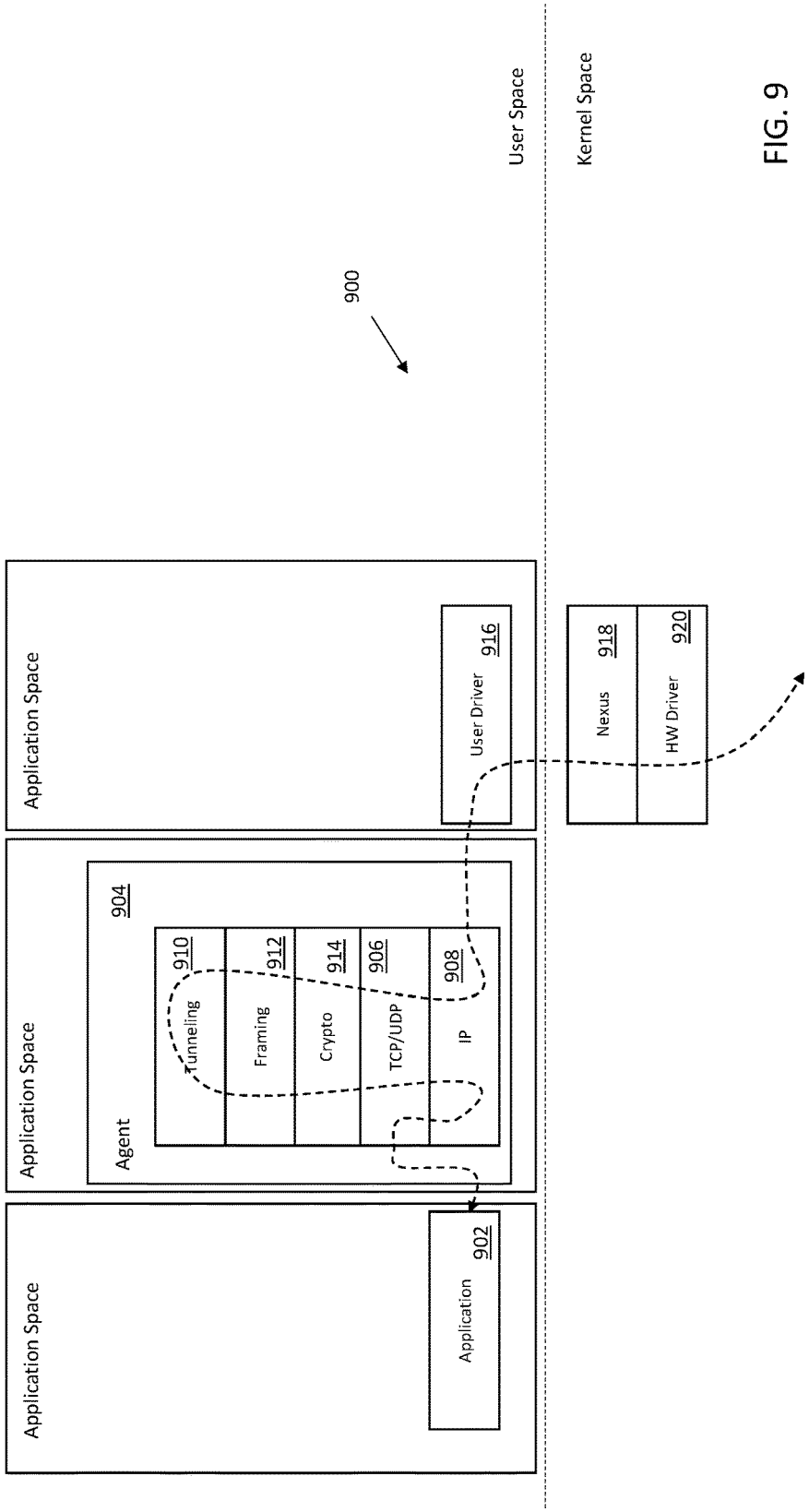

900
User Space
Kernel Space
Application Space
User Driver
916
Nexus
918
HW Driver
920
Application Space
Agent
904
Tunneling
910
Framing
912
Crypto
914
TCP/UDP
906
IP
908
Application Space
Application
902

FIG. 9

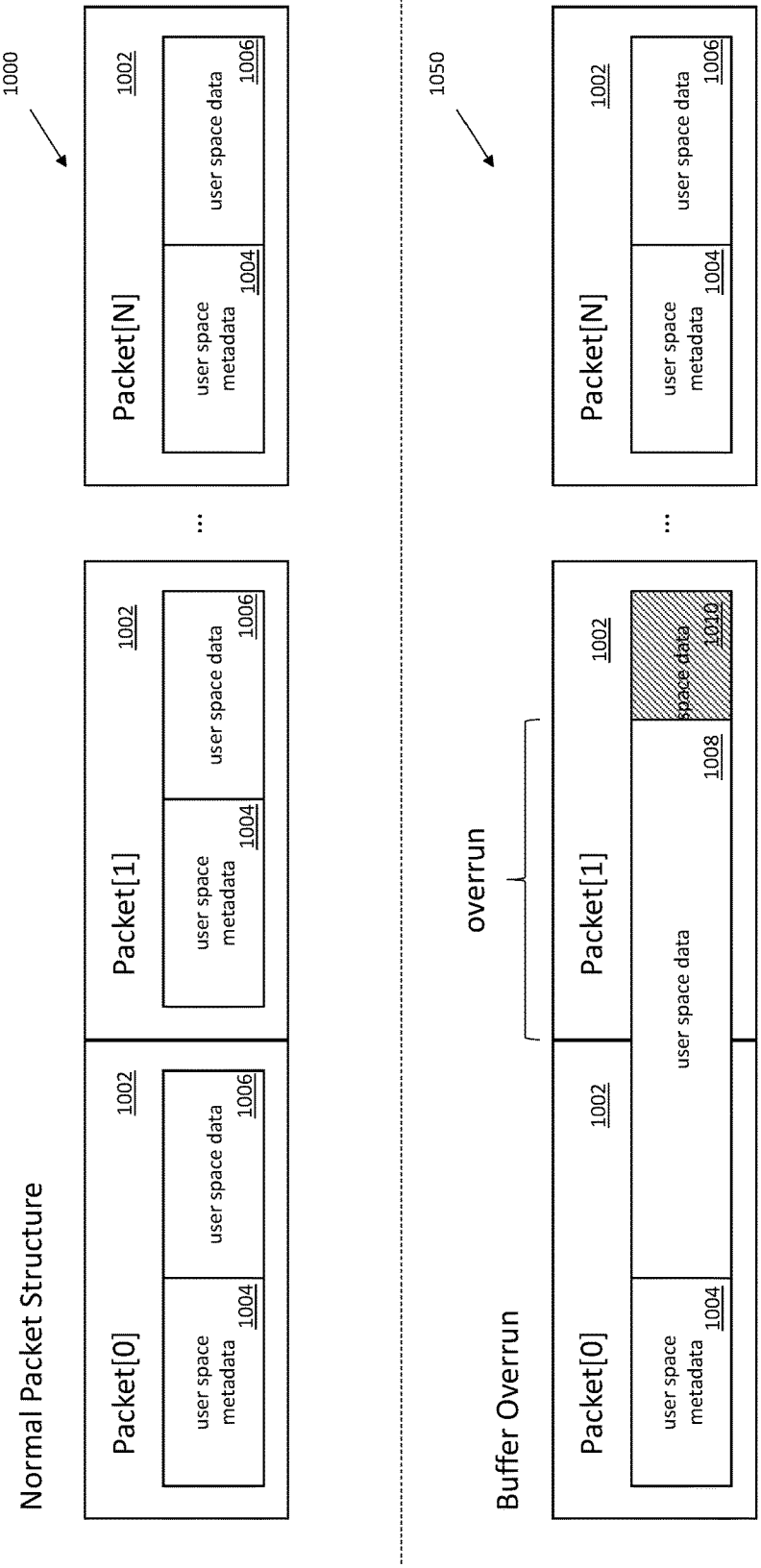

Normal Packet Structure
1000
Packet[0]
1002
user space metadata
1004
user space data
1006
Packet[1]
1002
user space metadata
1004
user space data
1006
...
Packet[N]
1002
user space metadata
1004
user space data
1006
Buffer Overrun
1050
overrun
Packet[0]
1002
user space metadata
1004
user space data
1008
Packet[1]
1002
space data
1010
...
Packet[N]
1002
user space metadata
1004
user space data
1006

FIG. 10

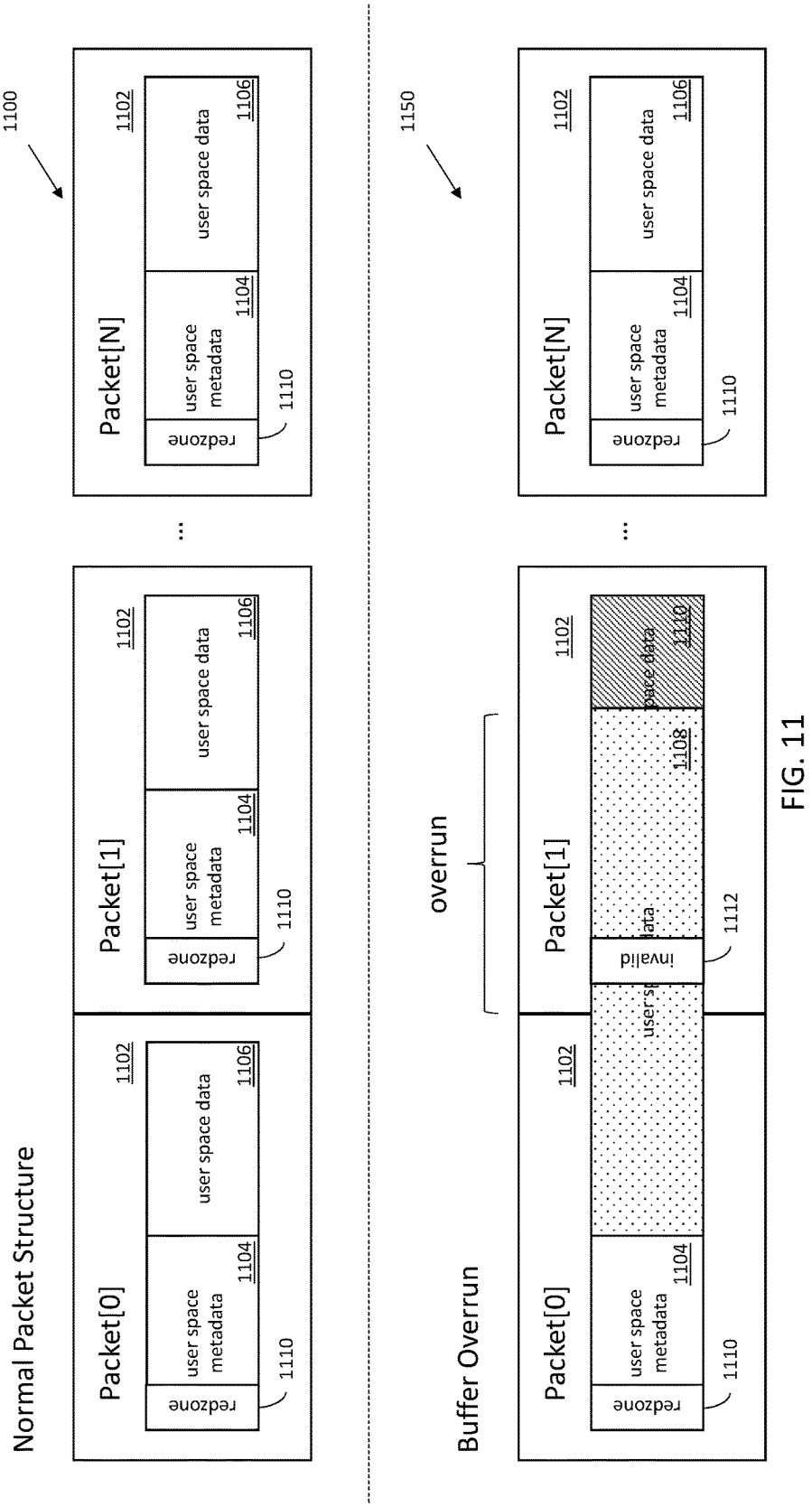

Normal Packet Structure
1100
Packet[0]
1102
redzone
user space metadata
1104
user space data
1106
1110
Packet[1]
Packet[N]
Buffer Overrun
1150
overrun
invalid
1108
1112

FIG. 11

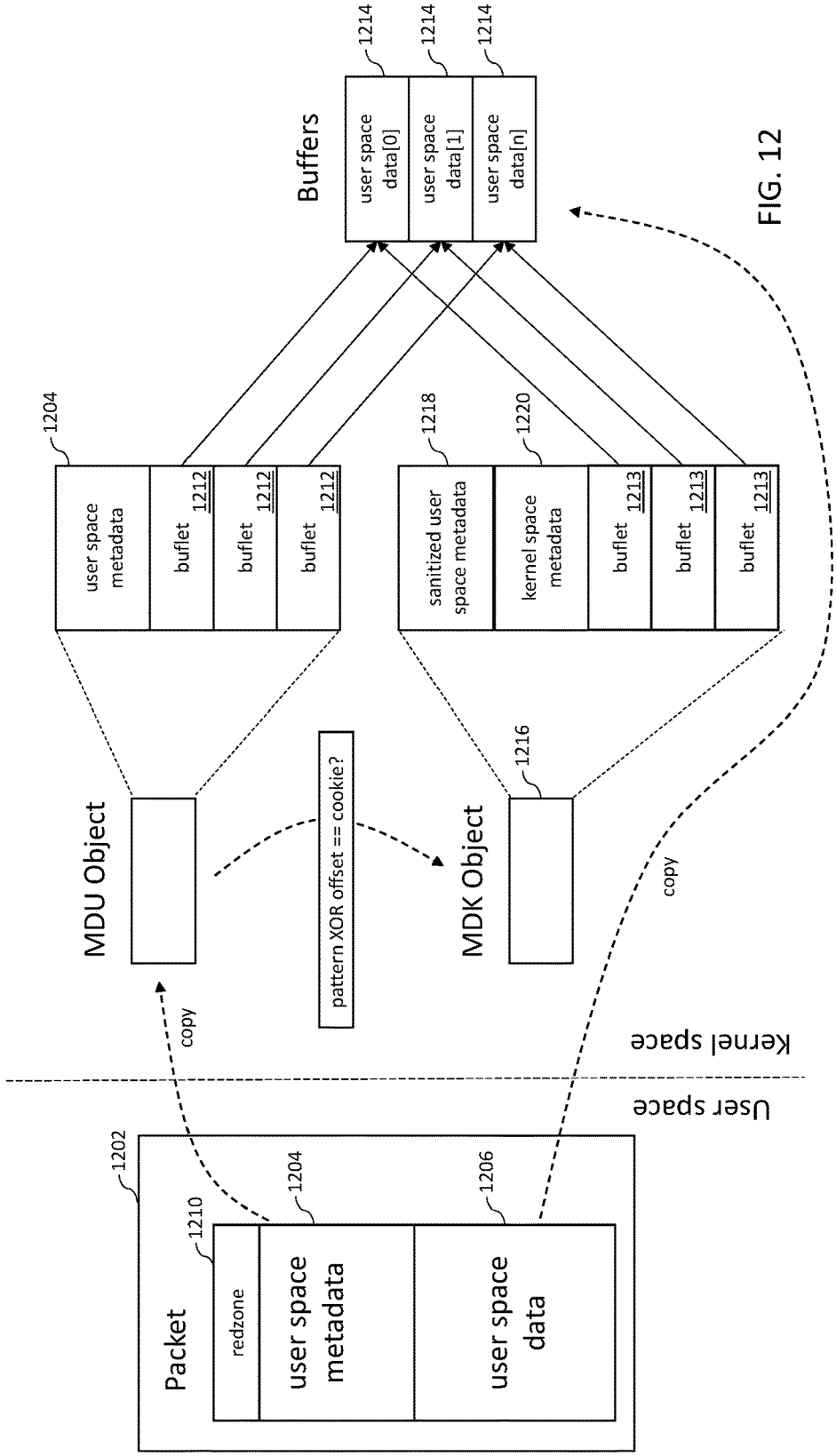

Buffers
1214
1214
1214
user space data[0]
user space data[1]
user space data[n]
1204
1218
1220
user space metadata
buflet 1212
buflet 1212
buflet 1212
sanitized user space metadata
kernel space metadata
buflet 1213
buflet 1213
buflet 1213
MDU Object
pattern XOR offset == cookie?
MDK Object
1216
copy
copy
Kernel space
User space
1202
1204
1206
1210
Packet
redzone
user space metadata
user space data

FIG. 12

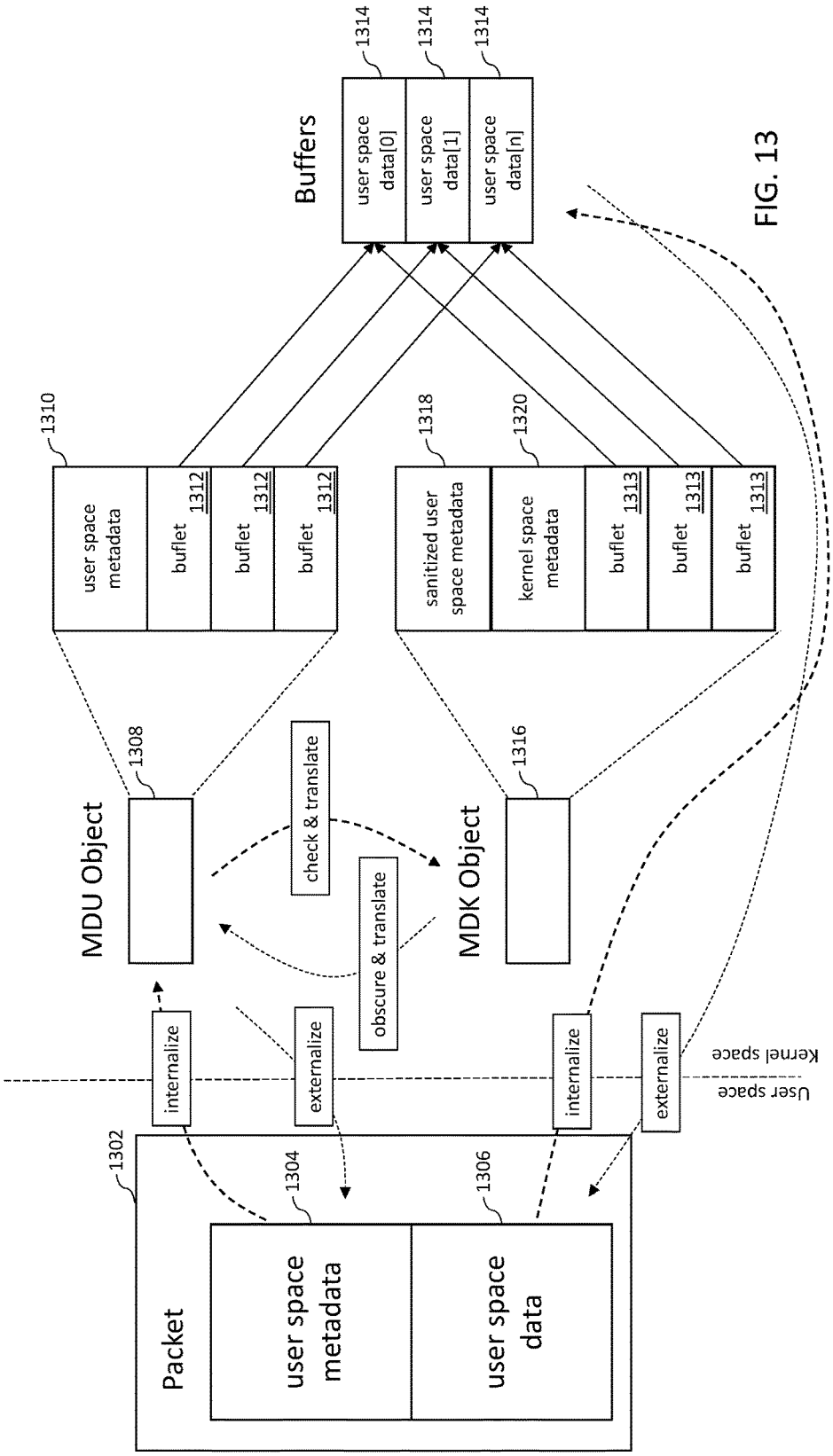

Buffers
1314
1314
1314
user space data[0]
user space data[1]
user space data[n]
1310
1318
1320
user space metadata
buflet 1312
buflet 1312
buflet 1312
sanitized user space metadata
kernel space metadata
buflet 1313
buflet 1313
buflet 1313
MDU Object
1308
MDK Object
1316
check & translate
obscure & translate
internalize
externalize
internalize
externalize
User space
Kernel space
1302
Packet
1304
user space metadata
1306
user space data

FIG. 13

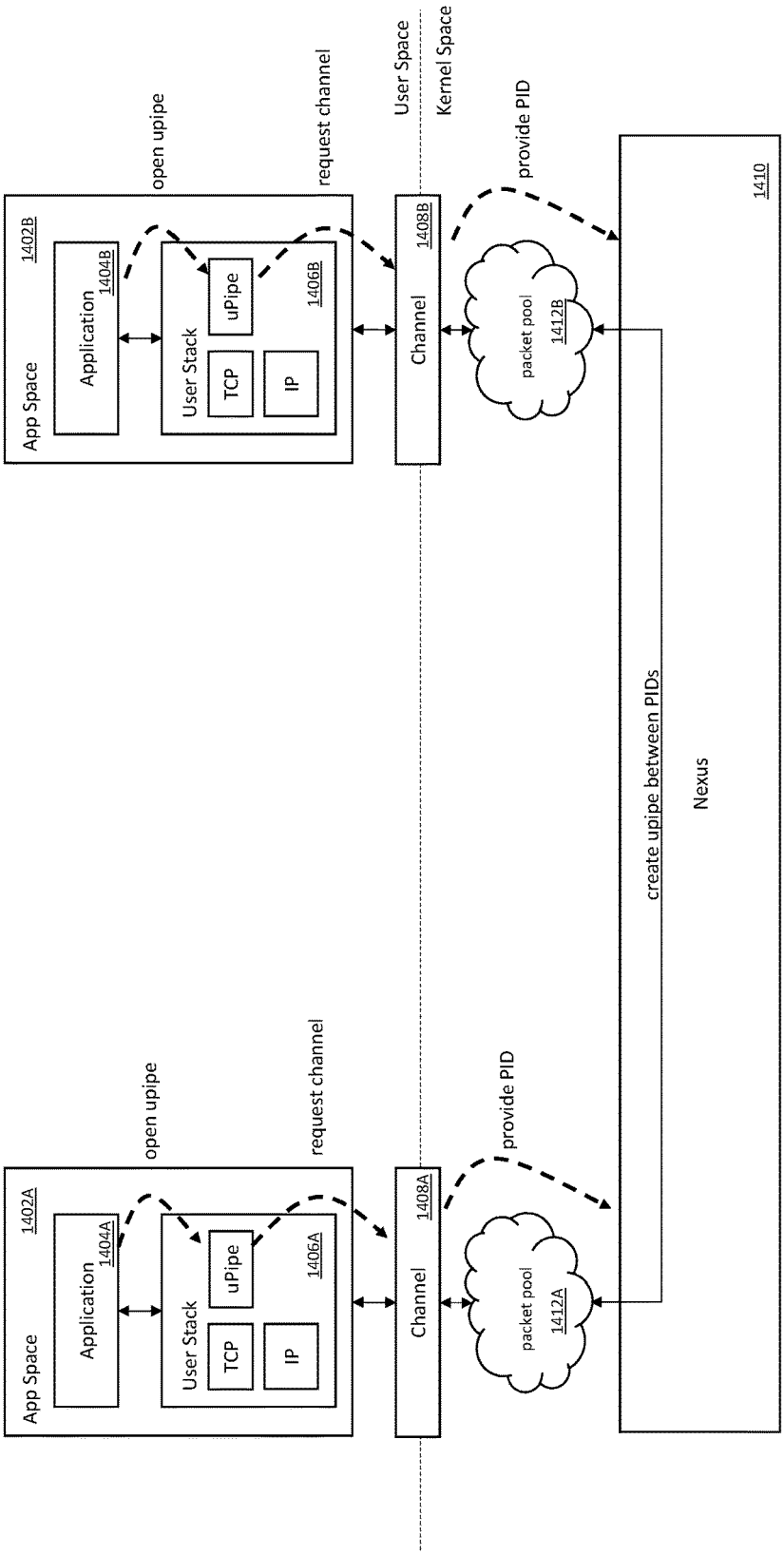

App Space
1402B
Application
1404B
User Stack
uPipe
TCP
IP
1406B
open upipe
request channel
User Space
Kernel Space
Channel
1408B
provide PID
packet pool
1412B
1410
create upipe between PIDs
Nexus
App Space
1402A
Application
1404A
User Stack
uPipe
TCP
IP
1406A
open upipe
request channel
Channel
1408A
provide PID
packet pool
1412A

FIG. 14

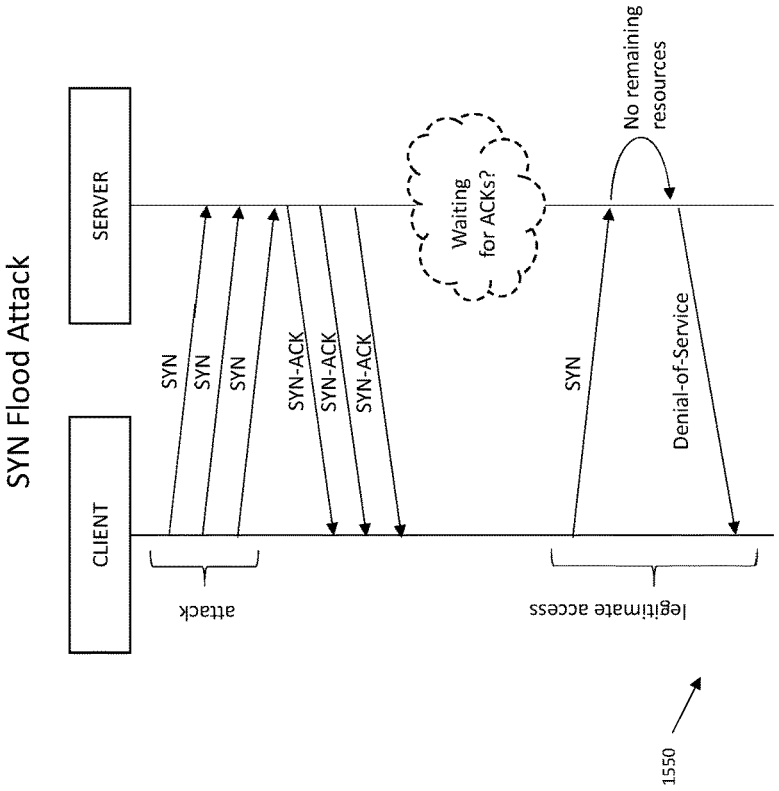


FIG. 15B

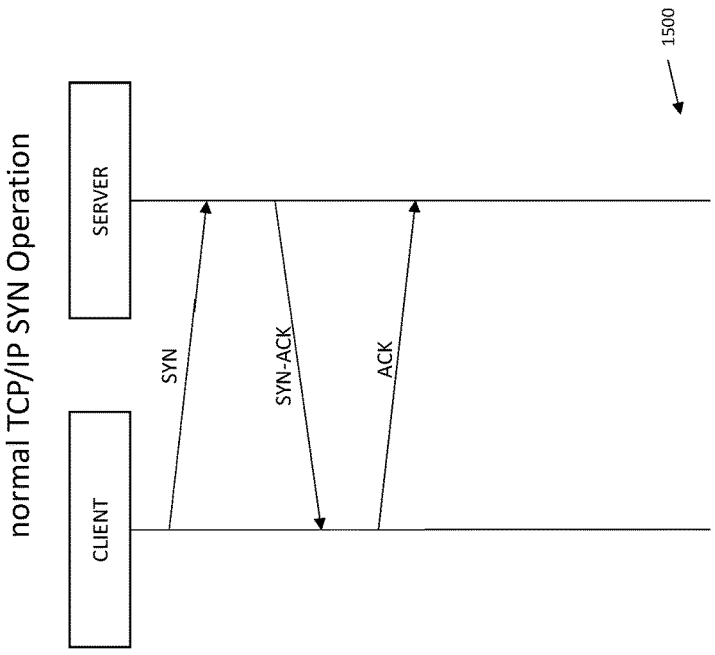


FIG. 15A

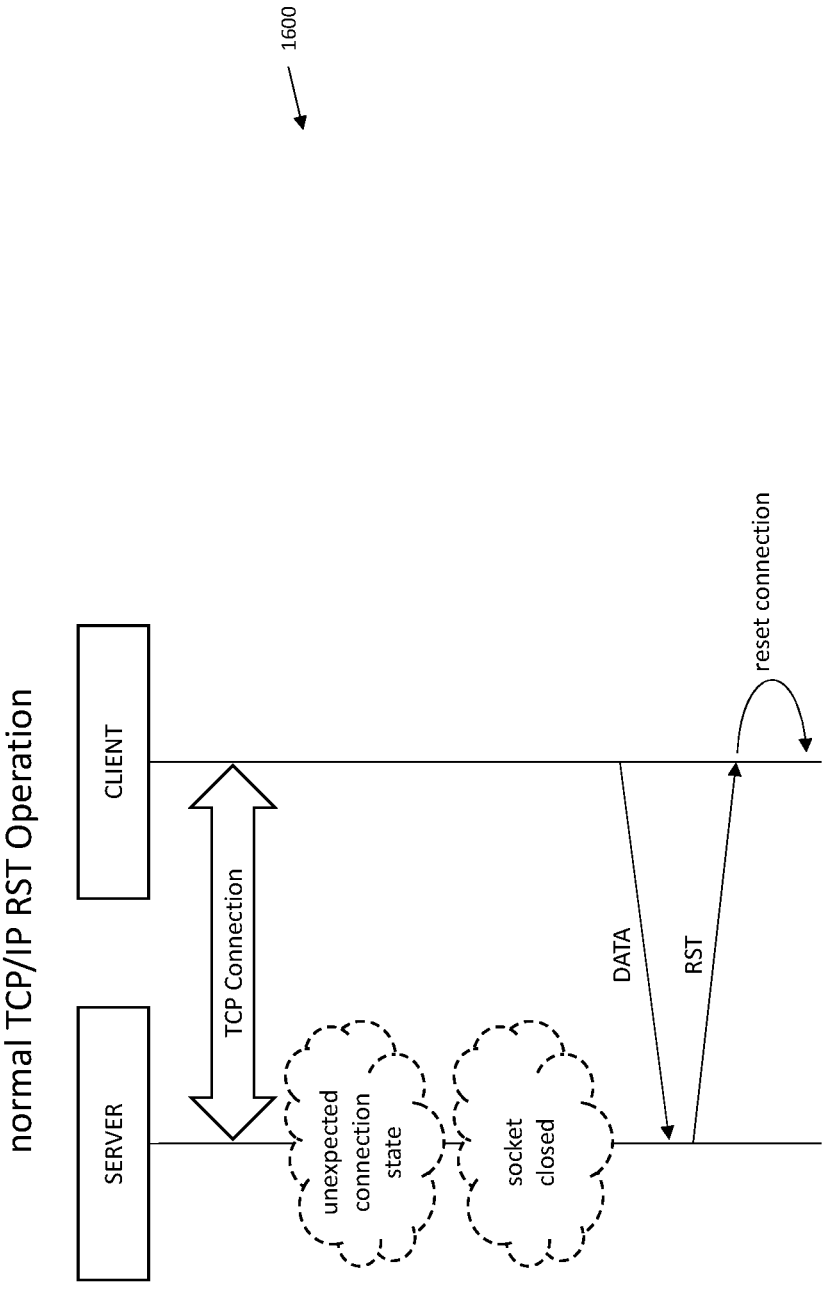

1600
normal TCP/IP RST Operation
CLIENT
SERVER
TCP Connection
unexpected connection state
socket closed
DATA
RST
reset connection

FIG. 16

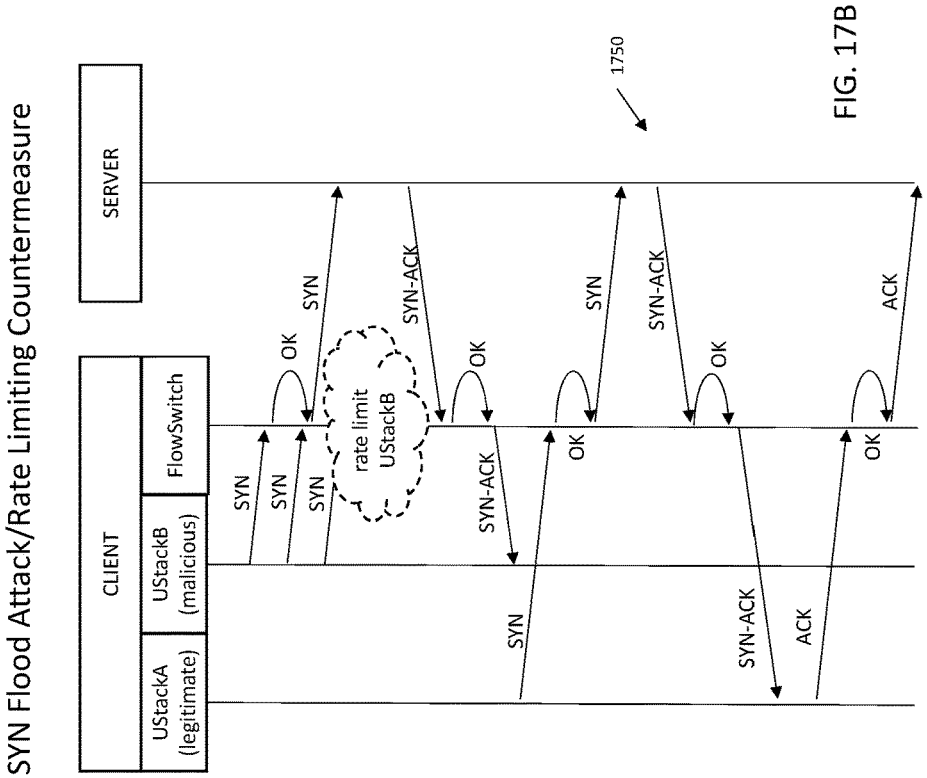


FIG. 17B

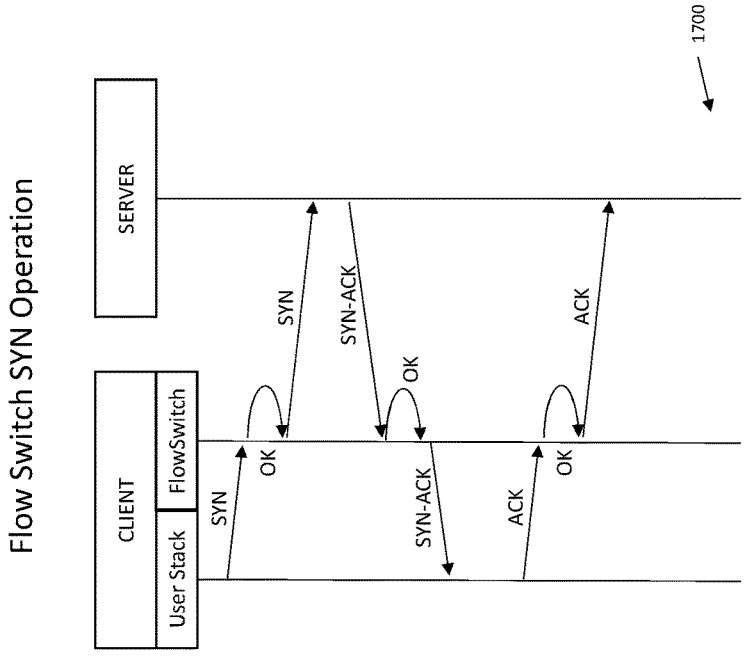


FIG. 17A

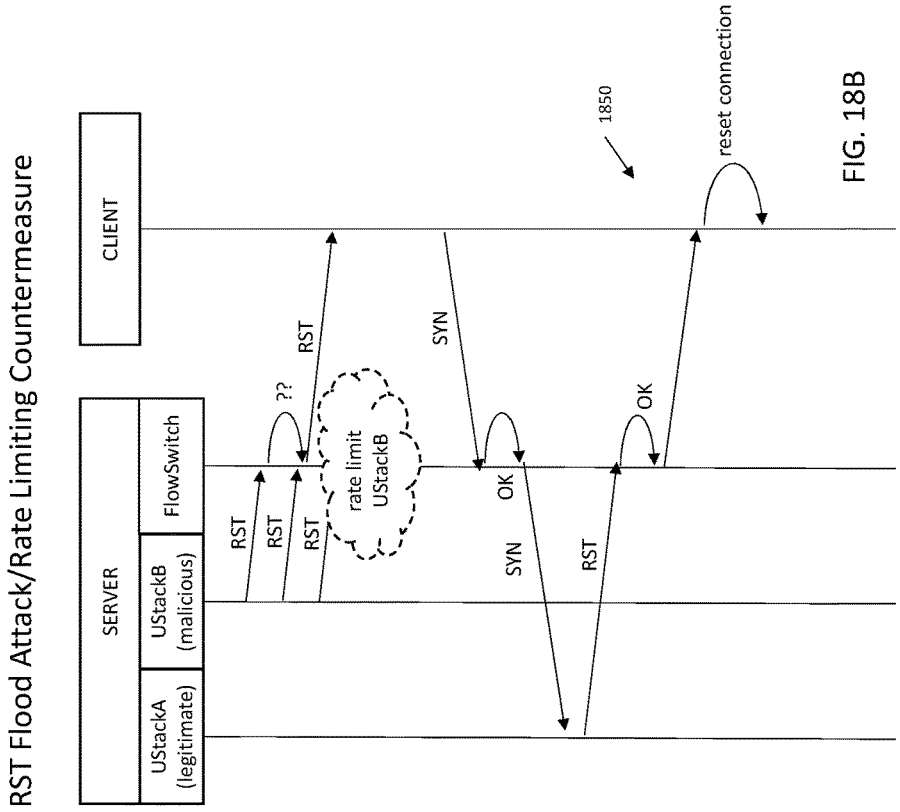


FIG. 18B

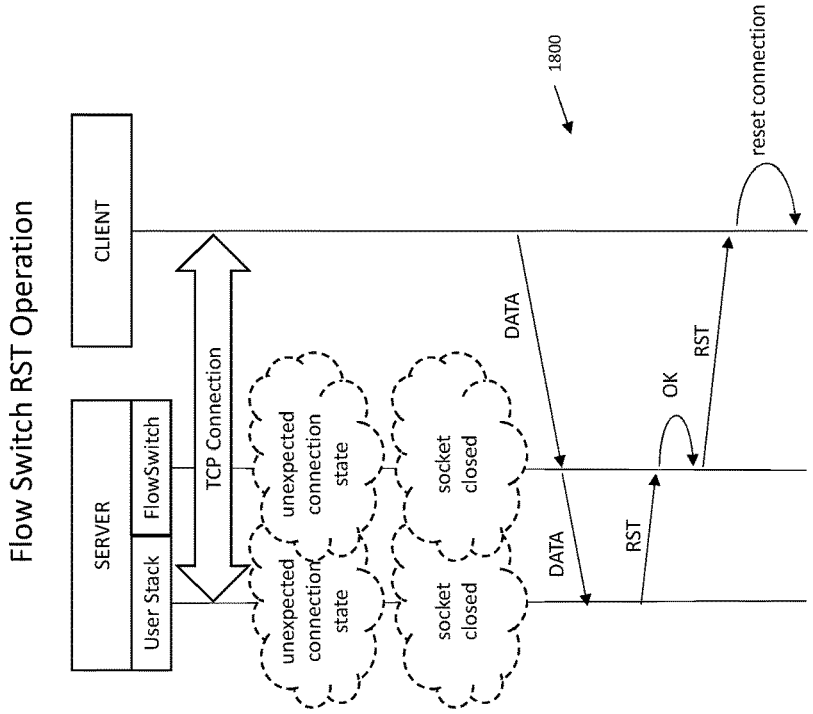


FIG. 18A

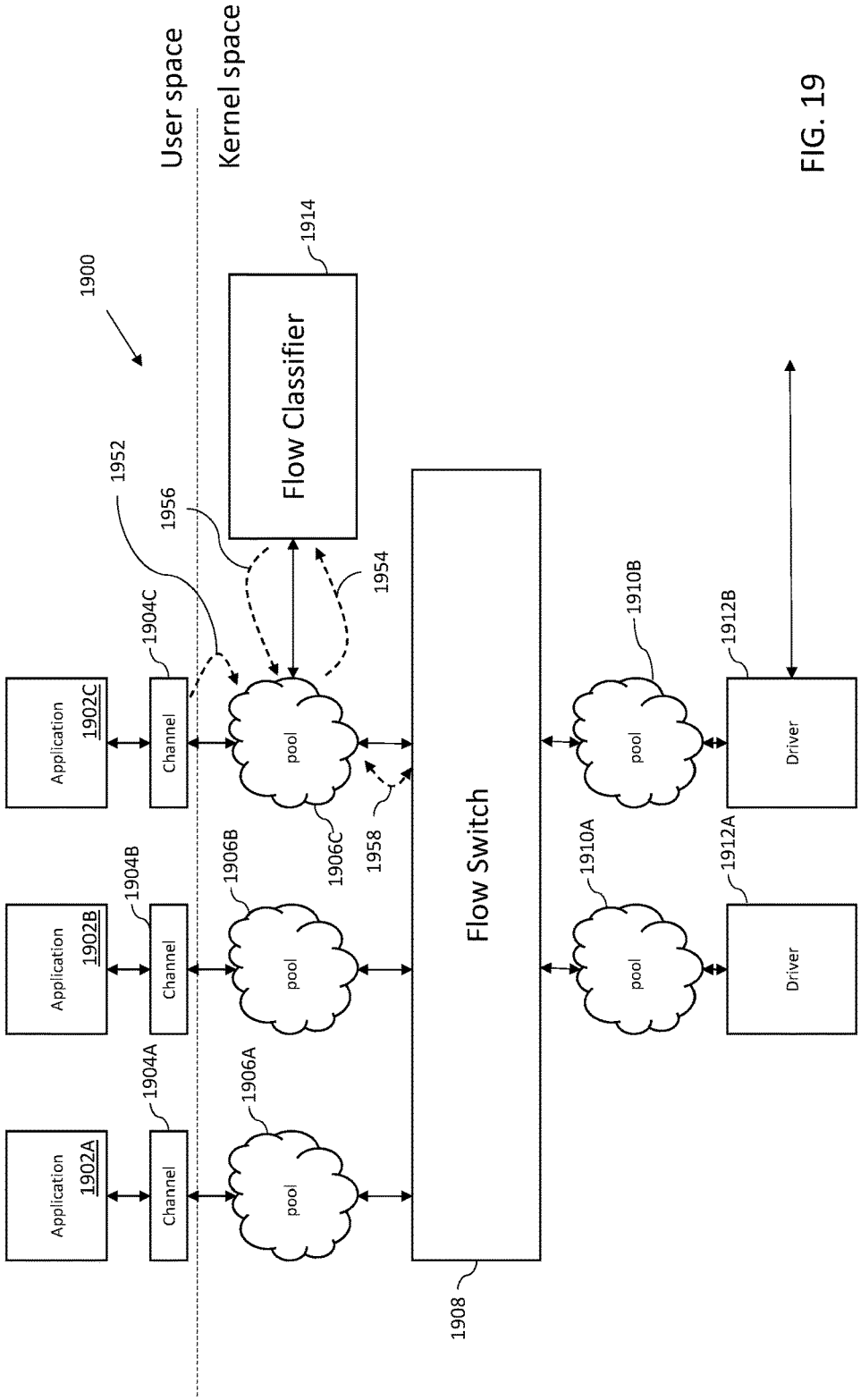

User space
Kernel space
1900
1914
Flow Classifier
1952
1956
1954
1904C
Application 1902C
Channel
pool
1906C
1958
1910B
1912B
Driver
1904B
Application 1902B
Channel
1906B
pool
Flow Switch
1910A
1912A
Driver
1904A
Application 1902A
Channel
1906A
pool
1908

FIG. 19

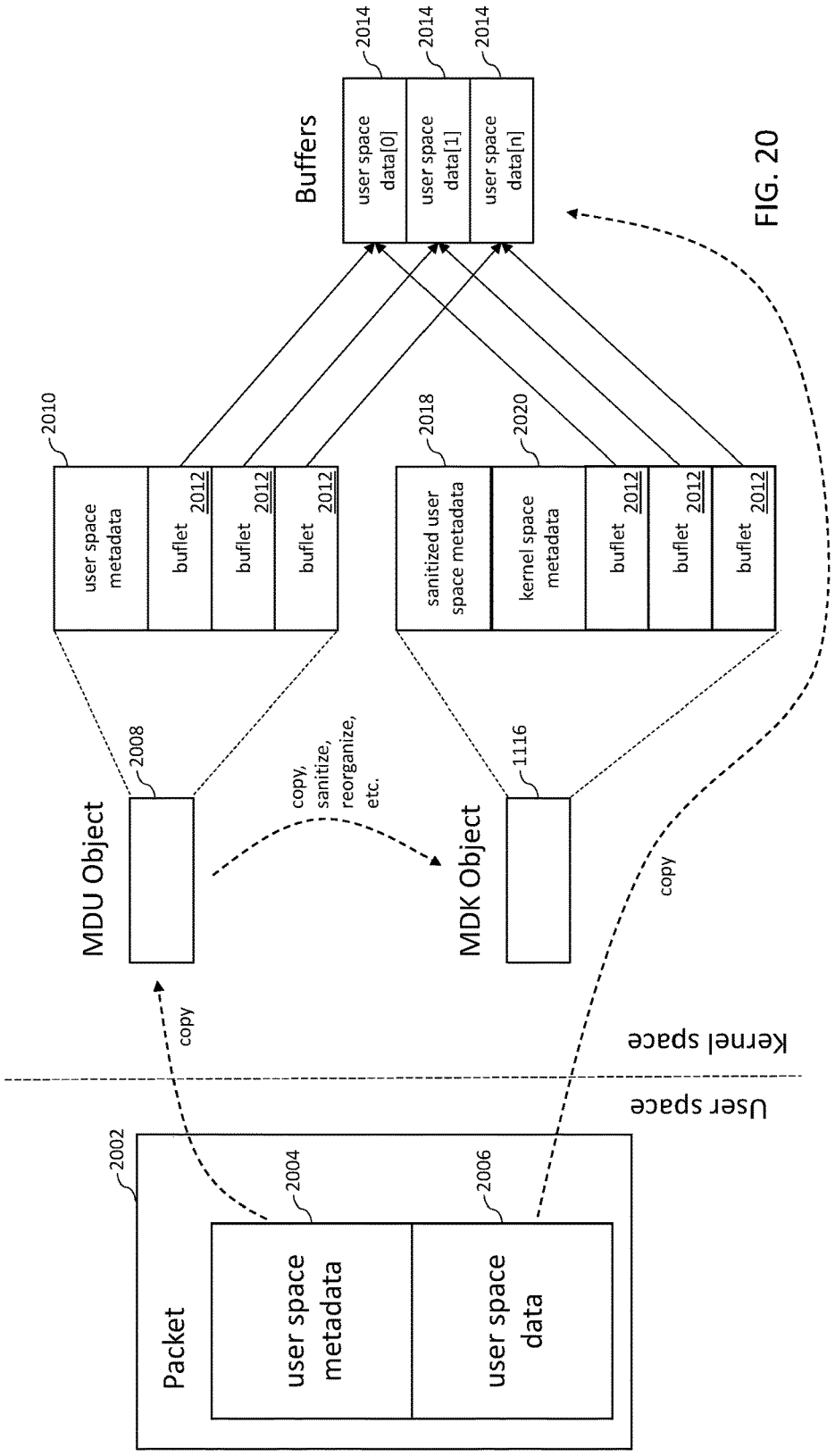

Buffers
2014
2014
2014
user space data[0]
user space data[1]
user space data[n]
2010
2018
2020
user space metadata
buflet 2012
buflet 2012
buflet 2012
sanitized user space metadata
kernel space metadata
buflet 2012
buflet 2012
buflet 2012
MDU Object
2008
copy, sanitize, reorganize, etc.
MDK Object
1116
copy
copy
Kernel space
User space
2002
2004
2006
Packet
user space metadata
user space data

FIG. 20

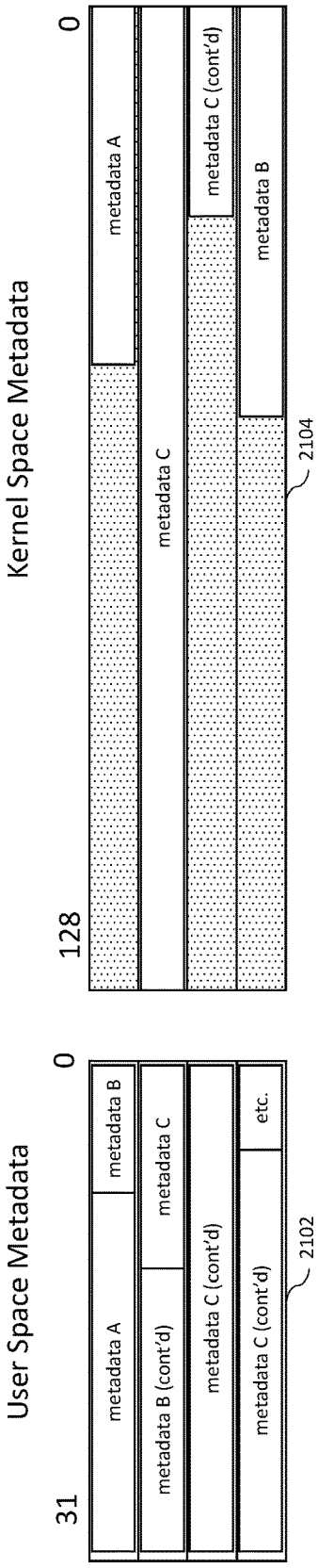

User Space Metadata
0
31
metadata A
metadata B
metadata B (cont'd)
metadata C
metadata C (cont'd)
metadata C (cont'd)
etc.
2102
Kernel Space Metadata
0
128
metadata A
metadata C
metadata C (cont'd)
metadata B
2104

FIG. 21

FIG. 22

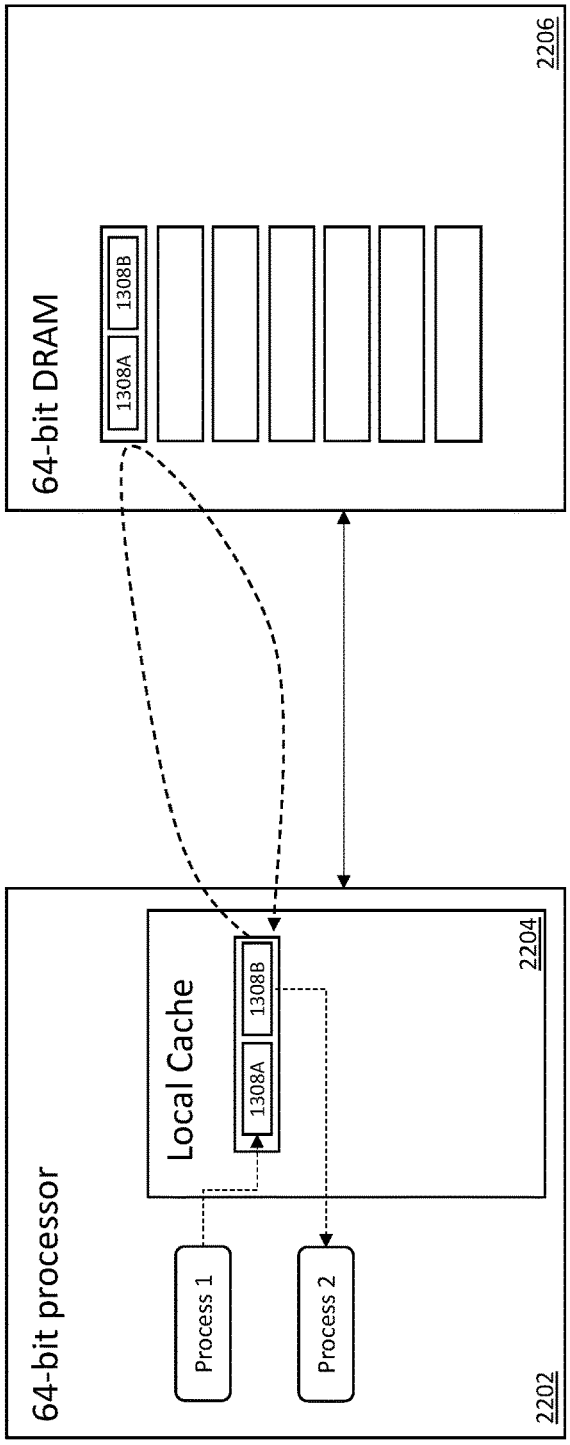
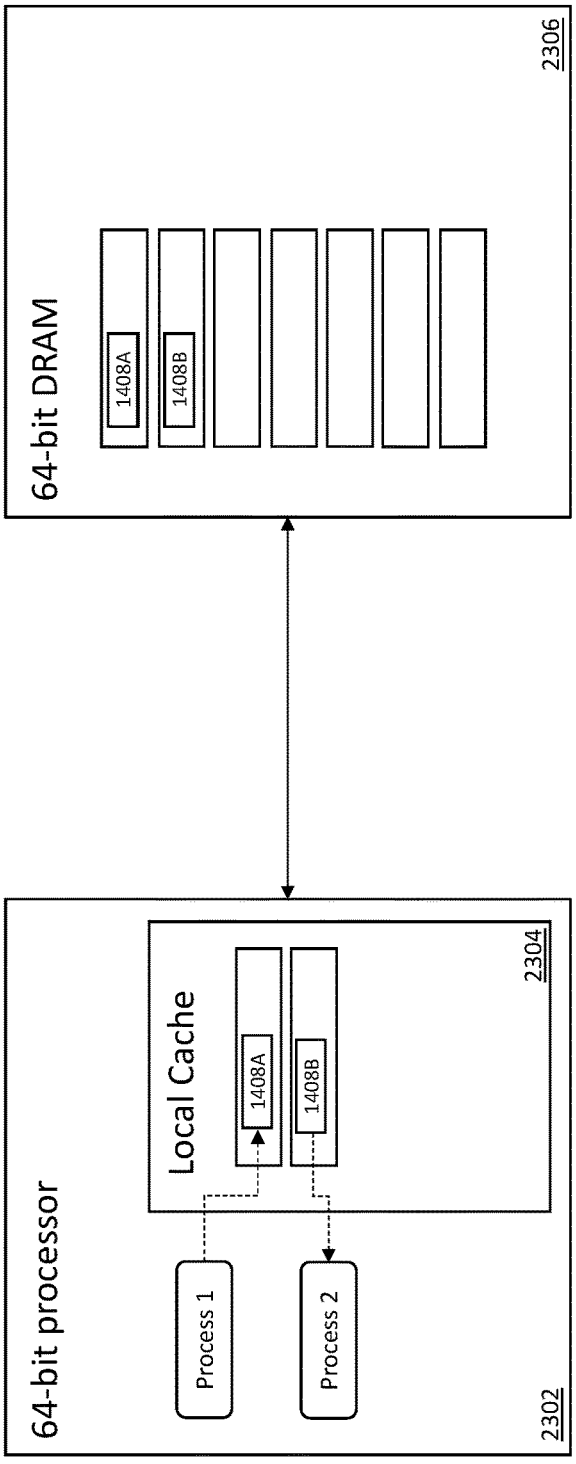

64-bit DRAM
2206
1308A
1308B
64-bit processor
Local Cache
2204
1308A
1308B
Process 1
Process 2
2202

FIG. 23


64-bit DRAM
2306
1408A
1408B
64-bit processor
Local Cache
2304
1408A
1408B
Process 1
Process 2
2302

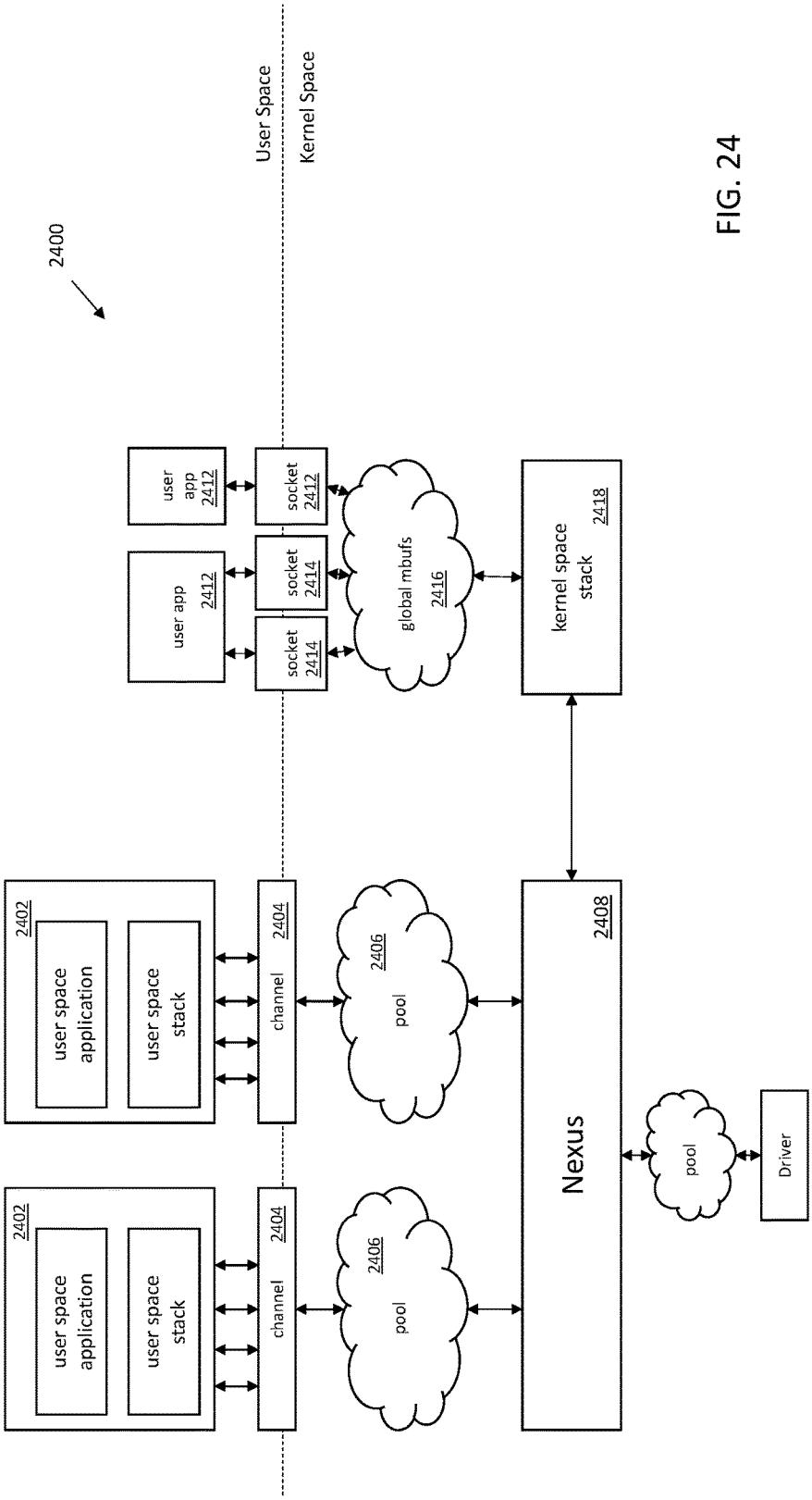

2400
User Space
Kernel Space
user app 2412
socket 2412
user app 2412
socket 2414
socket 2414
global mbufs 2416
kernel space stack 2418
2402
user space application
user space stack
channel 2404
2406
pool
Nexus 2408
pool
Driver
2402
user space application
user space stack
channel 2404
2406
pool

FIG. 24

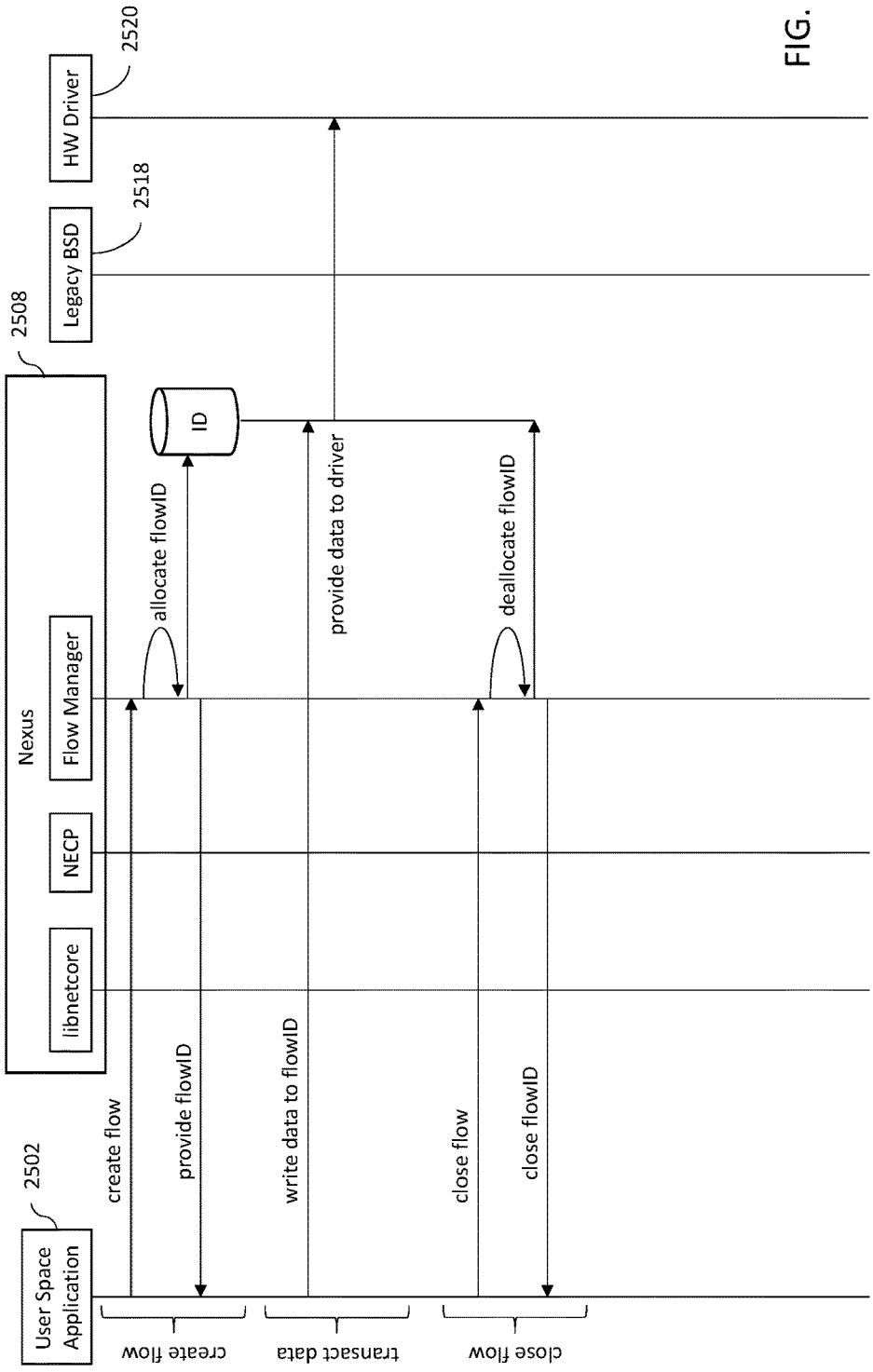

User Space Application
2502
Nexus
2508
libnetcore
NECP
Flow Manager
Legacy BSD
2518
HW Driver
2520
ID
create flow
provide flowID
write data to flowID
close flow
close flowID
allocate flowID
provide data to driver
deallocate flowID
create flow
transact data
close flow

FIG. 25A

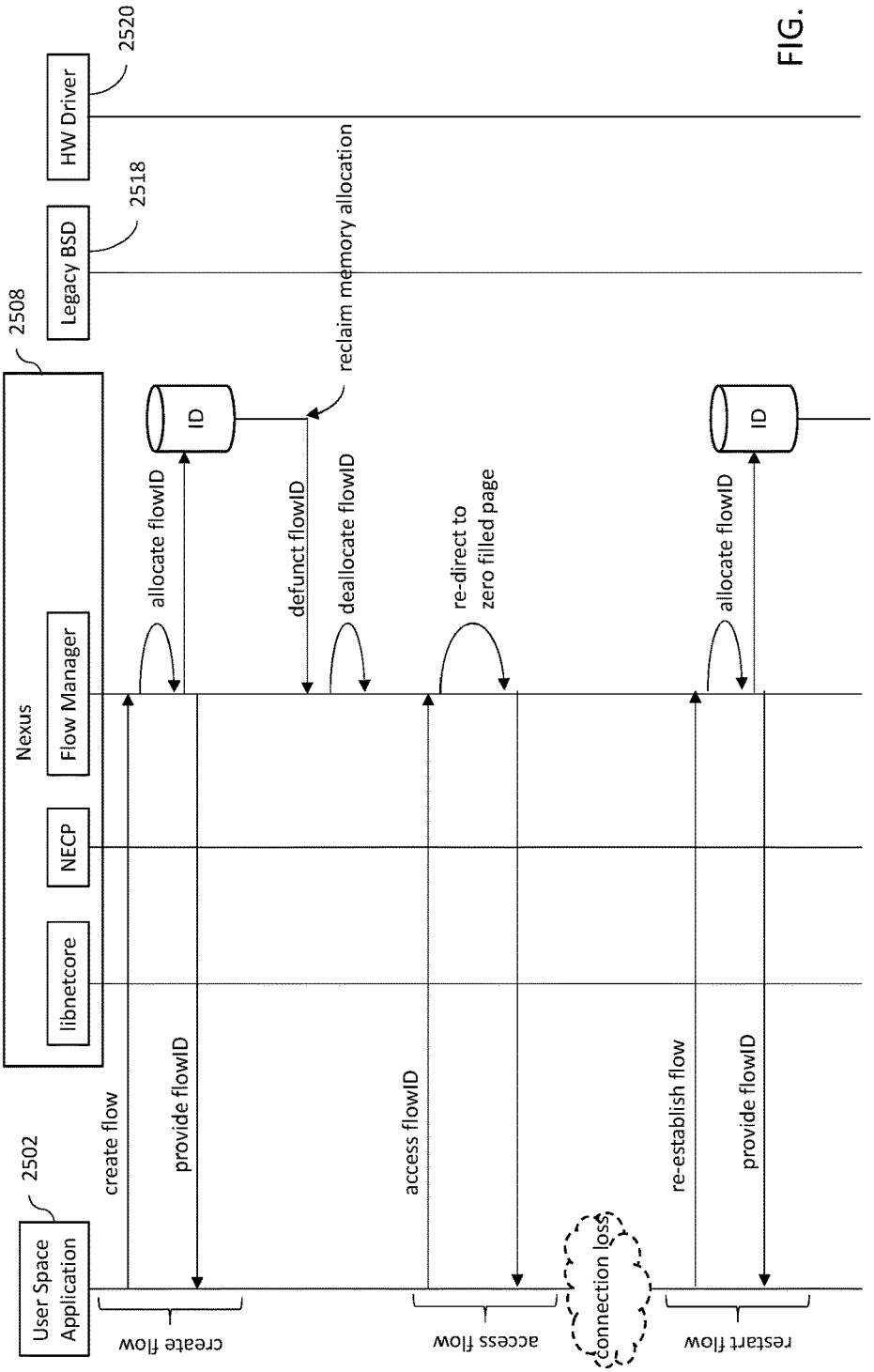

2520
HW Driver
2518
Legacy BSD
2508
Nexus
Flow Manager
NECP
libnetcore
2502
User Space Application
reclaim memory allocation
ID
allocate flowID
defunct flowID
deallocate flowID
re-direct to zero filled page
ID
allocate flowID
create flow
provide flowID
access flowID
connection loss
re-establish flow
provide flowID
create flow
access flow
restart flow

FIG. 25B

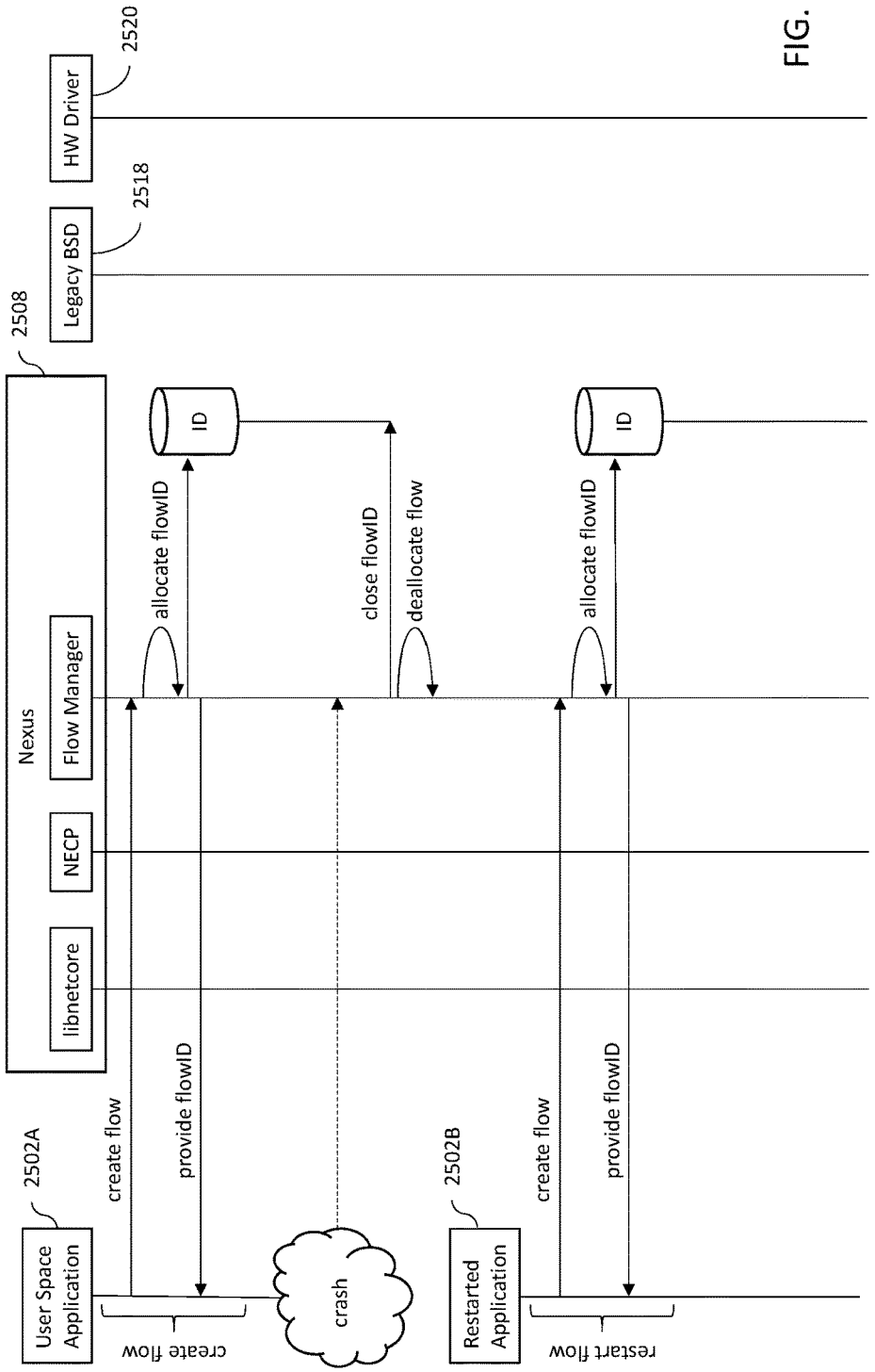

2520
HW Driver
2518
Legacy BSD
2508
Nexus
Flow Manager
NECP
libnetcore
ID
ID
allocate flowID
close flowID
deallocate flow
allocate flowID
2502A
User Space Application
create flow
provide flowID
crash
2502B
Restarted Application
create flow
provide flowID
create flow
restart flow

FIG. 25C

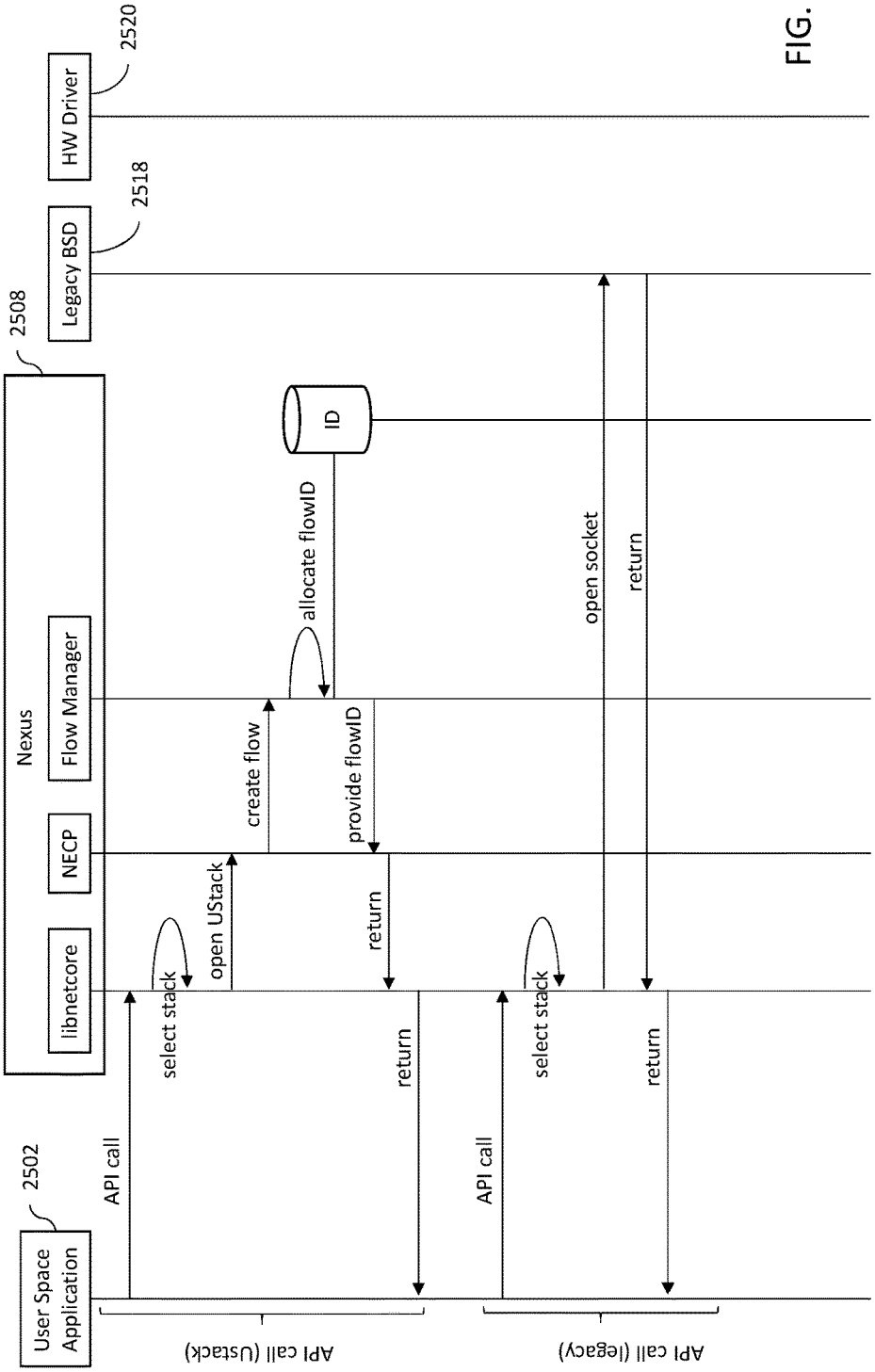

User Space Application
2502
API call
select stack
open UStack
create flow
allocate flowID
ID
provide flowID
return
return
API call (Ustack)
API call
select stack
open socket
return
return
API call (legacy)
Nexus
2508
libnetcore
NECP
Flow Manager
Legacy BSD
2518
HW Driver
2520

FIG. 25D

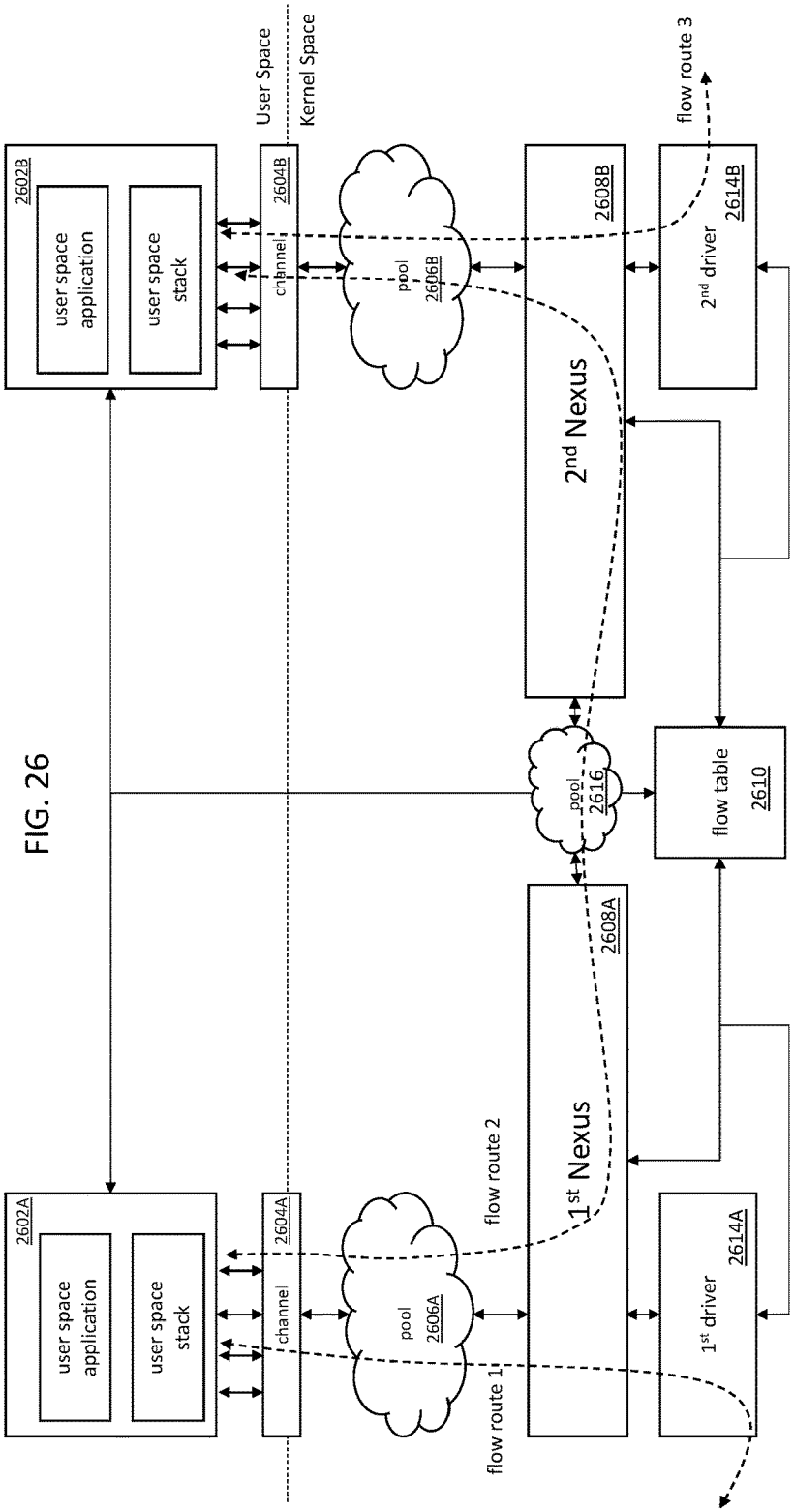

User Space
Kernel Space
flow route 3
2602B
user space application
user space stack
2604B
channel
pool 2606B
2608B
2nd Nexus
2nd driver
2614B
pool 2616
flow table 2610
2602A
user space application
user space stack
2604A
channel
pool 2606A
flow route 2
flow route 1
1st Nexus
2608A
1st driver
2614A

FIG. 26

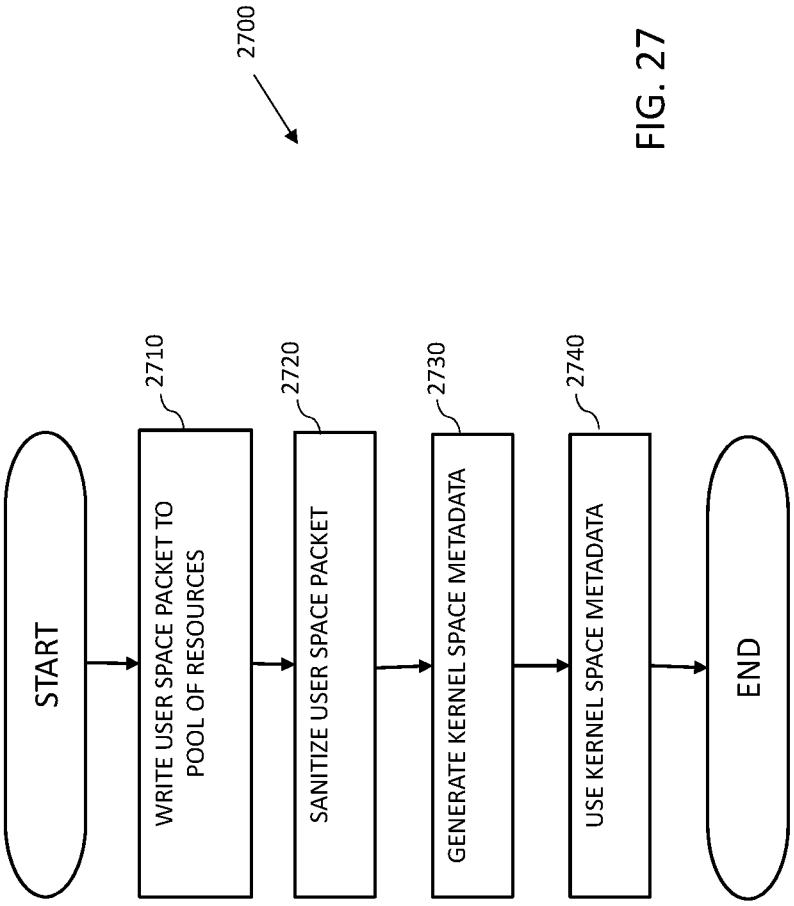

2700
START
2710
WRITE USER SPACE PACKET TO POOL OF RESOURCES
2720
SANITIZE USER SPACE PACKET
2730
GENERATE KERNEL SPACE METADATA
2740
USE KERNEL SPACE METADATA
END

FIG. 27

METHODS AND APPARATUS FOR SECURE OPERATION OF USER SPACE COMMUNICATION STACKS

PRIORITY

[0001] This application claims the benefit of priority to U.S. Provisional Patent Application Ser. No. 62/649,509 filed Mar. 28, 2018 and entitled "METHODS AND APPARATUS FOR EFFICIENT DATA TRANSFER WITHIN USER SPACE NETWORKING STACK INFRASTRUCTURES", which is incorporated herein by reference in its entirety.

RELATED APPLICATIONS

[0002] This application is related to U.S. patent application Ser. No. 16/144,992 filed Sep. 27, 2018 and entitled "Methods and Apparatus for Single Entity Buffer Pool Management", U.S. patent application Ser. No. 16/146,533 filed Sep. 28, 2018 and entitled "Methods and Apparatus for Regulating Networking Traffic in Bursty System Conditions", U.S. patent application Ser. No. 16/146,324 filed Sep. 28, 2018 and entitled "Methods and Apparatus for Preventing Packet Spoofing with User Space Communication Stacks", U.S. patent application Ser. No. 16/146,916 filed Sep. 28, 2018 and entitled "Methods and Apparatus for Channel Defunct Within User Space Stack Architectures", U.S. patent application Ser. No. 16/236,032 filed Dec. 28, 2018 and entitled "Methods and Apparatus for Classification of Flow Metadata with User Space Communication Stacks", U.S. patent application Ser. No. 16/363,495 filed Mar. 25, 2019 and entitled "Methods and Apparatus for Dynamic Packet Pool Configuration in Networking Stack Infrastructures", U.S. patent application Ser. No. 16/365,462 filed Mar. 26, 2019 and entitled "Methods and Apparatus for Sharing and Arbitration of Host Stack Information with User Space Communication Stacks", U.S. patent application Ser. No. 16/365,484 filed Mar. 26, 2019 and entitled "Methods and Apparatus for Virtualized Hardware Optimizations for User Space Networking", U.S. patent application Ser. No. _____, filed concurrently herewith on Mar. 28, 2019 and entitled "Methods and Apparatus for Memory Allocation and Reallocation in Networking Stack Infrastructures", U.S. patent application Ser. No. _____filed concurrently herewith on Mar. 28, 2019 and entitled "Methods and Apparatus for Active Queue Management in User Space Networking", and U.S. patent application Ser. No. _____, filed concurrently herewith on Mar. 28, 2019 and entitled "Methods and Apparatus for Self-Tuning Operation within User Space Stack Architectures", each of the foregoing being incorporated herein by reference in its entirety.

COPYRIGHT

[0003] A portion of the disclosure of this patent document contains material that is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

1. TECHNICAL FIELD

[0004] The disclosure relates generally to the field of electronic devices, as well as networks thereof. More particularly, the disclosure is directed to methods and apparatus for implementing computerized networking stack infrastructures. Various aspects of the present disclosure are directed to, in one exemplary aspect, data transfer within user space networking stack infrastructures.

2. DESCRIPTION OF RELATED TECHNOLOGY

[0005] The consumer electronics industry has seen explosive growth in network connectivity; for example, Internet connectivity is now virtually ubiquitous across many different device types for a variety of different applications and functionalities. The successful implementation of network connectivity over a myriad of different usage cases has been enabled by, inter alia, the principles of modular design and abstraction. Specifically, the traditional network communication paradigm incorporates multiple (generally) modular software "layers" into a "communication stack." Each layer of the communication stack separately manages its own implementation specific considerations, and provides an "abstracted" communication interface to the next layer. In this manner, different applications can communicate freely across different devices without considering the underlying network transport.

[0006] The traditional network communication paradigm has been relatively stable for over 30 years. The Assignee hereof has developed its own implementation of a computer networking stack (based on the traditional networking paradigm) that is mature, robust, and feature-rich (yet conservative). This networking stack is the foundation for virtually all networking capabilities, including those used across the Assignee's products (e.g., MacBook®, iMac®, iPad®, and iPhone®, etc.) and has been designed to handle a variety of protocols (such as TCP (Transmission Control Protocol), UDP (User Datagram Protocol) and IP (Internet Protocol)), and proprietary extensions and functionalities.

[0007] While the traditional network communication paradigm has many benefits, changes in the commercial landscape have stretched the capabilities of the existing implementations. Over the past years new use cases have emerged that require capabilities beyond those of the traditional networking stack design. For example, some use cases require control and data movement operations to be performed in so-called "user space" (software that is executed outside the kernel, and specific to a user process). Common examples of such applications include without limitation e.g. Virtual Private Networks (VPN), application proxy, content and traffic filtering, and any number of other network-aware user applications.

[0008] Furthermore, certain types of user applications (e.g., media playback, real-time or interactive network applications) would benefit from workload-specific customizations and performance optimizations of the networking stack.

[0009] Unfortunately, the current one-size-fits-all networking stack was not designed for (and is thus ill-suited to) the requirements of the aforementioned use cases (and others contemplated herein). More directly, supporting user space applications and associated components from within the traditional in-kernel networking stack architecture adds complexity, increases technical debts (the implied cost of rework attributed to deploying a faster, but suboptimal, implementation), brings in higher processing costs, and results in suboptimal performance and higher power consumption.

[0010] To these ends, a networking stack architecture and technology that caters to emerging non-kernel use cases is needed. Ideally, but not as a requisite, such solutions should preserve backwards compatibility with the traditional in-kernel networking stack.

[0011] More generally, improved methods and apparatus for manipulating and/or controlling lower layer networking communication protocols by higher layer software applications is desired.

SUMMARY

[0012] The present disclosure satisfies the foregoing needs by providing, inter alia, methods and apparatus for data transfer within user space networking stack infrastructures.

[0013] In a first aspect of the disclosure, methods and apparatus to address contiguous or adjacent memory objects (which are prone to inadvertent memory corruption due to buffer overrun issues) are disclosed, as are methods and apparatus for a user to detect such issues.

[0014] In one embodiment, data descriptors also known as packet or quantum have a metadata preamble placed at the beginning of the object. This metadata preamble is used to detect any inadvertent overwrite of the metadata. In one variant, each metadata object has a unique red zone pattern which is the XOR of a red zone cookie and the offset of the metadata object in the object's memory region. red zone cookies are initialized with random numbers on an OS boot. In the event the kernel detects a corruption, the user space process associated with the channel is terminated to prevent further damages.

[0015] In a second aspect, methods and apparatus for enhanced security are disclosed. In one embodiment, the architecture maintains a mirrored copy of the packet descriptor memory which is accessible only from the kernel. During packet handoff from user-space to kernel, the user accessible descriptor is validated (against the kernel copy) for any semantic issues and the sanitized data is copied to the kernel mapped descriptor.

[0016] In another aspect, methods and apparatus for access control on user space network architecture ports to prevent unauthorized clients from opening channels are disclosed. In one embodiment, an access control mechanism is provided based on one or more attributes associated with a channel client, namely process ID, process executable's UUID, or key blob. A process (e.g., Nexus provider) chooses to select one or a combination of those attributes for securing access to a port of a named instance.

[0017] In another aspect, methods and apparatus for RST flood detection and mitigation are disclosed. In one embodiment, mechanisms to prevent SYN flood and RST flood attacks originating from the user space stack are provided. A USNSI flow-switch implements flow tracking logic which can detect SYN flood and RST flood to prevent these attacks originating from a device. If an attack is detected, the flow-switch will rate-limit the SYN and RST packets coming from the user space stack.

[0018] In another aspect, methods and apparatus for split TX and RX packet pools (direction-specific DMA access for security) are disclosed. In one embodiment, buggy or hostile devices are prevented from using PCIe-mapped buffers to attack the host, such as by overwriting the content of in-use buffers, or performing timing/time-of-use based attacks. A USNSI setup maps segments to use the minimum possible memory access permissions on receive and transmit packet buffers.

[0019] In another aspect, methods and apparatus for use of randomized memory segment sizes are disclosed. As noted, buggy or hostile devices may use PCIe-mapped buffers to attack the host; to help mitigate this vulnerability, the system will randomize the PCIe address space mappings, to make it difficult for an attacker to find vulnerable host-side resources. To help support this security protection, A USNSI may in one variant randomize its segment size by randomizing the number of pages per segment at the time segments are allocated. In another variant, a USNSI may also randomize packet order within a segment, to make it more difficult to correlate packet address to position within a segment.

[0020] In another aspect, methods and apparatus for Device TOCTOU attack mitigation are disclosed. In one embodiment, the process (e.g., Nexus) makes a kernel only copy before accessing device supplied data, all subsequent "sanity" checks and uses on the data are carried out on the kernel only copy. Even if a compromised device launches TOCTOU attack, the kernel detects and uses the consistent kernel-only copy that is not affected.

[0021] In another aspect, methods and apparatus for managing entitlements to access statistics and Nexus operations are disclosed. In one embodiment, entitlements checks for privileged operations are conducted only by processes possessing such entitlements, e.g. trusted processes.

[0022] In another aspect, methods and apparatus for leveraging RTT estimation data for bounds checking are disclosed. RTT measurement is a critical value for TCP operations such as retransmission and fast recovery; hence, in one embodiment, the TCP stack(s) are in user space, and the kernel also performs its own rough RTT using flow tracker in the flow-switch. To accept measurements from user space, the kernel conducts a "sanity" check with its estimated upper and lower bounds. Only the RTT samples that passed the kernel sanity check could be published to other TCP stack instances.

[0023] In another aspect, methods and apparatus for malicious statistics detection before folding into trusted statistics are disclosed. In one embodiment, a process (e.g., the Nexus) also instantiates a shadow kernel-only statistics object in addition to the user space protocol stack instance shared statistics object. The kernel-only statistics object stores historical values of the user space protocol stack statistics. Before accepting the user space protocol stack statistics, the Nexus derives a delta of each uTCP statistics snapshot with the historical value and conducts an anomaly detection. Also for critical statistics, such as cellular data usage, the USNSI in one variant only relies on trusted flow-switch kernel observed statistics.

[0024] In another aspect, methods and apparatus for preventing IP Address/port spoofing are disclosed. In one embodiment, the TCP/IP stacks is/are in user space, a flow-switch is used to performs a flow 5-tuple lookup in the kernel with the registered flows before packets are transmitted; e.g., to make sure the sender has the 5-tuple registration. Any packets with un-matching 5-tuple and various other metadata such as flow ID would be dropped or otherwise handled.

[0025] In another aspect, methods and apparatus for trusted TFO & ECN are disclosed. In one embodiment, a

TCP stack that supports both TCP Fast Open (TFO) and Explicit Congestion Notification (ECN) is used; both the TCP options are enabled/disabled based on per network heuristics maintained on the system, so as to avoid using TFO and ECN on networks that either do not support these options, or blacklist devices if the options are present in the TCP header.

[0026] In one variant, the ECN and TFO heuristics is updated each time a TCP connection experiences a success or failure when using TFO or ECN, and the USNSI TCP protocol stack runs in the user process's context, and all processes can indicate to the system heuristics a failure of TFO or ECN—however, only processes that are trusted on the system can update the heuristics with TFO or ECN success. This prevents malicious apps from incorrectly updating TFO or ECN success on networks that do not support these options.

[0027] In one aspect, a method for validating user space packet descriptors is disclosed. In one embodiment, the method includes: receiving a plurality of packets from a user space process; for each packet of the plurality of packets: extracting a data structure from the packet; validating that the data structure conforms to a packet descriptor format; and providing validated packets to a kernel space process.

[0028] In one variant, the extracting the data structure from the each packet comprises extracting a pattern from a packet header associated with the each packet. In one such variant, validating that the data structure conforms to a packet descriptor format comprises checking the pattern based on a random number and an address associated with the each packet.

[0029] In another variant, the method includes terminating the user space process when at least one packet is invalid.

[0030] In another variant, the validating that the data structure conforms to a packet descriptor format comprises checking that the data structure comprises a valid checksum.

[0031] In another variant, the validating that the data structure conforms to a packet descriptor format comprises checking that the data structure comprises a base offset from an address. In one such case, the providing validated packets to the kernel space process further comprises translating the base offset from the address to a pointer in kernel virtual address space.

[0032] In another aspect, a method for tracking user space packet descriptors is disclosed. In one embodiment, the method includes: receiving a plurality of packets from a user space process; determining a connection state based at least in part on a plurality of packet descriptors associated with a first set of the plurality of packets from the user space process; and transmitting the first set of the plurality of packets from the user space process when the connection state is valid.

[0033] In one variant, the method includes determining a second connection state associated with a second set of the plurality of packets from the user space process; and rate limiting the second set of the plurality of packets from the user space process when the connection state is invalid. In one such case, the second set of the plurality of packets from the user space process comprise synchronization packets (SYN) that are sent while one or more acknowledgement packets (ACK) packets have not been received. In another such case, the second set of the plurality of packets from the user space process comprise reset packets (RST) that are sent without an outstanding socket connection. In a third such case, the second set of the plurality of packets from the user space process comprise reset packets (RST) that are sent without an outstanding synchronization packet (SYN).

[0034] In one variant, the method further includes generating one or more kernel space statistics based on the first set of the plurality of packets from the user space process when the connection state is valid.

[0035] In one variant, the method further includes determining a second connection state associated with a second set of the plurality of packets from the user space process; and ignoring one or more user space statistics based on the second set of the plurality of packets from the user space process when the connection state is valid.

[0036] In another aspect, a method for propagating sensitive network statistics is disclosed. In one embodiment, the method includes: receiving network statistics from a user space process; validating the network statistics based at least in part on a plurality of packet descriptors associated with a first set of a plurality of packets from the user space process; updating kernel space network statistics with the network statistics from the user space process based on the validation; and generating an obfuscated set of network statistics for a second user space process based on the kernel space network statistics.

[0037] In one variant, the receiving the network statistics from the user space process includes receiving round trip delay time (RTT) data. In one variant, validating the network statistics based at least in part on the plurality of packet descriptors associated with the first set of the plurality of packets from the user space process comprises determining an upper and a lower RTT bound.

[0038] In another variant, the user space process comprises a user space library that is self-contained. In one such case, the second user space process comprises a user space application that has entitlements to access the network statistics generated by the user space library. In another such case, the user space library and the second user space process are operating from a common application space.

[0039] In another aspect, methods and apparatus for flow classification are disclosed. In one embodiment, USNSI packets have a struct flow as part of packet metadata which contains most information that those layers need, and it is carried into BSD/user space, etc. The contents of this structure are filled once by the flow-switch.

[0040] In another aspect, methods and apparatus for flow management are disclosed. In one embodiment, flow life-cycle, e.g. flow creation, destroy, which interfaces with calls/events from other components, is managed. In one variant, a flow manager is the entity that provides such interface. It accepts calls to create/destroy/defunct flows. It also shuts down flow when the flow owner process exits. This allows proper clean-ups to be done regardless of how the process terminates.

[0041] In another aspect, methods and apparatus for flow entry management are disclosed. In one embodiment, a mechanism to facilitate efficient packet forwarding within a USNSI flow-switch includes packet forwarding based on the entries of a flow table, which allows facilitation of optimal forwarding data plane logic, where e.g., a multiple network interface Nexus is fused together to form a direct conduit for sending packets to one another. In another aspect, methods and apparatus for flow action management are disclosed. In one embodiment, a flow-switch flow carries action on packets for a given flow; a mechanism is disclosed whereby

possible actions that can be applied to packet, e.g. forward to a flow-switch port to user space protocol stack, forward to BSD stack, drop, transform, etc., are defined. This approach allows for, inter alia, an efficient way to apply traffic rules without involving separate database lookups.

[0042] In another aspect, methods and apparatus for flow route management are disclosed. In one embodiment, a USNSI flow route comprises a cache around those BSD info, such that for USNSI flow packets can find those information within USNSI context along with flow lookup. The flow route is notified when related events happen, e.g. route change, ARP expire, to maintain consistency. The flow routes allow for packets going out of the system via USNSI channels to not incur per-packet routing table lookup costs.

[0043] In another aspect, methods and apparatus for flow tracking are disclosed. In one embodiment, a process (e.g., flow-switch) has a flow tracker that passively tracks flow state/statistics during flow classification and classifier. It provides KPI for other component to query flow states and statistics. It also implements pro-active actions in cleaning up flows that are, e.g., deemed to be terminated (by both ends) and not expecting any more data.

[0044] In another aspect, methods and apparatus for achieving low latency for urgent packets using flow tracking are disclosed. In one embodiment, urgent packets like DNS queries/TCP control packets are identified and processed (e.g., via a flush/notify) when detected to ensure we deliver them with low latency. This allows for, inter alia, dynamic adjustment of the notifications posted to the user space process depending on the contents of the packets.

[0045] In another aspect, methods and apparatus for flow purging/"defunct-ing" are disclosed. In one embodiment, a flow tracker passively updates flow state; a process (e.g., flow-switch) actively scans through all flows and find those dead flows and close them; and orthogonally during defunct, a process (e.g., assertion) calls in to defunct flows when a target process goes suspended.

[0046] Other features and advantages of the present disclosure will immediately be recognized by persons of ordinary skill in the art with reference to the attached drawings and detailed description of exemplary embodiments as given below.

BRIEF DESCRIPTION OF THE DRAWINGS

[0047] FIG. **1** is a logical representation of a traditional network socket, useful for explaining various aspects of the present disclosure.

[0048] FIG. **2** is a logical representation of a computer system that implements Input/Output (I/O) network control, useful for explaining various aspects of the present disclosure.

[0049] FIG. **3** is a logical block diagram of one exemplary implementation of Transport Layer Security (TLS), useful for explaining various aspects of the present disclosure.

[0050] FIG. **4** is a logical block diagram of an exemplary implementation of a Virtual Private Network (VPN), useful for explaining various aspects of the present disclosure.

[0051] FIG. **5** is a logical block diagram of an exemplary implementation of application based tuning, useful for explaining various aspects of the present disclosure.

[0052] FIG. **6** is a logical representation of an exemplary networking stack architecture, in accordance with the various aspects of the present disclosure.

[0053] FIG. **7** is a logical block diagram of an exemplary user space networking stack, in accordance with the various aspects of the present disclosure.

[0054] FIG. **8** is a logical flow diagram useful to summarize the convoluted data path taken for a prior art application using a proxy agent application within the context of the traditional networking stack, useful for explaining various aspects of the present disclosure.

[0055] FIG. **9** is a logical flow diagram useful to summarize an exemplary proxy agent application within the context of the user space networking stack, in accordance with various aspects of the present disclosure.

[0056] FIG. **10** is a graphical comparison of a packet data structure during normal operation, and a buffer overrun, useful for explaining various aspects of the present disclosure.

[0057] FIG. **11** is a graphical comparison of an exemplary packet data structure during normal operation, and a buffer overrun, in accordance with various aspects of the present disclosure.

[0058] FIG. **12** is a logical block diagram of an exemplary user space packet and its corresponding components in kernel space useful to demonstrate red zone pattern operation, in accordance with various aspects of the present disclosure.

[0059] FIG. **13** is a logical block diagram of an exemplary user space packet and its corresponding components in kernel space useful to demonstrate internalization and externalization of data, in accordance with various aspects of the present disclosure.

[0060] FIG. **14** is a logical block diagram useful to illustrate the creation of a user space inter-process pipe ("upipe"), in accordance with various aspects of the present disclosure.

[0061] FIGS. **15**A and **15**B are graphical representations of normal TCP/IP SYN operation and SYN flood attacks, useful for explaining various aspects of the present disclosure.

[0062] FIG. **16** is a graphical representation of normal TCP/IP RST, useful for explaining various aspects of the present disclosure.

[0063] FIGS. **17**A and **17**B are graphical representations of exemplary TCP/IP SYN operation and SYN flood attack countermeasures, in accordance with various aspects of the present disclosure.

[0064] FIGS. **18**A and **18**B are graphical representations of exemplary TCP/IP RST operation and RST flood attack countermeasures, in accordance with various aspects of the present disclosure.

[0065] FIG. **19** is a logical flow diagram of a system for flow classification, in accordance with various aspects of the present disclosure.

[0066] FIG. **20** is a logical representation of one exemplary data packet structure for use with, for example, a system for flow classification, in accordance with various aspects of the present disclosure.

[0067] FIG. **21** is a logical representation of exemplary metadata packed within 32-bit words in user space and the exemplary metadata padded and naturally aligned to 64-bit word boundaries in kernel space, in accordance with various aspects of the present disclosure.

[0068] FIG. **22** is a logical representation of one exemplary 64-bit access of 32-bit packed metadata, useful for explaining various aspects of the present disclosure.

[0069] FIG. **23** is a logical representation of one exemplary 64-bit access of naturally aligned metadata, in accordance with various aspects of the present disclosure.

[0070] FIG. **24** is a logical block diagram of one exemplary nexus supporting multiple user space applications under both user space networking stacks as well as legacy networking stacks, useful for explaining various aspects of the present disclosure.

[0071] FIGS. **25**A-**25** D are software ladder diagrams of various flow management tasks made possible by the exemplary user space networking stack architecture, in accordance with various aspects of the present disclosure.

[0072] FIG. **26** is a logical block diagram of a logical conduit for fusing nexuses, in accordance with various aspects of the present disclosure.

[0073] FIG. **27** is a logical flow diagram illustrating an exemplary methodology for using, for example, a system for flow classification, in accordance with various aspects of the present disclosure.

[0074] All figures © Copyright 2017-2019 Apple Inc. All rights reserved.

DETAILED DESCRIPTION

[0075] Reference is now made to the drawings, wherein like numerals refer to like parts throughout.

Detailed Description of Exemplary Embodiments

[0076] Exemplary embodiments of the present disclosure are now described in detail. While embodiments are primarily discussed in the context of use in conjunction with an inter-processor communication (IPC) link such as that described in, for example, commonly owned U.S. patent application Ser. No. 14/879,024 filed Oct. 8, 2015 and entitled "METHODS AND APPARATUS FOR RUNNING AND BOOTING AN INTER-PROCESSOR COMMUNICATION LINK BETWEEN INDEPENDENTLY OPERABLE PROCESSORS", now U.S. Pat. No. 10,078,361, and co-owned and co-pending U.S. patent application Ser. No. 16/112,480 filed Aug. 24, 2018 and entitled "METHODS AND APPARATUS FOR CONTROL OF A JOINTLY SHARED MEMORY-MAPPED REGION", each of which being incorporated herein by reference in its entirety, it will be recognized by those of ordinary skill that the present disclosure is not so limited.

Existing Network Socket Technologies—

[0077] FIG. **1** illustrates one logical representation of a traditional network socket **102**, useful for explaining various aspects of the traditional networking interface. A network "socket" is a virtualized internal network endpoint for sending or receiving data at a single node in a computer network. A network socket may be created ("opened") or destroyed ("closed") and the manifest of network sockets may be stored as entries in a network resource table which may additionally include reference to various communication protocols (e.g., Transmission Control Protocol (TCP) **104**, User Datagram Protocol (UDP) **106**, Inter-Processor Communication (IPC) **108**, etc.), destination, status, and any other operational processes (kernel extensions **112**) and/or parameters); more generally, network sockets are a form of system resource.

[0078] As shown in FIG. **1**, the socket **102** provides an application programming interface (API) that spans between the user space and the kernel space. An API is a set of clearly defined methods of communication between various software components. An API specification commonly includes, without limitation: routines, data structures, object classes, variables, remote calls and/or any number of other software constructs commonly defined within the computing arts.

[0079] As a brief aside, user space is a portion of system memory that a processor executes user processes from. User space is relatively freely and dynamically allocated for application software and a few device drivers. The kernel space is a portion of memory that a processor executes the kernel from. Kernel space is strictly reserved (usually during the processor boot sequence) for running privileged operating system (O/S) processes, extensions, and most device drivers. For example, each user space process normally runs in a specific memory space (its own "sandbox"), and cannot access the memory of other processes unless explicitly allowed. In contrast, the kernel is the core of a computer's operating system; the kernel can exert complete control over all other processes in the system.

[0080] The term "operating system" may refer to software that controls and manages access to hardware. An O/S commonly supports processing functions such as e.g., task scheduling, application execution, input and output management, memory management, security, and peripheral access. As used herein, the term "application" refers to software that can interact with the hardware only via procedures and interfaces offered by the O/S. The term "privilege" may refer to any access restriction or permission which restricts or permits processor execution. System privileges are commonly used within the computing arts to, inter alia, mitigate the potential damage of a computer security vulnerability. For instance, a properly privileged computer system will prevent malicious software applications from affecting data and task execution associated with other applications and the kernel.

[0081] As used herein, the term "in-kernel" and/or "kernel space" may refer to data and/or processes that are stored in, and/or have privilege to access to, the kernel space memory allocations. In contrast, the terms "non-kernel" and/or "user space" refers to data and/or processes that are not privileged to access the kernel space memory allocations. In particular, user space represents the address space specific to the user process, whereas non-kernel space represents address space which is not in-kernel, but which may or may not be specific to user processes.

[0082] As previously noted, the illustrated socket **102** provides access to Transmission Control Protocol (TCP) **104**, User Datagram Protocol (UDP) **106**, and Inter-Processor Communication (IPC) **108**. TCP, UDP, and IPC are various suites of transmission protocols each offering different capabilities and/or functionalities. For example, UDP is a minimal message-oriented encapsulation protocol that provides no guarantees to the upper layer protocol for message delivery and the UDP layer retains no state of UDP messages once sent. UDP is commonly used for real-time, interactive applications (e.g., video chat, voice over IP (VoIP)) where loss of packets is acceptable. In contrast, TCP provides reliable, ordered, and error-checked delivery of data via a retransmission and acknowledgement scheme; TCP is generally used for file transfers where packet loss is unacceptable, and transmission latency is flexible.

[0083] As used herein, the term "encapsulation protocol" may refer to modular communication protocols in which

logically separate functions in the network are abstracted from their underlying structures by inclusion or information hiding within higher level objects. For example, in one exemplary embodiment, UDP provides extra information (ports numbering).

[0084] As used herein, the term "transport protocol" may refer to communication protocols that transport data between logical endpoints. A transport protocol may include encapsulation protocol functionality.

[0085] Both TCP and UDP are commonly layered over an Internet Protocol (IP) **110** for transmission. IP is a connectionless protocol for use on packet-switched networks that provides a "best effort delivery". Best effort delivery does not guarantee delivery, nor does it assure proper sequencing or avoidance of duplicate delivery. Generally these aspects are addressed by TCP or another transport protocol based on UDP.

[0086] As a brief aside, consider a web browser that opens a webpage; the web browser application would generally open a number of network sockets to download and/or interact with the various digital assets of the webpage (e.g., for a relatively common place webpage, this could entail instantiating ~300 sockets). The web browser can write (or read) data to the socket; thereafter, the socket object executes system calls within kernel space to copy (or fetch) data to data structures in the kernel space.

[0087] As used herein, the term "domain" may refer to a self-contained memory allocation e.g., user space, kernel space. A "domain crossing" may refer to a transaction, event, or process that "crosses" from one domain to another domain. For example, writing to a network socket from the user space to the kernel space constitutes a domain crossing access.

[0088] In the context of a Berkeley Software Distribution (BSD) based networking implementation, data that is transacted within the kernel space is stored in memory buffers that are also commonly referred to as "mbufs". Each mbuf is a fixed size memory buffer that is used generically for transfers (mbufs are used regardless of the calling process e.g., TCP, UDP, etc.). Arbitrarily sized data can be split into multiple mbufs and retrieved one at a time or (depending on system support) retrieved using "scatter-gather" direct memory access (DMA) ("scatter-gather" refers to the process of gathering data from, or scattering data into, a given set of buffers). Each mbuf transfer is parameterized by a single identified mbuf.

[0089] Notably, each socket transfer can create multiple mbuf transfers, where each mbuf transfer copies (or fetches) data from a single mbuf at a time. As a further complication, because the socket spans both: (i) user space (limited privileges) and (ii) kernel space (privileged without limitation), the socket transfer verifies that each mbuf copy into/out of kernel space is valid. More directly, the verification process ensures that the data access is not malicious, corrupted, and/or malformed (i.e., that the transfer is appropriately sized and is to/from an appropriate area).

[0090] The processing overhead associated with domain crossing is a non-trivial processing cost. Processing cost affects user experience both directly and indirectly. A processor has a fixed amount of processing cycles every second; thus cycles that are used for transfer verification detract from more user perceptible tasks (e.g., rendering a video or audio stream). Additionally, processor activity consumes power; thus, increases in processing overhead increases power consumption.

[0091] Referring back to FIG. **1**, in addition to the generic TCP **104**, UDP **106**, and IPC **108** communication suites, the illustrated socket **102** also may provide access to various kernel extensions **112**. A kernel extension is a dynamically loaded bundle of executable code that executes from kernel space. Kernel extensions may be used to perform low-level tasks that cannot be performed in user space. These low-level tasks typically fall into one or more of: low-level device drivers, network filters, and/or file systems. Examples of sockets and/or extensions include without limitation: route (IP route handling), ndry (packet 802.1X handling), key (key management), unix (translations for Unix systems), kernel control, kernel events, parental controls, intrusion detection, content filtering, hypervisors, and/or any number of other kernel tasking.

[0092] Kernel extensions and public APIs enable, for example, 3rd party software developers to develop a wide variety of applications that can interact with a computer system at even the lowest layers of abstraction. For example, kernel extensions can enable socket level filtering, IP level filtering, and even device interface filtering. In the current consumer applications space, many emerging technologies now rely on closely coupled interfaces to the hardware and kernel functionality. For example, many security applications "sniff" network traffic to detect malicious traffic or filter undesirable content; this requires access to other application sandboxes (a level of privilege that is normally reserved for the kernel).

[0093] Unfortunately, 3rd partykernel extensions can be dangerous and/or undesirable. As previously noted, software applications are restricted for security and stability reasons; however the kernel is largely unrestricted. A 3rd party kernel extension can introduce instability issues because the 3rd party kernel extensions run in the same address space as the kernel itself (which is outside the purview of traditional memory read/write protections based on memory allocations). Illegal memory accesses can result in segmentation faults and memory corruptions. Furthermore, unsecure kernel extension can create security vulnerabilities that can be exploited by malware. Additionally, even where correctly used, a kernel extension can expose a user's data to the 3rd party software developer. This heightened level of access may raise privacy concerns (e.g., the 3rd party developer may have access to browsing habits, etc.).

Existing Performance Optimization Technologies—

[0094] FIG. **2** illustrates one logical representation of a computer system that implements Input/Output (I/O) network control, useful for explaining various aspects of traditional network optimization. As depicted therein, a software application **202** executing from user space opens multiple sockets **204** to communicate with e.g., a web server. Each of the sockets interfaces with a Data Link Interface Layer (DLIL) **206**.

[0095] The DLIL **206** provides a common interface layer to each of the various physical device drivers which will handle the subsequent data transfer (e.g., Ethernet, Wi-Fi, cellular, etc.). The DLIL performs a number of system-wide holistic network traffic management functions. In one such implementation, the DLIL is responsible for BSD Virtual Interfaces, IOKit Interfaces (e.g., DLIL is the entity by

which IOKit based network drivers are connected to the networking stack), Active Queue Management (AQM), flow control and advisory action, etc. In most cases, the device driver **208** may be handled by an external device (e.g., a baseband co-processor), thus the DLIL **206** is usually (but not always) the lowest layer of the network communication stack.

[0096] During normal operation, the computer system will logically segment its tasks to optimize overall system operation. In particular, a processor will execute a task, and then "context switch" to another task, thereby ensuring that any single process thread does not monopolize processor resources from start to finish. More directly, a context switch is the process of storing the state of a process, or of a thread, so that it can be restored and execution resumed from the same point later. This allows multiple processes to share a single processor. However, excessive amounts of context switching can slow processor performance down. Notably, while the present discussion is primarily discussed within the context of a single processor for ease of understanding, multi-processor systems have analogous concepts (e.g., multiple processors also perform context switching, although contexts may not necessarily be resumed by the same processor).

[0097] For example, consider the following example of a packet reception. Packets arrive at the device driver **208**A. The hardware managed by the device driver **208**A may notify the processor via e.g., a doorbell signal (e.g., an interrupt). The device driver **208**A work loop thread handles the hardware interrupt/doorbell, then signals the DLIL thread (Loop 1 **210**). The processor services the device driver **208**A with high priority, thereby ensuring that the device driver **208**A operation is not bottlenecked (e.g., that the data does not overflow the device driver's memory and/or that the device driver does not stall). Once the data has been moved out of the device driver, the processor can context switch to other tasks.

[0098] At a later point, the processor can pick up the DLIL **206** execution process again. The processor determines which socket the packets should be routed to (e.g., socket **204**A) and routes the packet data appropriately (Loop 2 **212**). During this loop, the DLIL thread takes each packet, and moves each one sequentially into the socket memory space. Again, the processor can context switch to other tasks so as to ensure that the DLIL task does not block other concurrently executed processing.

[0099] Subsequently thereafter, when the socket has the complete packet data transfer the processor can wake the user space application and deliver the packet into user space memory (Loop 3 **214**). Generally, user space applications are treated at lower priority than kernel tasks; this can be reflected by larger time intervals between suspension and resumption. While the foregoing discussion is presented in the context of packet reception, artisans of ordinary skill in the related arts will readily appreciate, given the contents of the present disclosure, that the process is substantially reversed for packet transmission.

[0100] As demonstrated in the foregoing example, context switching ensures that tasks of different processing priority are allocated commensurate amounts of processing time. For example, a processor can spend significantly more time executing tasks of relatively high priority, and service lower priority tasks on an as-needed basis. As a brief aside, human perception is much more forgiving than hardware operation. Consequently, kernel tasks are generally performed at a much higher priority than user space applications. The difference in priority between kernel and user space allows the kernel to handle immediate system management (e.g., hardware interrupts, and queue overflow) in a timely manner, with minimal noticeable impact to the user experience.

[0101] Moreover, FIG. **2** is substantially representative of every implementation of the traditional network communications stack. While implementations may vary from this illustrative example, virtually all networking stacks share substantially the same delivery mechanism. The traditional network communications stack schema (such as the BSD architecture and derivatives therefrom) have been very popular for the past 30 years due to its relative stability of implementation and versatility across many different device platforms. For example, the Assignee hereof has developed and implemented the same networking stack across virtually all of its products (e.g., MacBook®, iMac®, iPad®, and iPhone®, Apple Watch®, etc.).

[0102] Unfortunately, changing tastes in consumer expectations cannot be effectively addressed with the one-size-fits-all model and the conservative in-kernel traditional networking stack. Artisans of ordinary skill in the related arts will readily appreciate, given the contents of the present disclosure, that different device platforms have different capabilities; for example, a desktop processor has significantly more processing and memory capability than a mobile phone processor. More directly, the "one-size-fits-all" solution does not account for the underlying platform capabilities and/or application requirements, and thus is not optimized for performance. Fine-tuning the traditional networking stack for performance based on various "tailored" special cases results in an inordinate amount of software complexity which is untenable to support across the entire ecosystem of devices.

Emerging Use Cases

[0103] FIG. **3** illustrates a logical block diagram of one exemplary implementation of Transport Layer Security (TLS) (the successor to Secure Sockets Layer (SSL)), useful to explain user/kernel space integration complexities of emerging use cases.

[0104] As shown, an application executing from user space can open a Hypertext Transfer Protocol (HTTP) session **302** with a TLS security layer **304** in order to securely transfer data (Application Transport Security (ATS) services) over a network socket **306** that offers TCP/IP transport **308**, **310**.

[0105] As a brief aside, TLS is a record based protocol; in other words, TLS uses data records which are arbitrarily sized (e.g., up to 16 kilobytes). In contrast, TCP is a byte stream protocol (i.e., a byte has a fixed length of eight (8) bits). Consequently, the TCP layer subdivides TLS records into a sequentially ordered set of bytes for delivery. The receiver of the TCP byte stream reconstructs TLS records from the TCP byte stream by receiving each TCP packet, re-ordering the packets according to sequential numbering to recreate the byte stream, and extracting the TLS record from the aggregated byte stream. Notably, every TCP packet of the sequence must be present before the TLS record can be reconstructed. Even though TCP can provide reliable delivery under lossy network conditions, there are a number of situations where TLS record delivery could fail. For example, under ideal conditions TCP isolates packet loss

from its client (TLS in this example), and a single TCP packet loss should not result in failed TLS record delivery. However, the TLS layer or the application above may incorporate a timeout strategy in a manner that is unaware of the underlying TCP conditions. Thus, if there's significant packet loss in the network, the TLS timeout may be hit (and thus result in a failure to the application) even though TCP would normally provide reliable delivery.

[0106] Referring back to FIG. **3**, virtually every modern operating system executes TLS from user space when e.g., securely connecting to other network entities, inter alia, a web browser instance and a server. But existing implementations of TLS are not executed from the kernel (or other privileged software layer) due to e.g., the complexity of error handling within the kernel. However, as a practical matter, TLS would operate significantly better with information regarding the current networking conditions (held in the kernel).

[0107] Ideally, the TLS layer should set TLS record sizes based on network condition information. In particular, large TLS records can efficiently use network bandwidth, but require many successful TCP packet deliveries. In contrast, small TLS records incur significantly more network overhead, but can survive poor bandwidth conditions.

[0108] Unfortunately, networking condition information is lower layer information that is available to the kernel space (e.g., the DLIL and device drivers), but generally restricted from user space applications. Some 3rd party application developers and device manufacturers have incorporated kernel extensions (or similar operating system capabilities) to provide network condition information to the TLS user space applications; however, kernel extensions are undesirable due to the aforementioned security and privacy concerns. Alternately, some 3rd party applications infer the presence of lossy network conditions based on historic TLS record loss. Such inferences are an indirect measure and significantly less accurate and lag behind real-time information (i.e., previous packet loss often does not predict future packet loss).

[0109] FIG. **4** illustrates a logical block diagram of an exemplary implementation of a Virtual Private Network (VPN), useful to explain recursive/cross-layer protocol layer complexities of emerging use cases.

[0110] As shown, an application executing from user space can open a Virtual Private Network (VPN) session **402** over a network socket **406** that offers TCP/IP transport **408**, **410**. The VPN session is secured with Encapsulating Security Protocol (ESP) **412**. The encrypted packet is securely tunneled via TLS **404** (in user space) and recursively sent again over TCP/IP transport **408**, **410**.

[0111] As illustrated within FIG. **4**, the exemplary VPN tunnel starts in user space, crosses into kernel space, returns back to user space, and then crosses back into kernel space before being transferred. Each of the domain crossings results in costly context switches and data shuffling both of which are processor intensive and inefficient. More directly, every time data traverses from user space to kernel space, the data must be validated (which takes non-trivial processing time). Additionally, context switching can introduce significant latency while the task is suspended.

[0112] Artisans of ordinary skill in the related arts, given the contents of the present disclosure, will readily appreciate that the exemplary recursive cross layer transaction of FIG. **4** is merely illustrative of a broad range of applications which use increasingly exotic protocol layer compositions. For example, applications that traverse the application proxy/agent data path commonly require tunneling TCP (kernel space) over application proxy/agent data path (user space) over UDP/IP (kernel space). Another common implementation is IP (kernel space) over Quick UDP Internet Connections (QUIC) (user space) over UDP/IP (kernel space).

[0113] FIG. **5** illustrates a logical block diagram of an exemplary implementation of application based tuning, useful to explain various other workload optimization complexities of emerging use cases.

[0114] As shown, three (3) different concurrently executed applications (e.g., a real time application **502**, interactive application **504**, and file transfer applications **506**) in user space, each open a session over network sockets **508** (**508**A, **508**B, **508**C) that offer TCP/UDP/IP transport **510**/**512**. Depending on the type of physical interface required, the sessions are switched to BSD network interfaces (ifnet) **514** (**514**A, **514**B, **514**C) which handle the appropriate technology. Three different illustrated technology drivers are shown: Wi-Fi **516**, Bluetooth **518**, and cellular **520**.

[0115] It is well understood within the networking arts that different application types are associated with different capabilities and requirements. One such example is real time applications **502**, commonly used for e.g., streaming audio/visual and/or other "live" data. Real time data has significant latency and/or throughput restrictions; moreover, certain real time applications may not require (and/or support) retransmission for reliable delivery of lost or corrupted data. Instead, real time applications may lower bandwidth requirements to compensate for poor transmission quality (resulting in lower quality, but timely, delivered data).

[0116] Another such example is interactive applications **504**, commonly used for e.g., human input/output. Interactive data should be delivered at latencies that are below the human perceptible threshold (within several milliseconds) to ensure that the human experience is relatively seamless. This latency interval may be long enough for a retransmission, depending on the underlying physical technology. Additionally, human perception can be more or less tolerant of certain types of data corruptions; for example, audio delays below 20 ms are generally imperceptible, whereas audio corruptions (pops and clicks) are noticeable. Consequently, some interactive applications may allow for some level of error correction and/or adopt less aggressive bandwidth management mechanisms depending on the acceptable performance requirements for human perception.

[0117] In contrast to real time applications and interactive applications, file transfer applications **506** require perfect data fidelity without latency restrictions. To these ends, most file transfer technologies support retransmission of lost or corrupted data, and retransmission can have relatively long attempt intervals (e.g., on the order of multiple seconds to a minute).

[0118] Similarly, within the communication arts, different communication technologies are associated with different capabilities and requirements. For example, Wi-Fi **516** (wireless local area networking based on IEEE 802.11) is heavily based on contention based access and is best suited for high bandwidth deliveries with reasonable latency. Wi-Fi is commonly used for file transfer type applications. Bluetooth **518** (personal area networking) is commonly used for low data rate and low latency applications. Bluetooth is

commonly used for human interface devices (e.g., headphones, keyboards, and mouses). Cellular network technologies **520** often provide non-contention based access (e.g., dedicated user access) and can be used over varying geographic ranges. Cellular voice or video delivery is a good example of streaming data applications. Artisans of ordinary skill in the related arts will readily recognize that the foregoing examples are purely illustrative, and that different communication technologies are often used to support a variety of different types of application data. For example, Wi-Fi **516** can support file transfer, real time data transmission and/or interactive data with equivalent success.

[0119] Referring back to FIG. **5**, the presence of multiple concurrently executing applications of FIG. **5** (real time application **502**, interactive application **504**, and file transfer applications **506**) illustrates the complexities of multi-threaded operation. As shown therein, the exemplary multi-threaded operation incurs a number of server loops. Each server loop represents a logical break in the process during which the processor can context switch (see also aforementioned discussion of Existing Performance Optimization Technologies, and corresponding FIG. **2**).

[0120] Moreover, in the computing arts, a "locking" synchronization mechanism is used by the kernel to enforce access limits (e.g., mutual exclusion) on resources in multi-threaded execution. During operation, each thread acquires a lock before accessing the corresponding locked resources data. In other words, at any point in time, the processor is necessarily limited to only the resources available to its currently executing process thread.

[0121] Unfortunately, each of the applications has different latency, throughput and processing utilization requirements. Since, each of the network interfaces is sending and receiving data at different times, in different amounts, and with different levels of priority. From a purely logistical standpoint, the kernel is constantly juggling between high priority kernel threads (to ensure that the high priority hardware activities do not stall out) while still servicing each of its concurrently running applications to attempt to provide acceptable levels of service. In some cases, however, the kernel is bottlenecked by the processor's capabilities. Under such situations, some threads will be deprioritized; currently, the traditional networking stack architecture is unable it clearly identify which threads can be deprioritized while still providing acceptable user service.

[0122] For example, consider an "expected use" device of FIG. **5**; the processor is designed for the expected use case of providing streaming video. Designing for expected use cases allows the device manufacturer to use less capable, but adequate components thereby reducing bill of materials (BOM) costs and/or offering features at a reasonable price point for consumers. In this case, a processor is selected that nominally meets the requirements for a streaming video application that is receiving streaming video data via one of the network interfaces (e.g., the Wi-Fi interface), and constantly servicing the kernel threads associated with it. Rendering the video with a real time application **502** from the received data is a user space application that is executed concurrently but at a significantly lower priority. During expected usage, the video rendering is adequate.

[0123] Unfortunately, the addition of an unexpected amount of additional secondary interactive applications **504** (e.g., remote control interface, headphones, and/or other interface devices) and/or background file transfer applications can easily overwhelm the processor. Specifically, the primary real time application does not get enough CPU cycles to run within its time budget, because the kernel threads handling networking are selected at a higher priority. In other words, the user space application is not able to depress the priority of kernel networking threads (which are servicing both the primary and secondary processes). This can result in significantly worse user experience when the video rendering stalls out (video frame misses or video frame drops); whereas simply slowing down a file transfer or degrading the interaction interface may have been preferable.

[0124] Prior art solutions have tailored software for specific device implementations (e.g., the Apple TV®). For example, the device can be specifically programmed for an expected use. However, tailored solutions are becoming increasingly common and by extension the exceptions have swallowed the more generic use case. Moreover, tailored solutions are undesirable from multiple software maintenance standpoints. Devices have limited productive lifetimes, and software upkeep is non-trivial.

[0125] Ideally, a per-application or per-profile workload optimization would enable a single processor (or multiple processors) to intelligently determine when and/or how too intelligently context switch and/or prioritize its application load (e.g., in the example of FIG. **5**, to prioritize video decode). Unfortunately, such solutions are not feasible within the context of the existing generic network sockets and generic network interfaces to a monolithic communications stack.

Exemplary Networking Architecture—

[0126] A networking stack architecture and technology that caters to the needs of non-kernel based networking use cases is disclosed herein. Unlike prior art monolithic networking stacks, the exemplary networking stack architecture described hereinafter includes various components that span multiple domains (both in-kernel, and non-kernel), with varying transport compositions, workload characteristics and parameters.

[0127] In one exemplary embodiment, a networking stack architecture is disclosed that provides an efficient infrastructure to transfer data across domains (user space, non-kernel, and kernel). Unlike the traditional networking paradigm that hide the underlying networking tasks within the kernel and substantially limits control thereof by any non-kernel applications, the various embodiments described herein enable faster and more efficient cross domain data transfers.

[0128] Various embodiments of the present disclosure provide a faster and more efficient packet input/output (I/O) infrastructure than prior art techniques. Specifically, unlike traditional networking stacks that use a "socket" based communication, disclosed embodiments can transfer data directly between the kernel and user space domains. Direct transfer reduces the per-byte and per-packet costs relative to socket based communication. Additionally, direct transfer can improve observability and accountability with traffic monitoring.

[0129] In one such variant, a simplified data movement model that does not require mbufs (memory buffers) is described in greater detail herein. During one such exemplary operation, the non-kernel processes can efficiently transfer packets directly to and from the in-kernel drivers.

[0130] In another embodiment, a networking stack architecture is disclosed that exposes the networking protocol stack infrastructure to user space applications via network extensions. In one such embodiment, the network extensions are software agents that enable extensible, cross-platform-capable, user space control of the networking protocol stack functionality. In another such embodiment, an in-process user space networking stack facilitates tighter integration between the protocol layers (including TLS) and the application or daemon. In some cases, the user space architecture can expose low-level networking interfaces to transport protocols and/or encapsulation protocols such as UDP, TCP, and QUIC; and enable network protocol extensions and rapid development cycles. Moreover, artisans of ordinary skill in the related arts, given the contents of the present disclosure, will readily appreciate that the various principles described herein may be applied to a variety of other operating systems (such as Windows, Linux, Unix, Android), and/or other cross platform implementations.

[0131] In some variants, exemplary embodiments of the networking stack can support multiple system-wide networking protocol stack instances (including an in-kernel traditional network stack). Specifically, in one such variant, the exemplary networking stack architecture coexists with the traditional in-kernel networking stack so as to preserve backwards compatibility for legacy networking applications. In such implementations, the in-kernel network stack instance can coexist with the non-kernel network stack via namespace sharing and flow forwarding.

[0132] As used herein, an "instance" may refer to a single copy of a software program or other software object; "instancing" and "instantiations" refers to the creation of the instance. Multiple instances of a program can be created; e.g., copied into memory several times. Software object instances are instantiations of a class; for example, a first software agent and second software instance are each distinct instances of the software agent class.

[0133] In one such implementation, load balancing for multiple networking stacks is handled within the kernel, thereby ensuring that no single networking stack (including the in-kernel stack) monopolizes system resources.

[0134] As a related variant, current/legacy applications can be handled within the in-kernel stack. More directly, by supporting a separate independent in-kernel BSD stack, legacy applications can continue to work without regressions in functionality and performance.

[0135] FIG. **6** illustrates one logical representation of an exemplary networking stack architecture, in accordance with the various aspects of the present disclosure. While the system depicts a plurality of user space applications **602** and/or legacy applications **612**, artisans of ordinary skill will readily appreciate given the contents of present disclosure that the disclosed embodiments may be used within single application systems with equivalent success.

[0136] As shown, a user space application **602** can initiate a network connection by instancing user space protocol stacks **604**. Each user space protocol stacks includes network extensions for e.g., TCP/UDP/QUIC/IP, cryptography, framing, multiplexing, tunneling, and/or any number of other networking stack functionalities. Each user space protocol stack **604** communicates with one or more nexuses **608** via a channel input/output (I/O) **606**. Each nexus **608** manages access to the network drivers **610**. Additionally shown is legacy application **612** support via existing network socket technologies **614**. While the illustrated embodiment shows nexus connections to both user space and in-kernel networking stacks, it is appreciated that the nexus may also enable e.g., non-kernel networking stacks (such as may be used by a daemon or other non-kernel, non-user process).

[0137] The following topical sections hereinafter describe the salient features of the various logical constructs in greater detail.

[0138] Exemplary I/O Infrastructure

[0139] In one exemplary embodiment, the non-kernel networking stack provides a direct channel input output (I/O) **606**. In one such implementation, the channel I/O **606** is included as part of the user space protocol stack **604**. More directly, the channel I/O **606** enables the delivery of packets as a raw data I/O into kernel space with a single validation (e.g., only when the user stack provides the data to the one or more nexuses **608**). The data can be directly accessed and/or manipulated in situ, the data need not be copied to an intermediary buffer.

[0140] In one exemplary implementation, a channel is an I/O scheme leveraging kernel-managed shared memory. During an access, the channel I/O is presented to the process (e.g., the user process or kernel process) as a file descriptor based object, rather than as data. In order to access the data, the process de-references the file descriptor for direct access to the shared memory within kernel space. In one such implementation, the file descriptor based object based I/O is compatible with existing operating system signaling and "eventing" (event notification/response) mechanisms. In one exemplary variant, the channel I/O is based on Inter Process Communication (IPC) packets.

[0141] As used herein, the term "descriptor" may refer to data structures that indicate how other data is stored. Descriptors generally include multiple parameters and can be used to identify more complex data structures; for example, a descriptor may include one or more of type, size, address, tag, flag, headers, footers, metadata, structural links to other data descriptors or locations, and/or any other number of format or construction information.

[0142] Within the context of the present disclosure, as used herein, the term "pointer" may refer to a specific reference data type that "points" or "references" a location of data in memory. Typically, a pointer stores a memory address that is interpreted by a compiler as an absolute location in system memory or a relative location in system memory based on e.g., a base address, reference address, memory window, or other memory subset. During operation, a pointer is "de-referenced" to recover the data that is stored in the location of memory.

[0143] As used herein, the term "metadata" refers to data that describes data. Metadata varies widely in application, but generally falls into one of the descriptive, structural, and/or administrative categories. Descriptive metadata describes data in a manner to enable e.g., discovery and/or identification. Common examples include without limitation e.g., type, size, index tags, and keywords. Structural metadata describes the structure of the data e.g., how compound objects are put together. Common examples include without limitation e.g., prefix, postfix, table of contents, order, and/or any other information that describes the relationships and other characteristics of digital materials. Administrative metadata provides information to help manage a resource; common examples include e.g., authorship and creation

information, access privileges, and/or error checking and security based information (e.g., cyclic redundancy checks (CRC), parity, etc.).

[0144] In one exemplary embodiment, the channel I/O can be further leveraged to provide direct monitoring of its corresponding associated memory. More directly, unlike existing data transfers which are based on mbuf based divide/copy/move, etc., the channel I/O can provide (with appropriate viewing privileges) a direct window into the memory accesses of the system. Such implementations further simplify software development as debugging and/or traffic monitoring can be performed directly on traffic. Direct traffic monitoring can reduce errors attributed to false positives/false negatives caused by e.g., different software versioning, task scheduling, compiler settings, and/or other software introduced inaccuracies.

[0145] More generally, unlike prior art solutions which relied on specialized networking stack compositions to provide different degrees of visibility at different layers, the monitoring schemes of the present disclosure provide consistent system-wide channel monitoring infrastructures. Consistent frameworks for visibility, accounting, and debugging greatly improve software maintenance and upkeep costs.

[0146] Additionally, simplified schemes for egress filtering can be used to prevent traffic spoofing for user space networking stack instances. For example, various embodiments ensure that traffic of an application cannot be hijacked by another malicious application (by the latter claiming to use the same tuple information, e.g. TCP/UDP port).

[0147] In one exemplary embodiment, the in-kernel network device drivers (e.g. Wi-Fi, Cellular, Ethernet) use simplified data movement models based on the aforementioned channel I/O scheme. More directly, the user space networking stacks can directly interface to each of the various different technology based network drivers via channel I/O; in this manner, the user space networking stacks do not incur the traditional data mbuf based divide/copy/move penalties. Additionally, user space applications can directly access user space networking components for immediate traffic handling and processing.

Exemplary Nexus—

[0148] In one exemplary embodiment, the networking stack connects to one or more nexus **608**. In one such implementation, the nexus **608** is a kernel space process that arbitrates access to system resources including, without limitation e.g., shared memory within kernel space, network drivers, and/or other kernel or user processes. In one such variant, the nexus **608** aggregates one or more channels **606** together for access to the network drivers **610** and/or shared kernel space memory.

[0149] In one exemplary implementation, a nexus is a kernel process that determines the format and/or parameters of the data flowing through its connected channels. In some variants, the nexus may further perform ingress and/or egress filtering.

[0150] The nexus may use the determined format and/or parameter information to facilitate one-to-one and one-to-many topologies. For example, the nexus can create user-pipes for process-to-process channels; kernel-pipes for process-to-kernel channels; network interfaces for direct channel connection from a process to in-kernel network drivers, or legacy networking stack interfaces; and/or flow-switches for multiplexing flows across channels (e.g., switching a flow from one channel to one or more other channels).

[0151] Additionally, in some variants the nexus may provide the format, parameter, and/or ingress egress information to kernel processes and/or one or more appropriately privileged user space processes.

[0152] In one exemplary embodiment, the nexus **608** may additionally ensure that there is fairness and/or appropriately prioritize each of its connected stacks. For example, within the context of FIG. **6**, the nexus **608** balances the network priorities of both the existing user space application networking stacks **604**, as well as providing fair access for legacy socket based access **614**. For example, as previously alluded to, existing networking stacks could starve user space applications because the kernel threads handling the legacy networking stack operated at higher priorities than user space applications. However, the exemplary nexus **608** ensures that legacy applications do not monopolize system resources by appropriately servicing the user space network stacks as well as the legacy network stack.

[0153] In one such embodiment, in-kernel, non-kernel, and/or user space infrastructures ensure fairness and can reduce latency due to e.g., buffer bloat (across channels in a given nexus, as well as flows within a channel). In other words, the in-kernel and/or user space infrastructures can negotiate proper buffering sizes based on the expected amount of traffic and/or network capabilities for each flow. By buffering data according to traffic and/or network capability, buffers are not undersized or oversized.

[0154] As a brief aside, "buffer bloat" is commonly used to describe e.g., high latency caused by excessive buffering of packets. Specifically, buffer bloat may occur when excessively large buffers are used to support a real time streaming application. As a brief aside, TCP retransmission mechanism relies on measuring the occurrence of packet drops to determine the available bandwidth. Under certain congestion conditions, excessively large buffers can prevent the TCP feedback mechanism from correctly inferring the presence of a network congestion event in a timely manner (the buffered packets "hide" the congestion, since they are not dropped). Consequently, the buffers have to drain before TCP congestion control resets and the TCP connection can correct itself.

[0155] Referring back to FIG. **6**, in one exemplary embodiment, Active Queue Management (AQM) can be implemented in the kernel across one or more (potentially all) of the flow-switch clients (user space and in-kernel networking stack instances). AQM refers to the intelligent culling of network packets associated with a network interface, to reduce network congestion. By dropping packets before the queue is full, the AQM ensures no single buffer approaches its maximum size, and TCP feedback mechanisms remain timely (thereby avoiding the aforementioned buffer bloat issues).

[0156] While the foregoing example is based on "fairness" standard, artisans of ordinary skill in the related arts will readily appreciate that other schemes may be substituted with equivalent success given the contents of the present disclosure. For example, some embodiments may dynamically or statically service the user application networking space with greater or less weight compared to the legacy socket based access. For example, user application networking space may be more heavily weighted to improve overall

performance or functionality, whereas legacy socket based access may be preferred where legacy applications are preferentially supported (e.g., see Protocol Unloading Offloading, discussed infra).

Exemplary Network Extensions

[0157] In one exemplary embodiment of the present disclosure, a network extension is disclosed. A network extension is an agent-based extension that is tightly coupled to network control policies. The agent is executed by the kernel and exposes libraries of network control functionality to user space applications. During operation, user space software can access kernel space functionality through the context and privileges of the agent.

[0158] As used herein, the term "agent" may refer to a software agent that acts for a user space application or other program in a relationship of agency with appropriate privileges. The agency relationship between the agent and the user space application implies the authority to decide which, if any, action is appropriate given the user application and kernel privileges. A software agent is privileged to negotiate with the kernel and other software agents regarding without limitation e.g., scheduling, priority, collaboration, visibility, and/other sharing of user space and kernel space information. While the agent negotiates with the kernel on behalf of the application, the kernel ultimately decides on scheduling, priority, etc.

[0159] Various benefits and efficiencies can be gained through the use of network extensions. In particular, user space applications can control the protocol stack down to the resolution of exposed threads (i.e., the threads that are made available by the agent). In other words, software agents expose specific access to lower layer network functionality which was previously hidden or abstracted away from user space applications. For example, consider the previous examples of TLS record sizing (see e.g., FIG. **3**, and related discussion); by exposing TCP network conditions to the TLS application within the user space, the TLS application can correctly size records for network congestion and/or wait for underlying TCP retransmissions (rather than timing out).

[0160] Similarly, consider the previous examples of multi-threading within the context of expected use devices (see e.g., FIG. **5**, and related discussion); the primary user space application (e.g., video coding) and additional secondary interactive applications (e.g., remote control interface, headphones, and/or other interface devices) can internally negotiate their relative priority to the user's experience. The user space applications can appropriately adjust their priorities for the nexus (i.e., which networking threads are serviced first and/or should be deprioritized). Consequently, the user space applications can deprioritize non-essential network accesses, thereby preserving enough CPU cycles for video decode.

[0161] As a related benefit, since a software agent represents the application to the kernel; the agent can trust the kernel, but the kernel may or may not trust the agent. For example, a software agent can be used by the kernel to convey network congestion information in a trusted manner to the application; similarly, a software agent can be used by an application to request a higher network priority. Notably, since a software agent operates from user space, the agent's privilege is not promoted to kernel level permissions. In other words, the agent does not permit the user application to exceed its privileges (e.g., the agent cannot commandeer the network driver at the highest network priority, or force a read/write to another application's memory space without the other kernel and/or other application's consent).

[0162] Networking extensions allow the user space application to execute networking communications functionality within the user space and interpose a network extension between the user space application and the kernel space. As a result, the number of cross domain accesses for complex layering of different protocol stacks can be greatly reduced. Limiting cross domain accesses prevents context switching and allows the user space to efficiently police its own priorities. For example, consider the previous example of a VPN session as was previously illustrated in FIG. **4**. By keeping the TCP/IP, Internet Protocol Security (IPsec) and TLS operations within user space, the entire tunnel can be performed within the user space, and only cross the user/kernel domain once.

[0163] As used herein, the term "interposition" may refer to the insertion of an entity between two or more layers. For example, an agent is interposed between the application and the user space networking stack. Depending on the type of agent or network extension, the interposition can be explicit or implicit. Explicit interposition occurs where the application explicitly instances the agent or network extension. For example, the application may explicitly call a user space tunnel extension. In contrast, implicit interposition occurs where the application did not explicitly instance the agent or network extension. Common examples of implicit interposition occur where one user space application sniffs the traffic or filters the content of another user space application.

Namespace Sharing & Flow Forwarding Optimizations

[0164] In one exemplary optimization of the present disclosure, the nexus includes a namespace registration and management component that manages a common namespace for all of its connected networking stack instances. As a brief aside, a namespace generally refers to a set of unique identifiers (e.g., the names of types, functions, variables) within a common context. Namespaces are used to prevent naming "collisions" which occur where multiple processes call the same resource differently and/or call different resources the same.

[0165] In one such implementation, the shared networking protocol has a common namespace (e.g., {Address, Protocol, and Port}) across multiple networking stack instances. Sharing a namespace between different networking stacks reduces the amount of kernel burden, as the kernel can natively translate (rather than additionally adding a layer of network address translation).

[0166] For example, if a first application acquires port **80**, the namespace registration ensures that other applications will not use port **80** (e.g., they can be assigned e.g., port **81**, **82**, etc.) In some such implementations, legacy clients may use default namespaces that conflict (e.g., a default web client may always select port **80**); thus the shared namespace registration may also be required to force a re-assignment of a new identifier (or else translate for) such legacy applications.

[0167] In one exemplary embodiment, the namespace registration and management components control flow-switching and forwarding logic of each flow-switch nexus instance. For example, as previously noted, the nexus can create user-pipes for process-to-process channels; kernel-pipes for process-to-kernel channels; network interfaces for

direct channel connection from a process to in-kernel network drivers, or legacy networking stack interfaces; and/or flow-switches for multiplexing flows across channels (e.g., switching a flow from one channel to one or more other channels).

[0168] For example, during normal operation when an application requests a port, the namespace registration and management will create a flow and assign a particular port to the application. Subsequent packets addressed to the port will be routed appropriately to the flow's corresponding application. In one such variant, packets that do not match any registered port within the shared namespace registration and management will default to the legacy networking stack (e.g., the flow-switch assumes that the unrecognized packet can be parsed and/or ignored by the fallback legacy stack).

[0169] Artisans of ordinary skill in the related arts will readily appreciate, given the contents of the present disclosure that disparate and/or otherwise distinct namespace registrations and/or management components may be preferable based on other implementation specific considerations. For example, some implementations may prefer to shield namespaces from other external processes e.g., for security and/or privacy considerations. In other implementations, the benefits associated with native namespace translation may be less important than supporting legacy namespaces.

Protocol Onloading and Offloading

[0170] In the foregoing discussions, the improvements to user space operation may be primarily due to the user space networking stack, as shown in FIG. **6**. However, various embodiments of the present disclosure also leverage the existing legacy host networking infrastructure to handle networking transactions which are unrelated to user experience.

[0171] Colloquially, the term "hardware offload" may be commonly used to denote tasks which can be handled within dedicated hardware logic to improve overall processing speed or efficiency. One such example is the cyclic redundancy check (CRC) calculation which is an easily parameterized, closed, iterative calculation. The characteristics of CRC calculation lend itself to hardware offload because the CRC does not benefit from the flexibility of a general purpose processor, and CRC calculations are specialized functions that are not transferable to other processing operations.

[0172] By analogous extension, as used herein, the term "protocol offload" may refer to processes that should be handled within the legacy networking stack because they are not specific to a user space application or task. In contrast, the term "protocol onload" may refer to processes that should be handled within a user space networking stack because they are specific to a user space application or task and benefit the overall performance. As a general qualitative criteria, tasks which are "fast" (e.g., generally UDP/TCP/IP based user space applications) are protocol onloaded to improve user performance; in contrast "slow" tasks (e.g., ARP, IPv6 Neighbor Discovery, Routing table updates, control path for managing interfaces, etc.) are protocol offloaded.

[0173] For example, consider Address Resolution Protocol (ARP) request handling; when an ARP request comes in, the host processor responds with a reply. However, the ARP request is non-specific to a user space application; rather the ARP reply concerns the holistic system. More generally, any networking process that is not specific to an application space can be implemented within the kernel under legacy techniques. Alternatively, any process that can be handled regardless of device state should remain with the kernel (e.g., the kernel persists across low power states, and is never killed).

[0174] By allowing the mature in-kernel networking stack to retain ownership of certain control logic (e.g. routing and policy table, interface configuration, address management), various embodiments of the present disclosure avoid "split-brain" behaviors. In other words, the kernel ensures that networking data and/or availability remains consistent regardless of the user space application availability.

Exemplary User Space Networking Stack

[0175] Referring now to FIG. **7**, one logical block diagram of an exemplary user space networking stack **700** is depicted. As shown, the user space networking stack **700** includes an application interface **702**, and an operating system interface **704**. Additionally, the user space networking stack includes one or more user space instances of TLS **706**, QUIC **708**, TCP **710**, UDP **712**, IP **714**, and ESP **716**. The disclosed instances are purely illustrative, artisans of ordinary skill in the related arts will readily appreciate that any other user space kernel extension and/or socket functionality may be made available within the user space networking stack **700**.

[0176] In one exemplary embodiment, the user space networking stack **700** is instantiated within an application user space **718**. More directly, the user space networking stack **700** is treated identically to any one of multiple threads **710** within the application user space **718**. Each of the coexisting threads **720** has access to the various functions and libraries offered by the user space networking stack via a direct function call.

[0177] As a brief aside, each of the threads **720** reside within the same address space. By virtue of their shared addressability, each of the threads may grant or deny access to their portions of shared address space via existing user space memory management schemes and/or virtual machine type protections. Additionally, threads can freely transfer data structures from one to the other, without e.g., incurring cross domain penalties. For example, TCP data **710** can be freely passed to TLS **706** as a data structure within a user space function call.

[0178] As previously noted, the user space networking stack **700** may grant or deny access to other coexistent user space threads; e.g., a user space thread is restricted to the specific function calls and privileges made available via the application interface **702**. Furthermore, the user space networking stack **700** is further restricted to interfacing the operating system via the specific kernel function calls and privileges made available via the operating system interface **704**. In this manner, both the threads and the user space networking stack have access and visibility into the kernel space, without compromising the kernel's security and stability.

[0179] One significant benefit of the user space networking stack **700** is that networking function calls can be made without acquiring various locks that are present in the in-kernel networking stack. As previously noted, the "locking" mechanism is used by the kernel to enforce access limits on multiple threads from multiple different user space

applications; however in the user space, access to shared resources are handled within the context of only one user application space at a time, consequently access to shared resources are inherently handled by the single threading nature of user space execution. More directly, only one thread can access the user space networking stack **700** at a time; consequently, kernel locking is entirely obviated by the user space networking stack.

[0180] Another benefit of user space based network stack operation is cross platform compatibility. For example, certain types of applications (e.g., iTunes®, Apple Music® developed by the Assignee hereof) are deployed over a variety of different operating systems. Similarly, some emerging transport protocols (e.g. QUIC) are ideally served by portable and common software between the client and server endpoints. Consistency in the user space software implementation allows for better and more consistent user experience, improves statistical data gathering and analysis, and provides a foundation for enhancing, experimenting and developing network technologies used across such services. In other words, a consistent user space networking stack can be deployed over any operating system platform without regard for the native operating system stack (e.g., which may vary widely).

[0181] Another important advantage of the exemplary user space networking stack is the flexibility to extend and improve the core protocol functionalities, and thus deliver specialized stacks based on the application's requirements. For example, a video conferencing application (e.g., FaceTime® developed by the Assignee hereof) may benefit from a networking stack catered to optimize performance for real-time voice and video-streaming traffics (e.g., by allocating more CPU cycles for video rendering, or conversely deprioritizing unimportant ancillary tasks). In one such variant, a specialized stack can be deployed entirely within the user space application, without specialized kernel extensions or changes to the kernel. In this manner, the specialized user space networking stack can be isolated from networking stacks. This is important both from a reliability standpoint (e.g., updated software doesn't affect other software), as well as to minimize debugging and reduce development and test cycle times.

[0182] Furthermore, having the network transport layer (e.g. TCP, QUIC) reside in user space can open up many possibilities for improving performance. For example, as previously alluded to, applications (such as TLS) can be modified depending on the underlying network connections. User space applications can be collapsed or tightly integrated into network transports. In some variants, data structure sizes can be adjusted based on immediate lower layer network condition information (e.g., to accommodate or compensate for poor network conditions). Similarly, overly conservative or under conservative transport mechanisms can be avoided (e.g., too much or not enough buffering previously present at the socket layer). Furthermore, unnecessary data copies and/or transforms can be eliminated and protocol signaling (congestion, error, etc.) can be delivered more efficiently.

[0183] In yet another embodiment, the exemplary user space networking stack further provides a framework for both networking clients and networking providers. In one such variant, the networking client framework allows the client to interoperate with any network provider (including the legacy BSD stack). In one such variant, the network provider framework provides consistent methods of discovery, connection, and data transfer to networking clients. By providing consistent frameworks for clients and providers which operate seamlessly over a range of different technologies (such as a VPN, Bluetooth, Wi-Fi, cellular, etc.), the client software can be greatly simplified while retaining compatibility with many different technologies.

Exemplary Proxy Agent Application Operation

[0184] FIG. **8** depicts one logical flow diagram useful to summarize the convoluted data path taken for a prior art application using a proxy agent application within the context of the traditional networking stack. As shown therein, an application **802** transmits data via a socket **804**A to route data packets to a proxy agent application **814** via a TCP/IP **806/808** and a BSD network interface **810**A. The data packets enter kernel space; this is a first domain crossing which incurs validation and context switching penalties.

[0185] Inside the kernel, the data is divided/copied/moved for delivery via the TCP/IP stack **806/808** to the BSD network interface **810**A. The BSD network interface **810**A routes the data to a virtual driver **812**A. These steps may introduce buffering delays as well as improper buffer sizing issues such as buffer bloat.

[0186] In order to access the application proxy (which is in a different user space), the virtual driver reroutes the data to a second socket **804**B which is in the different user space from the original application. This constitutes a second domain crossing, which incurs additional validation and context switching penalties.

[0187] In user space, the data enters an agent **814** which prepares the data for delivery (tunneling **816**, framing **818**, and cryptographic security **820**). Thereafter, the proxy agent **814** transmits the prepared data via a socket **804**B to route data packets to a user space driver **822** via the TCP/IP **806/808** and a separate BSD network interface **810**B. Again, the data is passed through the socket **804**B. This is a third domain crossing, with validation and context switching penalties.

[0188] Inside the kernel, the data is divided/copied/moved for delivery via the TCP/IP stack **806/808** to a BSD network interface **810**B. The steps of The BSD network interface **810**B routes the data to a virtual driver **812**B. These steps introduce additional buffering delays as well as improper buffer sizing issues such as buffer bloat.

[0189] Finally, the virtual driver **812**B reroutes the data to the user space driver (e.g., a Universal Serial Bus (USB) driver), which requires another socket transfer from **804**B to **804**C; the data crosses into the user space for the user based driver **822**, and crosses the domain a fifth time to be routed out the USB Hardware (H/W) driver **824**. Each of these domain crossings are subject to the validation and context switching penalties as well as any buffering issues.

[0190] FIG. **9** depicts one logical flow diagram useful to summarize an exemplary proxy agent application within the context of the user space networking stack, in accordance with the various aspects of the present disclosure.

[0191] As shown therein, an application **902** provides data via shared memory space file descriptor objects to the agent **904**. The agent **904** internally processes the data via TCP/IP **906/908** to the tunneling function **910**. Thereafter, the data is framed **912**, cryptographically secured **914**, and routed via TCP/IP **906/908** to the user driver **916**. The user driver uses a channel I/O to communicate with nexus **918** for the one

(and only) domain crossing into kernel space. Thereafter, the nexus **918** provides the data to the H/W driver **920**.

[0192] When compared side-by-side, the user space networking stack **900** has only one (1) domain crossing, compared to the traditional networking stack **800** which crossed domains five (5) times for the identical VPN operation. Moreover, each of the user space applications could directly pass data via function calls within user memory space between each of the intermediary applications, rather than relying on the kernel based generic mbuf divide/copy/move scheme (and its associated buffering inefficiencies).

Security Enhancements—

[0193] Traditionally, communication stack operation has been handled within the kernel space. Even though exposing communication stack operation to user space applications provides a plethora of benefits, the differences in treatment, privileges, privacy, priority, and security between user and kernel space operation are substantial. User space applications are commonly exposed to, and attacked by, malicious 3rd parties (e.g., hackers) and/or unwanted intrusion (e.g., sniffing, surreptitious monitoring). The following discussions elaborate various schemes and countermeasures for ensuring that user and/or kernel space applications are protected from undesirable behaviors.

Metadata Red Zones

[0194] Within the context of user space generated data, contiguous or adjacent memory objects could be attacked and/or inadvertently corrupted due to buffer overrun issues. In a shared memory architecture the ability for the consumer of the data (e.g., a user or kernel space process) to detect these issues in the least expensive manner is important.

[0195] FIG. **10** is a graphical comparison of a packet data structure **1000** during normal operation, and a buffer overrun **1050**.

[0196] As shown, the normal packet data structure **1000** includes a sequence of packets that are contiguously arranged. For example, packet [**0**] is adjacent to packet [**1**], etc. Each packet includes metadata **1004** that describes the data payload **1006**. For example, a TCP/IP packet may include TCP/IP packet descriptor information. In the illustrated embodiment, the metadata **1004** is stored in a header (read before the data payload **1006**); in other embodiments, the metadata may be stored in a footer (read after the data payload **1006**). Still other embodiments may have metadata in both header and footer locations (e.g., a TCP/IP header, and a CRC footer).

[0197] During a buffer overrun **1050**, a first packet overwrites the contents of an adjacent packet. For example, the data payload **1008** may extend into its neighbor packet, corrupting the overwritten packet **1010**. In some cases, buffer overruns may be an unintentional error (caused by software bugs); unintentional overruns can be detected but are undesirable in that the corrupted data packet cannot be reconstructed. More worrisome however, is where the buffer overrun is intentionally caused by malicious software. Notably, metadata may be easy to predict and falsify; for example, a TCP/IP packet header can be reconstructed with source/destination addresses and packet sequence numbers. By masking a buffer overrun with bogus TCP/IP packet header information, a malicious attacker could hide from overrun detection mechanisms and/or create bogus packets that appear legitimate. Consequently, various embodiments of the present disclosure include user space network stack infrastructure data descriptors (also known as packet or quantum) that have a metadata preamble placed at the beginning of the object. This metadata preamble is used to detect any inadvertent overwrite of the metadata. Each metadata object may include a unique red zone pattern which, in one such variant, is the XOR of a red zone cookie and the offset of the metadata object in the object's memory region. Red zone cookies are initialized with random numbers on an OS boot. In the event the kernel detects a corruption, the user space process associated with the channel may be terminated (or other corrective action) to prevent further damages.

[0198] FIG. **11** is a graphical comparison of an exemplary packet data structure **1100** that includes a red zone pattern, and an exemplary buffer overrun **1150** that can be detected based on red zone pattern corruption.

[0199] As before, the normal packet data structure **1100** stores packets contiguously and in sequence; the metadata **1104** describes the data payload **1106**. In the illustrated embodiment, the metadata **1104** further includes a red zone pattern (e.g., the XOR of a red zone cookie and the offset of the metadata object in the object's memory region). The purpose of red zone is to detect inadvertent memory corruption inexpensively. In some variants, the red zone verification may also be done in the user space via APIs which are used to access the user packet metadata (e.g., the libsyscall wrapper functions provided via user space library). More generally, red zone checks can be used as an inexpensive check for the library (or any other process) to determine if a packet is corrupted. In order to verify that the packet has not been overwritten, any kernel process can XOR the red zone pattern with the memory offset; if anything other than the red zone cookie results, then the kernel process can flag an overrun. XOR operations are inexpensive, and the flow-switch can easily check red zone patterns before forwarding user space packets to other kernel space entities (e.g., driver packet pools, or other processes).

[0200] In the illustrated exemplary buffer overrun **1150**, a buffer overrun **1108** that has corrupted user space data **1110** will have an invalid red zone pattern **1112**. As a result, the kernel process can take remedial measures; for example, either or both of the packet [**0**] and packet [1] may be discarded and/or the user process may be terminated.

[0201] FIG. **12** is a logical block diagram of a user space packet and its corresponding components in kernel space. As shown therein, the user space packet **1202** is composed of user space metadata **1204** and user space data payload **1206**; the user space metadata **1204** includes a red zone pattern **1210**. The user space metadata is copied into kernel space as a metadata user space (MDU) object. Specifically, the user space data payload **1206** is copied into buffers **1214**; and corresponding buflets **1212** provide offset references thereto. Notably, in one exemplary variant, the MDU object uses offsets from a base address rather than pointer addresses.

[0202] Kernel space processes do not trust (and thus cannot directly use) the metadata in MDU objects. Thus, the kernel space process (e.g., a flow-switch) validates and sanitizes the MDU data. In one exemplary embodiment, the validation process includes checking the red zone pattern by XOR with the memory offset. If the XOR results in the red zone cookie, then the MDU object has not been corrupted by a buffer overrun. The resulting data is translated to a

metadata kernel space (MDK) object **1216**. The MDK object includes a sanitized version of the user space metadata **1218**, any kernel space metadata **1220**, and buflets **1213**. Unlike buflets **1212** which use offsets from a base address, the kernel space buflets **1213** are trusted and may use pointers that can be de-referenced to memory addresses.

[0203] Artisans of ordinary skill in the related arts given the contents of the present disclosure will appreciate that any technique for generating and checking for buffer overruns may be substituted with equivalent success. For example, instead of a red zone cookie, a temporary random value, cryptographically generated seed, nonce, or other obfuscator value may be used as a preamble/post-amble. Similarly, while XOR is inexpensive from a processing standpoint, more or less computationally expensive operations may be used. For example, some techniques may use e.g., cryptographic hashes, factorizations.

[0204] While the illustrated example is shown with respect to a buffer overrun, artisans of ordinary skill in the related arts will further appreciate that the red zone pattern also detects buffer underrun type attacks. For example, even if a malicious attacker shortened a packet (underrun) and inserted a bogus packet in the remaining space; the bogus packet would have an invalid red zone pattern. In other words, the disclosed solutions protect against any malicious attack that relies on packet formatting.

Internalization and Externalization

[0205] As previously alluded to, sharing user space generated metadata (e.g., packet descriptors, etc.) with kernel space processes (and vice versa) could be prone to security and privacy vulnerabilities. Thus, various embodiments of the present disclosure "internalize" metadata from the user space to the kernel space and/or "externalize" kernel space metadata for use in user space.

[0206] As used herein, the terms "internal", "internalize", and "internalization" refer to the process of propagating information from the user space to the kernel space. In contrast, the process of propagating information from the kernel space to the user space is referred to with the terms "external", "externalize", and "externalization."

[0207] In one exemplary embodiment, a user space network stack infrastructure architecture maintains a first packet descriptor memory in user space and a mirrored copy of the packet descriptor memory which is accessible only from the kernel. During packet handoff from the user space to kernel space, the first packet descriptor memory in user space is validated against the kernel copy for any semantic issues and the sanitized data is copied to the kernel's mirrored packet descriptor memory. Similarly, during packet handoff from the kernel space to user space, the kernel's mirrored packet descriptor memory is scrubbed of any sensitive information before it is written back to the packet descriptor memory in user space.

[0208] As previously noted, the access rights and privileges of the user space are different than the kernel space. For example, the kernel space must manage and track information for the entire system, including other user space processes. Moreover, the kernel space controls all device functionality. As a result, the kernel space has access to data that is sensitive, private, and/or device critical. With few exceptions, user space processes are limited to manipulating only their own data and functionality.

[0209] Within the context of user space network stack architectures, the user space generates metadata that may be useful for other kernel space processes. For example, packet descriptor information may be used by the kernel space drivers to control packet delivery. Ideally, the kernel space can enforce access limitations and/or obfuscate sensitive kernel activity from the user space, while still leveraging the packet descriptor information that is generated by user space network stack processing. More generally, artisans of ordinary skill in the related arts will appreciate, given the contents of the present disclosure, that user space data can be leveraged for other process (both kernel and user space), provided that security and privacy protections judiciously remain intact.

[0210] FIG. **13** is a logical block diagram of a user space packet, its corresponding components in kernel space, and secure and private translations therebetween via e.g., internalization and externalization. As shown therein, the user space packet **1302** is composed of user space metadata **1304** and user space data payload **1306**.

[0211] During internalization, the user space metadata **1304** is copied into the metadata user space object (MDU) **1308** and its corresponding data payload **1306** is internalized into buffers **1314**. In one exemplary embodiment, internalization includes converting the MDU **1308** into a metadata kernel space object (MDK) **1316**. During conversion, the MDU **1308** is checked for validity and/or translated before updating the MDK data **1316**.

[0212] As used herein, the terms "valid", "validate", and "validity" refer to the process of ensuring that data has not been tampered with or corrupted. In some embodiments, validity can be measured when data is internally consistent (external verification of the data is not necessary). In more rigorous implementations, validity may require external verification with e.g., a trusted third party. For example, validity may include trust certificates, verification of digital signatures, encryption/decryption, and/or other security measures. More generally, artisans of ordinary skill in the related arts will appreciate that validity may balance both processing cost, memory consumption, power consumption, confidence, data integrity, reliability, robustness, and/or any number of other system considerations. Notably, the MDU **1308** and MDK **1316** have different functions, and thus are often composed of different data types and/or structures. Consequently, validity checks may entail more rigor than direct comparison. For example, in one exemplary embodiment, the MDK **1316** includes buflets **1313** that use pointers to locations in kernel virtual address (KVA), whereas the MDU **1310** uses offsets from a base address (which obscures sensitive KVA information from the user space). Under such embodiments, the MDU buflets **1312** are translated from offsets to pointers and checked to ensure that the pointers are acceptable (e.g., within the memory allocation allocated for the user space process). In another such example, MDU **1310** may have internal formatting information that should be consistent (e.g., the internal formatting information matches the data structure format). For example, packet descriptors often include e.g., length information and/or other information used for validation. In one exemplary embodiment, the MDU handed over to the kernel contains offsets to data that can be validated with a checksum computation. The kernel computes the checksum during the copy operation into the device memory pool. During internalization processes, the offset location may be checked to

ensure that it lies within the bounds of the packet buffer. The metadata can also include offsets to protocol headers in the packet buffer, internalize operation validates these bounds too. Validity checks may include checking that the length is consistent and that the offsets are in expected locations.

[0213] In one exemplary embodiment, if the MDU **1310** passes the validity check, then the MDK **1316** is updated with the MDU **1310** values. Notably, exemplary embodiments of the MDK **1316** are not overwritten with MDU **1310** values. As previously noted, the MDK **1316** is managed by the kernel which operates at a higher privilege than the MDU **1310**; thus, the MDK **1316** may or may not always accept the MDU **1310** values. For example, a MDU **1310** may attempt (but be prohibited from) changing e.g., a data stream priority, etc. In alternative systems, the MDK **1316** may (by default) be overwritten with MDU **1310**; such embodiments may be particularly useful where the user space application is trusted to some degree (e.g., the user space application may be a 1st party application or a trusted 2nd party application)

[0214] Artisans of ordinary skill in the related arts will appreciate that validity checks balance security and privacy with processing complexity; the degree of validation must be balanced against other considerations. Existing system parameters (e.g., privileges and/or per missioning schemes) may also affect the internalization process; for example, a kernel space may differentiate between user space metadata **1308** that overwrites the sanitized user space metadata **1318**, and user space metadata **1308** that may be used to update the kernel space metadata **1320** with discretion. In some cases, the kernel may use more stringent checks for overwriting, whereas less dangerous updates may be more lax.

[0215] If the MDU **1310** fails the validity check, then a number of remedial actions may be taken. In some embodiments, a failed validity check may result in a draconian termination to prevent further corruption and/or system resource consumption by the errant user space process. In other embodiments, a failed validity check may result in dropping the packet and/or ignoring the metadata. More lenient remedies may be warranted where failure is not fatal and/or where failure is expected. For example, certain network protocols may aggregate multiple data streams into a single network channel. Such systems often allow for, and implement recovery schemes where a single data stream fails (e.g., the aggregate data stream should not be terminated due to a single stream failure).

[0216] Within the context of user space networking stacks, the user space data is not part of the data path and generally not tampered with. Thus, exemplary embodiments of the present disclosure minimize processing load by internalizing user space data with little or no validation. However, artisans of ordinary skill in the related arts, given the contents of the present disclosure, will appreciate that more complex internalization techniques may be used where malicious user space data could result in adverse behavior. For example, user space drivers could cause errant driver misbehavior; such applications may warrant validity checking the user space data payloads.

[0217] Referring back to FIG. **13**, the externalization process allows kernel information to be externalized to the user space. Generally, the kernel is secure and trusted relative to the user space. Consequently, various embodiments of the present disclosure are primarily concerned with ensuring that the kernel does not leak sensitive information to the user space.

[0218] As shown in FIG. **13**, the MDK object includes both sanitized user space metadata **1318** and kernel space metadata **1320**. In one exemplary embodiment, only the sanitized user space data **1318** can be externalized. In other embodiments, both the sanitized user space data **1318** and the kernel space metadata **1320** can be externalized with obfuscation. As used herein, the terms "obscure", and "obfuscate" refer to the process of ensuring that a process is not exposed to data that exceeds its access privileges via either anonymity or other modification. As used herein, the terms "scrubbed" refer to the process of removing data that exceeds a processes' access privileges. Data may be removed (or scrubbed) and/or anonymized (obscured), depending on its application requirements.

[0219] As an aside, a user process generally is limited to access only the data required for operation. For example, a user process may be restricted from accessing e.g., other user process information, kernel process information, historic information, and/or other user information. However, a kernel process may need to propagate a scrubbed or obfuscated version of privileged information to ensure proper user space operation. Consider a scenario where a two user space networking stacks operate in isolation from one another. When network congestion adversely impacts device performance, a kernel space flow-switch that supports both user space networking stacks may need to provide an obfuscated congestion metric. The obfuscated congestion metric enables each user space networking stack to make intelligent decisions to e.g., back-off data transmissions.

[0220] As previously noted, the MDU **1308** and MDK **1316** are often composed of different data types and/or structures. Consequently, the MDK **1316** may need translation to MDU **1308** formats. For example, the MDK buffets **1313** may be translated from pointers to offsets from a base address. In some cases, the MDK may additionally need to update formatting as well (e.g., length, checksums, etc.).

[0221] It is appreciated that the MDK **1316** is a data structure that is optimized for kernel space operation. Conceivably, malicious third parties could infer kernel space operation from MDK **1316** optimizations. For example, a malicious third party could infer kernel space activity based on changes to the MDU **1308** and the timing of concurrently monitored e.g., power consumption, and/or response times. This information is commonly used in cryptographic exploits. To these ends, MDU updates may be deferred in time or other manner to conceal how changes to the MDK affected device operation.

[0222] As previously noted, the kernel space is generally assumed to be trusted. However, artisans of ordinary skill in the related arts, given the contents of the present disclosure, will appreciate that more complex externalization techniques may be used where the user space does not trust the kernel space. For example, the user space process may additionally validate and/or sanitize kernel space changes to the MDU **1308** before incorporating such information within the user space metadata **1304** and/or user space data **1306**.

[0223] More generally, the concepts of internalization and externalization may be broadly applied to a range of different application architectures, the foregoing being purely illustrative. Other schemes with greater (or fewer) security

domains and/or greater (or fewer) processes associated therewith may benefit from the techniques described herein.

Secure and/or Anonymous Resource Reservation

[0224] As previously alluded to, many emerging use cases have been developed that rely on inter-process collaboration via socket-based connectivity. These emerging use cases often suffer from substantial domain crossing penalties and/or may introduce security holes that can be exploited by malicious parties. For example, a processor executing a first process may write data to a first socket, thereafter the processor context switches to a monolithic stack process to transfer the data to a second socket, and finally the processor context switches to a second process to read data from the second socket. More directly, existing architectures handle inter-process communication via a monolithic communication stack which is time shared to achieve inter-process communication.

[0225] As extensively discussed in previous sections, time-sharing monolithic communication stacks can be computationally expensive, insecure, and inefficient because every context switch requires domain crossing and data validation. However, the monolithic stack does simplify resource management because it vests resource management in a single entity. In other words, the monolithic stack is free to divide a global pool of resources in whichever manner it chooses.

[0226] When multiple user space stacks intend to form an inter-process communication pipe, each user space stack must allocate its own resources to support the pipe and bind the allocated resources to the other user space stack's corresponding resources. Unlike monolithic stacks, user space communication stacks are isolated from one another;

[0227] resources that are reserved by one user space process are not visible to other user space processes. Thus, various embodiments of the present disclosure, are directed to securely and/or anonymously reserving resources for communications.

[0228] In one such embodiment, an access control mechanism based on an identifying characteristic or attribute associated with a channel client can be used to control access to a reserved resource. In one such variant, the identifying characteristic or attribute may include e.g., process ID (PID), universally unique ID (UUID), and/or binary large object (BLOB) key. During operation, the nexus selects one or a combination of those identifying characteristics or attributes for securing access to a resource (e.g., a port of a named nexus instance) for the channel client.

[0229] As a brief aside, a process ID (PID) is an identifier used by most operating system kernels to uniquely identify an active process. The identifier may be used as a parameter in various function calls, allowing processes to be manipulated, such as adjusting the process's priority or killing it altogether.

[0230] A universally unique identifier (UUID) is another commonly used form of identification; UUIDs are a 128-bit number used to identify information in computer systems. The term globally unique identifier (GUID) is also used. The most commonly used UUIDs are generated according to a recognized standard (e.g., described in for example, RFC 4122, published July 2005 and entitled "A Universally Unique IDentifier (UUID) URN Namespace", incorporated herein by reference in its entirety.) UUIDs are unique (for practical purposes) when generated in accordance with the recognized standard. Unlike most other numbering schemes, UUID generation does not depend on e.g., a central registration authority or coordination between the parties generating them.

[0231] A binary large object (BLOB) key is any large binary data structure of sufficient size to provide unique identification. BLOB key generation is well known in the existing art, and not further described herein. Common such examples include without limitation, e.g., random number generators, linear feedback shift registers (LFSR), and/or other sources of random or pseudorandom noise.

[0232] FIG. **14** is an exemplary logical block diagram useful to illustrate the creation of a user space inter-process pipe ("upipe"). As shown therein, a first application space **1402**A includes a first application **1404**B with a first user space communications stack **1406**A, and a second application space **1402**B includes a second application **1404**B with a second user space communications stack **1406**B. Both the first and second applications are in isolated application spaces and are unable to share resources; however, the first and second applications can negotiate and reserve resources for one another. In some cases, the negotiations and reservations may be implied by virtue of their functionality. For example, applications that assume a client/server role, will accept the customary associated port number for e.g. the server application will open port 1 and the client application will open port 0.

[0233] The first user space communications stack **1406**A initiates an inter-process pipe by issuing an "open upipe" instruction. Responsively, the first user space communications stack **1406**A requests the allocation of resources and creation of a channel **1408**A. Similarly, a distinct second application **1404**B of a second application space **1402**B procures a second channel **1408**B from its user space communications stack **1406**B. In one exemplary embodiment, the requests for channel allocation and creation may be accompanied by an identifying characteristic or attribute (e.g., a PID) associated with the channel client. For example, the request to reserve resources for channel **1408**A is associated with the PID of user space application **1404**A; similarly, the request to reserve resources for channel **1408**B is associated with the PID of user space application **1404**B.

[0234] The exemplary nexus **1410** supports inter-process communication pipes (upipes) and can determine, based on the provided identifiers, whether or not resources should be reserved and to who the resources should be connected. Specifically, responsive to receiving the first request from the first channel **1408**A, the nexus **1410** checks that the channel **1408**A has the ability to open a first port. Similarly, responsive to receiving the second request from the second channel **1408**B, the nexus **1410** checks that the channel **1408**B has the ability to open a second port. Once the first and second ports are allocated, the nexus can connect the ports with an upipe instance. Notably, the foregoing process does not require the transfer of sensitive data (e.g., the PIDs); in other words, both channels have established a connection with privacy intact.

[0235] The foregoing examples of PIDs are purely illustrative of the characteristics and/or attributes that may be used. In the foregoing example the requesting PID is known at the time of request and can be used. Artisans of ordinary skill in the related arts, given the contents of the present disclosure will readily appreciate that a myriad of other identifier data structures may be substituted with equivalent success. For example, the aforementioned QUID may be

used to reserve resources between many different devices (PIDs are only unique in the context of a single device; UUIDs are considered globally unique). In other embodiments, the aforementioned BLOB key may be used where a PID is not yet known (for example, resources are reserved for a process that has not yet been created).

[0236] Additionally, while the foregoing example is presented in the context of an inter-process pipe and packet pool, virtually any shared resource may be bound. For example, the nexus may use port binding for e.g., network communication stacks, user-kernel pipes, device-to-device pipes, and/or any other port reservation scenario. Similarly, packet pools are just one example of a resource reservation; other common examples include without limitation: processing cycles, memory space, power consumption, time slots, frequency bands, spreading codes, bandwidth, etc.

Flood Detection and Mitigation

[0237] While much of the foregoing discussion has been focused on protecting the user space application from external attacks, artisans of ordinary skill in the related arts will appreciate that user space networking stack architectures may be compromised and used to initiate malicious attacks on others. Malicious behavior originating from within the user space may also greatly affect device performance.

[0238] One common example of malicious behavior is a so-called "flood" attack. A flood attack is a form of denial of service attack in which an attacker sends a succession of requests to a target server in an attempt to consume enough of the target server resources to make the target server unresponsive to legitimate traffic. Within the context of traditional BSD stacks, a compromised device could send floods of synchronization (SYN) messaging.

[0239] FIG. **15**A is a graphical representation of normal TCP/IP SYN operation **1500** in a BSD stack. TCP uses a three-way handshake to establish connections. Specifically, the three-way (or 3-step) handshake includes: a synchronization (SYN) message, a synchronization acknowledge (SYN-ACK), and an acknowledge (ACK). The SYN message is sent by a transmitter (client) to set a sequence number to a random value (A). In response, the receiver (server) replies with a SYN-ACK. The SYN ACK includes: (i) an acknowledgment number that is one more than the received sequence number (A+1), and (ii) a sequence number that is another random number (B). On receiving the SYN ACK, the client sends an ACK back to the server. The ACK includes: (i) an acknowledgement of the sequence number (A+1), and an incremented acknowledgement number (B+1).

[0240] FIG. **15**B is a graphical representation of a SYN flood operation **1550** in a BSD stack. As shown therein, rather than following the three-way (or 3-step) handshake protocol, a malicious transmitter (client) can send a flood of SYN requests. The receiver (server) will construe each SYN request as a valid request and attempt to generate SYN ACK responses. Unfortunately, the half-open connections created by the malicious client bind resources on the receiver (server) that eventually exceed the resources available. In other words, the server is stuck waiting for ACKs; the server cannot connect to any clients, legitimate or otherwise. This effectively denies service to legitimate clients.

[0241] As an important corollary, RST messages are another common message used in TCP/IP protocols. FIG. **16** provides a graphical representation of normal TCP/IP RST operation **1600** in a BSD stack. In an ongoing TCP connection, the RST message is sent by the transmitter when an unexpected packet arrives. A RST packet lacks a payload and only includes a RST bit set in the TCP packet descriptor information. There are a few circumstances in which a TCP packet might not be expected; the two most common are: (i) the packet arrives on a TCP connection that was previously established, but socket is already closed (e.g., the application was closed), or (ii) the packet is an initial SYN packet trying to establish a connection to a server port on which no process is listening. Since RST packets are generated within the BSD stack in response to an unexpected connection state, existing BSD stacks cannot create RST packet floods.

[0242] Various aspects of the present disclosure are directed to monitoring user space activity to stop malicious behavior. In one embodiment, an exemplary flow switch implements flow tracking logic which can detect and handle SYN floods. Additionally, since the user space network stack infrastructure exposes RST logic to the user space, exemplary variants also track RST messaging to prevent RST floods. If an attack is detected, the flow-switch will rate-limit the SYN and RST packets coming from the user space stack.

[0243] Referring now to FIGS. **17**A and **17**B, graphical representations of normal TCP/IP SYN flow-switch operation **1700** and SYN flood countermeasure operation **1750** are shown.

[0244] During normal flow-switch operation **1700**, the user space network stack generates SYN messaging which is provided to the flow-switch. The flow-switch has an internal state machine that monitors TCP/IP connections. Within the context of SYN operation, the internal state machine verifies that no SYN messaging is currently outstanding. So long as the current SYN message complies with expected TCP/IP state operation, the SYN message is transmitted to a receiver. As shown in FIG. **17**A, subsequent SYN-ACK and SYN messaging is similarly inspected for conformity.

[0245] Referring now to FIG. **17**B, consider the scenario where a user space networking stack has been compromised and attempts to generate a SYN flood. As shown therein, the flow-switch monitors each TCP/IP connection and determines that user space networking stack B (UStackB) appears to be sending SYN packets without waiting for the SYN-ACK replies. In other words, UStackB does not appear to comply with expected TCP/IP state behavior. Consequently, the flow-switch implements rate limiting countermeasures for propagating SYN messages of UStackB. Specifically, the UStackB must generate an ACK for SYN-ACKs. If no ACKs are generated, then no further SYNs will be transmitted. Rate limiting UStackB's behavior ensures that UStackA and the server are unaffected by UStackB.

[0246] As a brief aside, the illustrated embodiment implements rate limiting countermeasures as a less severe response than termination. While the user space networking stack is probably malicious, there may be legitimate reasons for state-less SYN transmissions. Conservatively rate limiting provides an effective but not necessarily disabling countermeasure. Notably, artisans of ordinary skill in the related arts may substitute other countermeasures with equivalent success. For example, strict implementations may terminate misbehaving user space networking stacks. Other implementations may de-prioritize misbehaving user space networking stacks.

[0247] As previously noted, existing BSD stacks cannot implement RST flood attacks; however handling network

protocols in user space (rather than in kernel space) introduces the possibility that a malicious party might attempt to create a RST flood. To these ends, FIGS. **18**A and **18**B provide graphical representations of normal TCP/IP RST flow-switch operation **1800** and prospective RST flood countermeasure operation **1850**.

[0248] During normal operation **1800**, the flow-switch can independently determine whether or not a TCP connection is alive by monitoring the TCP/IP connection state of the user space communication stack. As is illustrated in FIG. **18**A, legitimate RST messaging can be passed without issue.

[0249] In contrast, FIG. **18**B illustrates the scenario where a user space networking stack has been compromised and attempts to generate a RST flood. Similar to SYN flood countermeasures, RST behavior that does not comply with TCP/IP protocols can be detected and remedied. In the illustrated embodiment, RST flooding of UStackB is rate limited so that the server can continue to service legitimate RST usage. For example, in FIG. **18**B, a client that issues a SYN message to an inactive port of the user space network stack A (UStackA) will receive a RST message.

[0250] The foregoing discussion are presented in the context of SYN and RST flooding; however artisans of ordinary skill in the related arts will readily appreciate that connection state tracking can be used to prevent a variety of abuses and/or detect incorrect behavior. Additionally, the various principles described herein may be applied to a variety of stateful connection technologies, TCP/IP being purely illustrative.

Other Security Countermeasures

[0251] The foregoing discussions have addressed many possible security intrusions by malicious parties. However, artisans of ordinary skill in the related arts will readily appreciate that an innumerable range of malicious behaviors may be addressed with the user space networking stack architecture.

[0252] For example, buggy or hostile devices may use PCIe-mapped buffers to attack the host, such as by overwriting the content of in-use buffers, or performing timing/time-of-use based attacks. One exemplary embodiment of the user space network stack infrastructure may map segments to use the minimum possible memory access permissions on receive and transmit packet buffers so as to counteract such behavior.

[0253] In another such example, buggy or hostile devices may use PCIe-mapped buffers to attack the host. To help mitigate this vulnerability, embodiments of the present disclosure randomize the PCIe address space mappings, to make it difficult for an attacker to find vulnerable host-side resources. To help support this security protection, variants of the user space network stack infrastructure randomize its segment size by randomizing the number of pages per segment at the time segments are allocated. The user space network stack infrastructure may also randomize packet order within a segment, to make it more difficult to correlate packet address to position within a segment. This could be done via a random slide when the segment is first split into packets. For instance, by randomly choosing which slice of the segment is the first packet, instead of always using index 0. Together, these protections make it difficult for an attacker to predict the segment start, end and position from a packet's address. This also makes it difficult for an attacker to predict the location of other segments.

[0254] Furthermore, certain networking devices such as Wi-Fi chip and baseband could be compromised. Such compromised firmware could launch attack against kernel on the application processor using DMA memory. Time of Check to Time of Use (TOCTOU) attacks are caused by changes in a system between the checking of a condition (such as a security credential) and the use of the results of that check. For example, a TOCTOU attack could change DMA'ed memory after the kernel has done the sanity check.

[0255] In one exemplary embodiment, the nexus makes a kernel only copy before accessing device supplied data to help mitigate this vulnerability, all subsequent sanity checks and uses on the data are carried out on the kernel only copy. So even if compromised device launches TOCTOU attack, the kernel sees and uses the consistent kernel-only copy that is not affected as such.

[0256] Still other possible countermeasures may be implemented by artisans of ordinary skill, given the contents of the present disclosure.

Privacy and Anonymization of Sensitive Statistics

[0257] Network statistics could be misused in a variety of ways. Malicious entities may attempt to e.g., surreptitiously monitor activity, collect statistics regarding sensitive activity, and/or pollute statistics in an attempt to influence device behavior. For example, a malicious application could monitor and/or collect cellular data usage to provide such information to 3rd party spyware. In another example, a malicious application could pollute network retransmission statistics so as to cause the network to treat a network connection at a higher (or lower) priority than the connection should be treated.

[0258] Traditional BSD stacks use network statistics to assist in efficient operation. In some cases, these statistics are valuable not only for the BSD stack, but also to external entities. For example, statistics on packet loss, round trip times, and/or packet corruption may be useful for network routers to address e.g., network congestion and/or network outages. In the past, the BSD stack was a kernel space entity, thus statistics could be generated and/or managed for a variety of scenarios with limited oversight (e.g., the kernel space is assumed to be secure.) Additionally, the same BSD stack handled all socket connections; thus, the same BSD stack had a historic record of the entire device's network activity that was not application specific.

[0259] Unfortunately, the exemplary user space networking stacks are user space processes. User space processes do not provide the same guarantees of security as kernel space processes. Furthermore, data associated with user space processes typically terminate with the user space process. As a related corollary, network statistics collected by a user space networking stack are also highly correlated with user activity; i.e., the statistics are isolated from other network activity. In other words, statistics collected by user space networking stacks (that have not been provided to the kernel) could expose private and sensitive information to e.g., spyware. Ideally, user space networking stacks should still generate networking statistics so as to enable existing network optimizations without also exposing sensitive data.

[0260] To these ends, various embodiments of the present disclosure are focused on protecting and preserving private and sensitive information such as network usage statistics. In one embodiment, the system performs entitlements checks for privileged operations, so that only privileged

operations can be performed by processes possessing the appropriate entitlements e.g., trusted processes.

[0261] As used herein, the term "entitlements" refers to a set of access rights associated with a data processing entity within a data processing context. Common examples of access rights include without limitation: read, write, modify, share, copy, etc. In some cases, the access rights may include the rights to exclude other entities: e.g., prevent others from reading, prevent others from writing, prevent others from modifying, prevent others from sharing, etc. Data processing contexts may be inter-device, device-specific, intra-device, inter-network, network-specific, intra-network, etc.

[0262] Existing software design paradigms are generally assumed to have access to data that they have generated and/or would be responsible for. Usually, software processes only call libraries to fulfill ancillary functionality. In contrast, requiring explicit user space applications is a significant departure from existing software design paradigms. While enforcement of entitlements is performed by the kernel for the entire process (entitlements are on a per-process basis), only the user space application is developed by external 3rd parties; the user space networking stack is provided via 1st party user space library. In other words, the entitlement is for the entire process, which includes all threads running in such process. The user space networking stack is in a library that comes with the operating system, that is linked against the application (assuming the application entitlements). In effect, this enables a multitude of differently entitled user space networking stacks (rather than a single monolithic stack without entitlements).

[0263] Various embodiments of the present disclosure require the user space application to actively request (and be granted) entitlements to access user space data from the 1st party user space library. The active step can be policed by the 1st party and/or security countermeasures for abuse. Explicit entitlements based paradigm ensures that only authorized processes are allowed access to network statistics.

[0264] While the foregoing discussion is presented in the context of a 1st party device manufacturer that provides the operating system (OS) and kernel software, artisans of ordinary skill in the related arts will readily appreciate that any sufficiently trusted party may be substituted with equivalent success. For example, 2nd party vendors may purchase a device and install their own OS thereon for resale to a consumer. Under such situations, the 2nd party could implement a similar scheme: e.g., enforcement of entitlements via their 2nd party user space libraries. In fact, a sufficiently trusted 3rd party entity could provide a similarly enabled OS to customers for installation on commodity devices. More generally, artisans of ordinary skill in the related arts will readily appreciate that the concepts described herein may be applied to any provided user space library that exerts self-contained control over user space application functionality.

[0265] As used herein, the term "self-contained" refers to a software process that may be called by another software process, but where the internal data structures and/or functions are isolated therefrom. In other words, a self-contained software process can be treated as a functional "black box" (it generates output data structures based on its inputs, but does not expose its inner workings) to the calling software process. In one exemplary embodiment, a self-contained process enforces its black box operation using entitlements that can only be explicitly granted by a privileged entity to the self-contained processes' calling process.

[0266] As used herein, the term "library" refers to a suite of software processes provided by the operating system (OS) (or similar kernel space process) that have a well-defined interface (e.g., an application programming interface (API)). The API enables the calling software process to invoke one or more library functions. For instance, application developers that write user space applications can use a library to make system calls instead of implementing those system calls themselves.

[0267] Self-contained operation enables user space libraries to record, track, and/or modify a variety of different statistics without exposing statistics to unentitled user space applications. However, a self-contained user space process does not persist beyond the termination of its calling user space process. Consequently, various aspects of the present disclosure are directed to propagating user space statistics into the kernel space in a manner that limits exposure to malicious behavior.

[0268] In one exemplary embodiment, a user space library may provide user space statistics updates to a kernel space nexus (e.g., flow-switch). In one such implementation, the kernel space nexus independently tracks statistics based on updates from its associated user space networking stacks. In order to update the kernel space statistics, the flow-switch may perform sanity checks on the user space generated statistics to determine whether or not the user space generated statistics should be used. In one specific implementation, the nexus instantiates a "shadow" kernel-only statistics object in addition to the user space protocol stack instance shared statistics object. The kernel-only statistics object stores historical values of the user space protocol stack statistics. Before accepting the user space protocol stack statistics, the nexus compares each user space statistics snapshot with the historical value and performs anomaly detection (based on a delta or change value).

[0269] In some embodiments, the kernel space nexus independently tracks statistics without using or trusting user space statistics. In particular, certain statistics are so critical that only the kernel-only statistics object is used. For example, cellular data usage may be used to bill services and would be a likely target for abuse; e.g., a user space statistic might misreport data usage to get around cellular usage accounting. For such statistics, the kernel-only statistics object can be used without user space input.

[0270] As a related note, using statistics that are generated from the user space libraries enhances performance because the statistics can be measured without context switching. For example, user space TCP/IP stack operation may monitor TCP/IP statistics (packet error rates, etc.) without switching to kernel space. However, a variety of different network statistics can be tracked within the kernel space with minimal performance cost; for example, any transactions performed by the nexus (flow-switch) can be monitored without context switch penalties. Thus, for example, certain kernel-only statistics objects can be gathered by monitoring connection state. In one such variant, the nexus monitors connection state based on packet information.

[0271] As but one example, consider round trip delay time (RTT) measurements which can be determined from TCP/IP timestamp information. RTT measurements are critical for TCP operations such as retransmission and fast recovery; during normal operation, RTT measurements can be used to

speed-up or slow-down packet retransmissions to compensate for network congestion, change service latency/throughput, etc. However, a malicious actor could attempt to provide false RTT measurements to attack the device or other network entities. For example, setting extremely small RTTs would cause unnecessary retransmissions to a network connection. Setting extremely large RTTs might cause retransmission logic to register excessive packet loss. Coordinating many devices with false RTT might create unexpected attack vectors; for example, an army of devices that have a malicious RTT values could launch a DoS (denial of service) attach against a network host.

[0272] In one embodiment, the nexus (in kernel space) independently calculates RTT measurements by monitoring the connection state in a flow tracker of the flow-switch. To accept measurements from user space, the kernel does a sanity check with its estimated upper and lower bounds. Only the RTT samples that pass the kernel sanity check may be used (and/or published to other TCP stack instances).

[0273] While the foregoing discussions have been presented in the context of a kernel space process receiving statistical data from a user space process, artisans of ordinary skill in the related arts will appreciate that the kernel space process may also provide statistics from a first user space networking stack to other user space processes. As previously alluded to, a kernel process may propagate a scrubbed or obfuscated version of privileged information. For example, network congestion adversely impacts the entire device system, a kernel space flow-switch that collects network congestion statistics from one user space networking stack may distribute the collected statistics to other user space networking stacks to assist in overall system performance.

[0274] In some cases, the user space statistics can be provided directly; in other variants, the kernel space may propagate the kernel-only statistics which have been updated with the user space statistics. Still other variants may enable the kernel space to e.g., provide some modified statistics. For example, the kernel space may provide an anonymized version of the statistics (e.g., the recipient user space does not know whether the source of the statistics is another user space process or the kernel itself). In some such cases, the kernel may intelligently change the statistics in view of the recipient application's activity; for example, the kernel may provide more accurate statistics to higher priority applications, whereas lower priority functions and/or background processes may only receive default statistics (or possibly even statistics that imply poor network conditions, even though the actual network conditions are acceptable).

[0275] More generally, artisans of ordinary skill in the related arts will appreciate that the foregoing techniques described herein enable a kernel space entity (e.g., nexus, flow-switch, etc.) to manage, collect, check, manipulate, and/or leverage statistics that are authorized for monitoring by user space networking stacks.

Trusted TFO & ECN

[0276] TCP in the user space network protocol stack supports both TCP Fast Open (TFO) and Explicit Congestion Notification (ECN). Both TCP options are enabled and/or disabled based on per network heuristics maintained on the system. This is done to avoid using TFO and ECN on networks that either do not support these options or blacklist devices if the options are present in the TCP header.

[0277] During normal operation, the ECN and TFO heuristics are updated each time a TCP connection experiences a success or failure when using TFO or ECN. If a TCP connection does not experience issues when using these options, new TCP flows would continue to use these TCP options. So, if the heuristics is updated with incorrect data, it could lead TFO and ECN being enabled on networks that do not support these options.

[0278] Within the context of a user space network stack infrastructure, the TCP protocol stack runs in the user process's context. Each time a user space TCP connection experiences success or failure while using TFO or ECN, it makes a system call into the kernel to update the heuristics. So, a malicious app could indicate a TFO or ECN success on networks that do not support TFO or ECN by simply making a system call. This would result in new flows on the system incorrectly using TFO and ECN option which could lead to bad user experience or in worst case scenarios, blacklisting of devices.

[0279] All processes can indicate to the system heuristics a failure of TFO or ECN. But, in one exemplary embodiment, only processes that are trusted on the system can update the heuristics with TFO or ECN success. This prevents malicious apps from incorrectly updating TFO or ECN success on networks that do not support these options.

Flow Enhancements—

[0280] Unlike traditional monolithic communication stack implementations, systems described herein may manage many concurrent e.g., user space communication stacks and/or drivers. Each of the user space applications and/or drivers may be associated with one or more data flows. Consequently, various aspects of the present disclosure are directed toward managing different data flows

Flow Classification—

[0281] In traditional communications stack implementations (e.g., a legacy BSD communications stack), user space data is written to the socket in a "bulk data" format (without packetization, TCP/IP headers, or other communication protocol information). Subsequent to being written to the socket, the user space data is read by a kernel process and packaged into packets for transfer via e.g., the BSD communication stack. These traditional communications stack implementations were generally considered "trusted" as the user space would not have access to, inter alia, the generation of network addresses used for the actual packet transfer. In other words, packet addressing was handled by the kernel space which is considered more secure than user space.

[0282] In order to improve operation, a traditional BSD communication stack may create metadata in the kernel that enabled quick access to e.g., frequently used and/or important data (e.g., TCP/IP headers, etc.). Notably, in a traditional BSD communications stack, kernel space metadata can be created from user space data (and/or user space metadata) in mbufs. The kernel does not distinguish (or treat differently) user space data and user space metadata, as both are transferred via generic mbuf data structures. For example, TCP/IP metadata (e.g., source and/or destination tuples (address, port), etc.) may be implicitly or explicitly set by the user space application via socket APIs. The TCP/IP metadata stays in the socket metadata, and can be directly parsed by

the kernel from mbufs. The TCP/IP metadata resides in kernel-only memory and is only modifiable via socket APIs.

[0283] Within the context of the present disclosure, user space communication stacks can generate packets and metadata within the user space. However, the packets and metadata generated from the user space cannot be trusted by the kernel (see e.g., co-owned and co-pending U.S. patent application Ser. No. 16/146,324 filed Sep. 28, 2018 and entitled "METHODS AND APPARATUS FOR PREVENTING PACKET SPOOFING WITH USER SPACE COMMUNICATION STACKS", the contents of which were incorporated supra). As described therein, software applications in the user space are restricted for security and stability reasons.

[0284] More directly, the metadata generated within user space may be useful for kernel operation, however using the user space metadata directly in the kernel may introduce potential security and stability issues. For example, if the kernel were to use metadata generated within the user space, it may open itself to illegal memory accesses, which can result in various memory problems (e.g., see Existing Network Socket Technologies for discussion of other potential issues supra).

[0285] Additionally, BSD packets and metadata are hardware agnostic. The BSD communications stack implementation was designed to accommodate any hardware platform; however, the BSD packets and metadata are likely not optimized for the particular hardware platform it is executed from. In fact, traditional communications stack implementations (e.g., a BSD stack) may even be designed for different design assumptions e.g., where memory is expensive and processors word sizes are smaller (e.g., 32-bit architecture). Such characteristics and associated design may be suboptimal and inefficient in modern systems. For instance, metadata generated by a user space communications stacks may not align to the "natural word boundaries" of a modern processor cache (e.g., 64-bit architecture).

[0286] Referring now to FIG. **19**, one exemplary implementation for a system **1900** that addresses the foregoing deficiencies is shown and described in detail. In one exemplary embodiment, a flow classifier **1914** sanitizes user space metadata and generates kernel space classifications and/or metadata therefrom to optimize accesses to e.g., frequently used and/or important data (e.g., TCP/IP headers, etc.).

[0287] The exemplary system **1900** may be implemented through the use of a non-transitory computer-readable medium (e.g., a computer-readable apparatus) which may be embodied as software, hardware, or combinations of the foregoing. The non-transitory computer-readable medium may include one or more computer programs with computer-executable instructions, that when executed by, for example, one or more processing apparatus may implement one or more of the methodologies described subsequently herein. Moreover, while a specific architecture is shown in FIG. **19**, the illustrated topology shown in, for example, FIG. **19** may be readily modified to include one or more applications **1902**, one or more channels **1904**, one or more pool of resources **1906** associated with a respective application, one or more flow-switches **1908**, one or more pool of resources **1910** managed by, for example, one or more drivers **1912**, and one or more flow classifiers **1914**. These and other variants would be readily understood by one or ordinary skill given the contents of the present disclosure with the illustration contained within FIG. **19** merely being exemplary.

[0288] FIG. **19** illustrates three applications **1902**A, **1902**B, and **1902**C that reside within user space. One or more of these applications **1902**A, **1902**B, and **1902**C may include its own communications stack as is described in additional detail supra. Each of these applications **1902**A, **1902**B, and **1902**C may further communicate with the kernel space through respective channels **1904**A, **1904**B, and **1904**C which are coupled with a respective pool of dedicated resources **1906**A, **1906**B, and **1906**C. Some (or all) of the data resident within these pools of dedicated resources **1906**A, **1906**B, and **1906**C may be communicated to managed pools of resources **1910**A, **1910**B via a flow-switch apparatus **1908**.

[0289] As shown in FIG. **19**, each single entity **1912** (e.g., driver) managed pool of resources **1910** is separate and distinct from the pool of resources **1906** associated with respective applications **1902**. The single entity **1912** may control access to the managed pool of resources **1910**; for example, the single entity **1912**B determines an allocation of pool resources **1910**B for transferring the data stored in any one or more of pool resources **1906**A, **1906**B, **1906**C. Similarly, each channel **1904** may control access to its managed pool of resources **1910**; for example, the channel **1904** reads and writes to its corresponding allocation of pool resources **1906** for receipt/delivery. The management and operation of these managed pools of resources **1906**, **1010** is described in co-owned and co-pending U.S. patent application Ser. No. 16/144,992 filed Sep. 27, 2018 and entitled "METHODS AND APPARATUS FOR SINGLE ENTITY BUFFER POOL MANAGEMENT", incorporated supra.

[0290] Consider an exemplary usage scenario where a communications stack application **1902**C within user space opens a channel **1904**C in order to transact data. In the illustrated embodiment, the communications stack **1904**C writes a user generated packet from a corresponding application **1902** into one of the pool of dedicated resources **1906**C associated with the respective application **1904**C (transaction **1952**).

[0291] Initially, when the user space application generates a packet, the kernel allocates and creates: a metadata kernel object (MDK), and a metadata user object (MDU). Initially, the MDU (written by the user space) may be copied into an MDK data structure. More directly, the user space only has access to the MDU, whereas the kernel space has access to both the MDU and MDK. The MDU and MDK are "parallel" but distinct objects; thus, only certain fields within the objects are transferred during a write from user space to kernel space (internalizing data from the MDU to the MDK) and reads from kernel space to user space (externalizing data from the MDK to the MDU). However, the MDK must be sanitized before it can be used in the kernel space at transaction **1954**.

[0292] As used herein, the term "sanitize", "sanitization", and/or "sanitizing" refers to a process of ensuring that data conforms to the privileges and/or requirements of the kernel and/or the user space. For example, when internalizing data from the MDU to the MDK, the data may be checked for appropriate formatting, validity, and/or malicious content (e.g., to avoid security and stability issues as discussed supra). Similarly, when externalizing data from the MDK to

the MDU, the data may be checked to ensure that kernel private flags, values, and/or other sensitive information are not unintentionally exposed.

[0293] If the MDK is successfully sanitized, then the flow classifier **1914** generates an optimized MDK from the sanitized MDK (transaction **1956**). For example, the MDK may be re-written to "naturally align" with the "natural word boundaries" of the processor cache.

[0294] As a brief aside, processor architectures perform memory accesses most efficiently when the data address is a multiple of the data size; this property is referred to as "natural alignment". For example, data structures are "naturally aligned" for a 32-bit processor when the data structure's data addresses are multiples of 32-bit words. Similarly, data structures that are addressable in multiples of 64-bit words are naturally aligned for 64-bit processors. By extension, a 64-bit processor that accesses data structures with data addresses that are multiples of 32-bit words are not naturally aligned. As used herein, the term "natural word boundary" refers to memory locations in computer memory that are "naturally aligned".

[0295] In some cases, the flow classifier **1914** may also generate other kernel-specific metadata. Notably, the kernel space metadata (MDK) and the user space metadata (MDU) are "parallel" but distinct objects in the exemplary embodiment. Specifically, the MDU can be used to share information with the kernel, but the kernel space MDK is never shared with the user space. The user space can continue to modify and/or update the MDU and the kernel space can independently modify the MDK without propagating changes to the MDU. As a result, the MDK can be modified in view of other system considerations without leaking privileged information back to the user space application.

[0296] In one exemplary embodiment, the kernel may refer to the MDU during internalization phases; thereafter (assuming the MDU passes all validations) only the MDK of the packet is used by kernel. More directly, once a packet is internalized, the kernel only uses the MDK until the packet is externalized.

[0297] In one exemplary embodiment, the kernel may refer to the MDU during externalization phases, when packet data/ownership is provided to the user process. During externalization, the MDU is initialized to reflect the state of the packet according to its MDK.

[0298] While the foregoing process is described in the context of an internalization/externalization scheme, other implementations may use more or less granular permissions and/or controls. For example, in other embodiments certain packet data modifications may be sanitized and/or propagated into the MDK, thereby allowing the kernel space to dynamically benefit from the user space application MDU updates and/or vice versa.

[0299] Additionally, the flow classifier may re-organize the MDK contents based on various other considerations (e.g., any hardware and/or software considerations). In some cases, certain types of metadata may be higher priority and/or more frequently accessed than other information. For example, the TCP/IP address header may be a particularly valuable and frequently accessed information. The MDK metadata can be re-arranged so that the TCP/IP header can be easily accessed. Subsequent accesses to the kernel space MDK may be made, by e.g., the flow-switch **1908** without accessing either the MDU or the packet data itself. In other words, by providing an easily accessible and secure location for kernel specific metadata, overall kernel operation can be greatly improved.

[0300] Referring now to FIG. **20**, the aforementioned exemplary data structures are depicted in greater detail. As shown, the packet data **2002** that is generated from user space is composed of user space metadata **2004** and user space data **2006**. The user space data packet **2002** is copied into: an MDU object **2008** and buffers **2014** in kernel space.

[0301] The exemplary MDU object **2008** includes user space metadata **2010** and one or more buflets **2012**. Each buflet may contain a pointer or index to portions of user space data that are stored in buffers **2014**. The flow classifier sanitizes the MDU object (and/or user space data) and generates an MDK object **2016** therefrom. The MDK object **2016** includes sanitized user space metadata **2018**, and may include additional kernel space classifications and/or metadata **2020** and/or the buflets **2012**. In this way, the MDK metadata is related but distinct from the MDU metadata and the backing data buffers are not duplicated. The kernel space classifications and/or metadata **2020** may be generated within the kernel space for the internal use of flow classifier as discussed supra.

[0302] In some embodiments, the MDU and/or MDK may use different types of reference data structures. For example, in some variants the MDU uses indices (an offset from a baseline address) to point to its corresponding buffer object (s). Index based addressing within the allocated user space region may be more efficient and secure, since the user space application should only have access to objects within its allocated region. In contrast, the MDK may use pointers which can be efficiently used in kernel space (and which does not have the same access restrictions).

[0303] As previously noted, classifications and/or metadata can be re-organized, naturally aligned, and/or further modified. FIG. **21** depicts a side-by-side logical representation of (i) metadata **2102** as generated within the user space, and (ii) metadata **2104** re-organized and/or optimized for the kernel. In this example, the traditional BSD data structure is designed to accommodate any device including e.g., memory constrained devices where memory is expensive and processors are smaller (e.g., 32-bit words). As a result, the BSD data structure saves memory by "packing" data close together; i.e., the user space metadata **2102** is packed for storage efficiency into mbufs with 32-bit word boundaries. The packed data structure **2102** is likely not optimal for most modern devices, and especially for 64-bit processor architectures.

[0304] In contrast, the kernel space metadata **2104** depicts how the same metadata (and/or sanitized or otherwise modified versions thereof) may be re-aligned to fit within e.g., 64-bit "natural boundaries" and/or re-organized according to frequency of use or importance, as discussed supra. As shown therein, the data is re-aligned so as to suit a 64-bit processor architecture. For example, metadata A is read from a 32-bit packing and placed at the naturally aligned 64-bit address 0; the remaining bits are padded with null data (e.g., zeros). Metadata C is unpacked from multiple 32-bit words, re-organized, and naturally aligned to the 64-bit address 64. Metadata C' is padded so as to ensure that metadata B is naturally aligned to the 64-bit address 192.

[0305] As used herein, the term "packing" and/or "packed" refers to techniques and/or methods to reduce the size of a data structure. A packed data structure generally has

very little (if any) null information, and in some cases may even be compressed, decimated, or otherwise lossy/loss-less encoded. In contrast, the term "padding" and/or "padded" refers to techniques and/or methods to add bits (e.g., null bits or redundant bits) to a data structure so as to e.g., achieve a certain size or other desirable property (e.g., error correction, parity, etc.)

[0306] FIG. **22** is a logical block diagram that illustrates the problem of "false sharing." As shown therein, a processor **2202** has a 128-byte local cache **2204** and is coupled to a 64-bit DRAM **2206**. However, the kernel space may be unaware of the local cache **2204** operation and/or parameters. As a result, the kernel may incorrectly allocate memory for objects. For example, as shown therein, a cache entry has two objects of metadata **2208**A **2208**B that are packed within the same 128 byte cache entry. In the illustrated example, a first process writes a first object **2208**A and a second process reads a second object **2208**B. However, since the write to the first object **2208**A "dirties" the cache, the cache entry must be re-written before the second object **2208**B can be read. Notably, the second object **2208**B was not affected by the first object **2208**A; in other words, the concurrent cache accesses are "falsely sharing" the same cache entry.

[0307] Various embodiments of the present disclosure optimize object allocations to improve cache operation. FIG. **23** is a logical block diagram that illustrates cache operation based on cache sized object allocations. As shown therein, a processor **2302** has a 128-byte local cache **2304** and is coupled to a 64-bit DRAM **2306**. The kernel determines the cache size and allocates object in commensurate multiples of cache sizes (e.g., 128 byte cache entries). Thus, the metadata object **2308**A has its own cache entry and the metadata object **2308**B has its own cache entry. In some cases, the metadata objects **2308** may additionally be padded and stored at natural word boundaries of 64-bit words by e.g., the aforementioned exemplary flow classifier. Thereafter, the processor **2302** can efficiently retrieve metadata of interest in naturally aligned memory accesses while maintaining cache coherency. In the illustrated embodiment, the 64-bit processor can write to the metadata object **2308**A without affecting metadata object **2308** and vice versa.

[0308] Notably, the packed metadata storage of FIG. **22** is optimized for compactness of storage, whereas FIG. **23** is optimized for processor access and local cache operation. As illustrated above, natural alignment of metadata can reduce memory accesses, the local cache footprint, and can free memory resources for other purposes within the processor. In other words, the intelligent partitioning and re-ordering performed by the flow classifier as described supra, can optimize performance based on the hardware considerations of the specific platform (e.g., word sizes, cache operation, etc.)

Flow Manager

[0309] Referring now to FIG. **24**, a logical block diagram of one exemplary nexus supporting multiple user space applications under both user space networking stacks as well as legacy networking stacks is presented. The nexus controls and manages the pool resources for a driver to effect data transfers.

[0310] As shown therein, a user space application **2402** may include both the application itself, as well as a user space networking stack. The user space applications **2402** read and write data packets via a channel **2404** into a user packet pool **2406**. As previously noted, each user space application **2402** is associated with its own packet pool **2406**. However, each user space application **2402** may have multiple data flows which share a common user packet pool. In contrast, the traditional networking stack operation uses a single kernel space protocol stack instance **2418** that manages a global pool of data buffers (mbufs) **1416**; applications in user space **2412** may open one or more sockets to read and write to the global pool of data buffers.

[0311] As previously alluded to, the exemplary user space networking stack architecture exposes control over individual data flows in a manner that was heretofore unavailable to kernel space networking stack architectures. For example, the user space networking stack can create new flows, destroy unused flows, and/or modify flows that are already in existence. Such control was not previously possible e.g., all flows were generically treated the same. The finer control enables a user space networking stack to focus its resources on flows that matter and/or divert resources away from unnecessary and/or lower priority flows.

[0312] Within this context, the nexus also needs to maintain some control over flow lifecycle. Specifically, as user stack instances are created and destroyed, there is a corresponding need to manage flow life-cycles within the nexus. For example, when a user stack is first created, the nexus may allocate a minimum number of memory resources to support any flows associated therewith. Similarly, when a user space networking stack ends, its corresponding flows should also end.

[0313] In one exemplary embodiment, the nexus includes a flow manager that couples to multiple communication stacks in order to manage flow life-cycles. In one such embodiment, the flow manager is a logical entity that accepts calls to create, destroy, defunct, and/or otherwise manipulate flows. In some variants, the flow manager may also automatically shut down flows when the flow owner process exits.

[0314] FIGS. **25**A-**25**D are software ladder diagrams of various flow management tasks made possible by the exemplary user space networking stack architecture.

[0315] Referring first to FIG. **25**A, a user space application **2502** can request the creation of a flow from the nexus **2508**. Responsively, the flow manager allocates a flow identifier (flowID) and allocates the corresponding memory allocation associated therewith. The flowID is provided back to the user space application **2502**.

[0316] Subsequent transactions can be made using the flowID. For example, a user space application **2502** may write data to the flowID; the data can be routed from the flow memory to the HW driver **2520**. For transfer.

[0317] Once the user space application no longer needs the flow, it can close the flow. The flow manager **2508** responsively deallocates the flowID, and corresponding memory allocations.

[0318] As a brief aside, the flowID uniquely identifies the memory allocations that are associated with a flow; the flowID persists for the duration of the flow lifetime. Existing monolithic stacks manage a global pool of memory buffers in a non-persistent manner; once written, the memory buffers are no longer associated to their originating sockets. In other words, the existing global pool of memory buffers is a "fire and forget" type of memory allocation. In contrast, the flowID can be used to cleanly and persistently address

only the memory allocation that backs a data flow. As a result, an application may for example, add to, remove, and/or modify its flow allocation. In other words, the flowID provides the user space application **2502**A (and other processes) a mechanism to refer to its flow in a manner that extends for the duration of the flow's use.

[0319] Referring now to FIG. **25**B, one such exemplary "reclamation" scenario is illustrated. The nexus **2508** is a kernel space process which often must balance many competing device interests. In some situations, the nexus may need to reclaim memory that is allocated to a flow.

[0320] As before the user space application **2502** requests the creation of a flow from the nexus **2508**; the flow is created and the flowID and memory allocation are reserved. However, at a later point, the nexus reclaims the memory allocated to the flowID (e.g., when the process goes defunct or the flow terminates/closes, etc.) The flowID is deallocated, and any subsequent references thereto will re-direct to a zero filled page.

[0321] Existing network stack architectures could not reclaim memory in this manner. For example, a generic pool of buffers could not be sorted based on their source or destination socket; e.g., in a BSD stack, once mbuf is written to kernel space it cannot be pulled back (the socket does not have a persistent association with the written data). In fact, the entire BSD stack global mbuf pool was "wired" under the assumption that mbufs could not be reclaimed once written (i.e., the total global mbuf pool had a static physical memory location).

[0322] Referring back to FIG. **25**B, the user space application **2502** will construe the zeroed data to be a connection loss during a subsequent access to the flowID. Responsively, the user space application **2502** can re-establish a flow (by requesting a new flow, etc.).

[0323] In a related example, the nexus **2508** may also perform clean-up tasks in the event of a user space application crash. For example, as shown in FIG. **25**C, a user space application **2502**A requests the creation of a flow; memory is allocated and a flowID is generated. Shortly thereafter, the user space application **2502**A crashes. The crash event can be provided to the nexus, which responsively closes the flow and deallocates its memory. Thereafter, user space application **2508**B can be restarted.

[0324] Existing network stack architectures could not reclaim memory from application crashes. As previously noted, a generic pool of buffers could not be sorted based on their source or destination socket. Consequently, even if an application were to crash and its socket was closed, whatever mbufs that were outstanding would remain. This could result in long term memory leakage.

[0325] As previously noted, the nexus may interact with both non-kernel and in-kernel networking stacks to e.g., enable legacy stack operation and/or other daemon based networking stacks. For example, the flow manager may interoperate with both legacy sockets as well as user space network extensions. Moreover, it is further appreciated that applications that reference higher layer APIs may not know whether it is best served by a socket, network extension or other protocol stack instance. Selection of the appropriate protocol stack be managed by the libnetcore (e.g., either a socket to a legacy stack or a network extension to a user space networking stack, etc.)

[0326] In one exemplary embodiment, the flow manager works with the Network Extension Control Policy (NECP) module that interfaces with the user space network stack (which resides within a larger library for network interfaces (libnetcore)). FIG. **25**D illustrates one such example. As shown therein, a higher API call to libnetcore, will communicate with the NECP, which communicates with the flow manager to obtain a flow for e.g., a socket or user space networking stack instance.

[0327] Notably, the user space application **2502**A is unaware of its underlying stack operation. Even though the user space application **2502**A may not leverage all of the attendant benefits of flow management, the overall system may still benefit. In particular, the nexus can still reclaim memory associated with a flow during e.g., a memory pressure event. Thus, even if the user space application **2502** is oblivious to its underlying transport mechanism, the benefits to user space networking may still be leveraged by other user and/or kernel space entities.

[0328] In summary, the user space networking stacks of the present disclosure expose fine granularity control over flows for both user space and kernel space. As a result, various embodiments of the present disclosure are directed to managing flow lifecycles e.g. flow creation and destruction at the flow-switch, which interfaces with calls/events from other components. In one exemplary embodiment, the flow manager is the entity that provides the interface. The flow manager accepts calls to create/destroy/defunct flows. It also shuts down flows when the flow owner process exits. This allows proper clean-ups to be done regardless of how the process terminates.

Flow Switch Actions, Routing, and Packet Processing

[0329] Most network technologies are optimized for specific uses and may have different priorities and/or capabilities. For example, Wi-Fi may be able to support (best effort, high latency, high throughput), whereas Bluetooth may support human interface device operation (isochronous, low latency, low throughput). Some emerging use cases may attempt to intermingle functionality among a number of different technologies. However, intermixing these technologies may require that each interface operates at the lowest common capability.

[0330] As a further complication, traditional networking infrastructures are built on the assumption that the device is connecting to external devices via one network. Existing networking infrastructures are poorly suited to intermixing different network technologies (e.g., an 802.11 network and a Bluetooth network). While solutions do exist for inter-device connectivity, these solutions are inefficient and require convoluted and excessive context switches.

[0331] As described supra (see e.g., Flow Classification), the exemplary flow classifier of the present disclosure generates kernel-specific metadata. Unlike traditional BSD stacks which uses an undifferentiated pool of buffers (mbufs), the exemplary packet pools are organized to preserve data flow organization. In other words, the exemplary system described herein provides both data structures and mechanisms to assist in packet processing. Various embodiments of the present disclosure can leverage the kernel-specific metadata to enable efficient flow-specific processing.

[0332] In one exemplary embodiment, each packet of a data flow has e.g., a kernel-specific metadata (MDK) component which can include data useful for the kernel-only processing of the packet. In one exemplary variant, the

MDK can include data useful for the kernel-only processing of the packet. Common examples of such data may include e.g., priority, flow route, nexus, source, destination, special handling, and/or any other information useful thereto.

[0333] In one exemplary embodiment, packet forwarding based on the entries of a flow table allows the system to facilitate optimal forwarding data plane logic; in one such variant, multiple network interface nexus may be fused together to form a direct conduit for sending packets to one another.

[0334] FIG. **26** is a logical block diagram of a logical conduit for fusing nexuses. As shown therein, a first user space application **2602**A communicates with a channel **2604**A to transact data via a pool **2606**A, using a first nexus **2608**A over a first driver **2614**A. Similarly, a second user space application **2602**B communicates with a channel **2604**B to transact data via a pool **2606**B, using a second nexus **2608**B over a second driver **2614**B. Additionally, the first nexus **2608**A and second nexus **2608**B may share a common pool of resources to facilitate transmissions therebetween.

[0335] For example, consider the use scenario where a first driver **2614**A (e.g., IEEE 802.11 network) receives a first flow of data packets; these data packets can be routed via the first nexus **2608**A to the first user space application **2602**A. The first user space application **2602**A may decide to transfer the data packets to a remote interface for playback (via e.g., a Bluetooth driver **2614**B). Responsively, the first nexus **2608**A opens a first conduit to a second nexus **2608**B via the conduit pool **2616**. Packets associated with the flow are switched through the conduit into the second user space application **2602**B, which renders the data for playback via the second driver **2614**B.

[0336] In one exemplary embodiment, the flow routing can be implemented via a flow routing table **2610**. The flow routing table **2610** enumerates, for each different flow, how the flow should be labeled in order to be efficiently routed by one or more nexuses. In one such implementation, the labels are identified by links (paths) between nodes rather than endpoints. As but one example, the kernel may create a first flow route **1** between a first driver **2614**A and a user space application **2602**A, a second flow route **2** between the user space application **2602**A and the second user space application **2602**B, and a third flow route between the second user space application **2602**B and the second driver **2614**B.

[0337] In the exemplary variant, the labels may be stored within kernel-space metadata (MDK) information associated with each packet. For example, during operation, data packets received by the first driver **2614**A are updated to include a "flow route **1**" label within the MDK; the first nexus **2608**A can switch the data packets to the appropriate flow, based on an inspection of the MDK data. Routing packets according to the MDK can be performed very efficiently, as the packet itself need not be inspected. In other words, the flow switch can direct data packets from one node to the next node based on short path labels in the MDK, rather than long network addresses, sockets tuples, etc. Short labels in the MDK avoid lookups in a routing table. In fact as a practical matter, "routing" packets within the context of FIG. **26** can occur almost entirely as direct memory access (DMA) transfers among the various device memory allocations (rather than e.g., packing and unpacking packets and context switches back and forth across user/kernel domains).

[0338] The foregoing forwarding scheme is purely illustrative; artisans of ordinary skill in the related arts will readily appreciate that the various techniques described herein may be broadly applied to a variety of different applications. For example, packets may be forwarded to e.g., a flow-switch port, a user space protocol stack, a kernel process, a user pipe, a kernel pipe, a legacy BSD stack, etc. Additionally, the techniques described herein may be broadly applied to a variety of other packet manipulations. For example, packets may be dropped, transformed, reformatted, and/or otherwise processed. For example, certain low priority flows may only be useful for limited purposes (e.g., stock ticker quotes, etc.); these flows need not be terminated, however packets for these flows may be preferentially dropped where the system is under stress. Still other flows may have timing requirements and/or fairness rules. More generally, artisans of ordinary skill in the related arts will readily appreciate that the techniques described herein allow for the efficient application of traffic rules without separate database lookups.

[0339] Some technologies may need/benefit from in-line packet modification; for example, certain technologies may benefit from in-line big-endian/little-endian conversion, 32-bit/64-bit word conversions. Other common "glue logic" transformations may be substituted with equivalent success, by artisans of ordinary skill in the related arts, given the contents of the present disclosure.

[0340] As a brief aside, existing BSD stack architectures were able to leverage information about data flows for a variety of system wide optimizations. For example, generalized data flow tracking enabled BSD stacks to intelligently e.g. schedule tasks, manage resources, and/or otherwise optimize system operation. In the exemplary user space network stack infrastructure, the TCP/IP protocol is in user space and kernel space processes do not have visibility into user space TCP/IP operation. Consequently, existing generalized data flow tracking techniques used in BSD stacks do not directly translate over to user space network stack infrastructures. However, the exemplary flow table and associated flow tracking logic can provide analogous functionality.

[0341] Various embodiments of the present disclosure can passively track flows so as to improve task scheduling and/or resources management. In one exemplary embodiment, the flow-switch has a flow tracker that tracks flow state and statistics within the flow classification. The flow tracker provides key performance indexes for other components; in particular, other kernel processes can use the flow states and/or statistics to e.g., detect malicious activity, optimize performance, prioritize resources, and/or any number of other optimizations.

[0342] Flow state and/or flow statistics can be gathered and/or inferred from metadata. In some cases, the metadata kernel object (MDK) and/or a metadata user object (MDU) for each packet can be inspected for important event information. For example, MDU/MDK information may be useful to determine when a data flow is active and/or inactive. Active packet flows can be prioritized, inactive packet flows can be de-prioritized or even terminated to free resources for other flows/processes. In some cases, the flow tracker may pro-actively clean up flows that are deemed to be terminated (by either or both of the source and destination); presumably, such flows will not service any more data.

[0343] Moreover, certain types of packets may require expedited handling, and/or treatment outside of the normal data path. For example, certain types of DNS queries and/or TCP control packets have a specific time urgency. Batching these packets with other in-line data may be undesirable because they may be unfavorably delayed.

[0344] Various embodiments of the present disclosure may identify special use packets and identify them within their associated MDK. Nexus flow trackers can check for the presence of these packets by inspecting their corresponding MDK. If a special use packet is identified, then the flow tracker can expedite and/or notify other entities so as to ensure that the packet is prioritized (and delivered with low latency). In other words, the flow tracker may escalate (and/or de-escalate) packet priorities and/or user space networking stack operation based on the contents of the packets.

[0345] Flow Routes

[0346] ARPing/routing is still managed by BSD stack, user space network stack infrastructure flows need to consult BSD stack for information like default gateway MAC, etc, which incurs overhead per packet.

[0347] User space network stack infrastructure flow route is a cache around those BSD info, such that for user space network stack infrastructure flow packets can find those information within user space network stack infrastructure context along with flow lookup. The flow route is notified when related events happen, e.g. route change, ARP expire, to maintain consistency. The flow routes allow for packets going out of the system via user space network stack infrastructure channels to not incur per-packet routing table lookup costs.

[0348] Methods for Flow Management—

[0349] Referring now to FIG. **27**, one exemplary method **2700** is shown for classifying and managing a packet flow by a kernel space entity. As described herein, the terms "classify", "classifying", "classification", refer to characterizing a flow of data with e.g., metadata, a class, a type, or a category. In one exemplary embodiment, one or more characteristics of the flow of data are described with metadata.

[0350] In one exemplary embodiment, a classifier process parses user space metadata and buffers to extract information indicative of the payload and/or how the payload will be used. For example, a classifier may determine whether user space data includes TCP/IP packets; if TCP/IP packets are present, the classifier may additionally extract e.g., tuple information (addresses, ports, etc.) In some variants, the classifier may also perform validations to ensure the packet is of the correct format (e.g., the protocol headers identify a valid length, and the packet is not malformed, etc.). In addition, the classifier may perform other protocol specific checks: e.g., version checks, etc.

[0351] In one exemplary embodiment, user space packets are associated with: a metadata kernel object (MDK), and a metadata user object (MDU). These parallel but distinct objects are created as part of the initial packet allocation process. In some variants, the MDU packet object may include, for example, user space metadata and buflets with indexes to user data in a buffer. In one exemplary variant, the MDK packet object may include, for example, a copy of user space metadata and buflets with pointers to the user data in a buffer.

[0352] At operation **2710** of the method **2700**, a user space packet is written to a pool of resources within the kernel space. In one exemplary embodiment, the packet is generated within a user space communication stack having a different level of trust from the kernel space. While the various aspects of the present disclosure are directed to the aforementioned user space communication stack, artisans of ordinary skill in the related arts given the contents of the present disclosure, will readily appreciate that the principles described herein may be used in any system having tiered privileges and/or trust levels. For example, other systems may have different levels of trust assigned to e.g., first party, second party, and/or third party applications running on the device.

[0353] In one such implementation, the MDK data object is related to, but distinct from the MDU data object. For example, the relationship between MDU and MDK may enable subsequent changes to the MDU to be propagated to the MDK. However, the MDU is distinct from the MDK and changes to the MDK may or may not be reflected in the MDU. In other words, the related but distinct relationship between MDU and MDK enables a one-way transfer of information from the user space to kernel space.

[0354] Still other variants may enable more or less permeability, for example user space data structures (e.g., the MDU) may be sanitized and directly used in kernel space. In other examples, the data structure (e.g., MDK) may be created and entirely stored in kernel space; the user space application can only access the data structure via a fixed set of function calls (e.g., APIs) that limit control. Still other implementations that use a hybridization of the foregoing may be substituted by artisans of ordinary skill in the related arts, given the contents of the present disclosure.

[0355] In some embodiments, the MDK may be created in a variety of different ways consistent with the present disclosure. For example, instead of copying and sanitizing an MDU into the MDK, the MDK can use a pointer to the MDU for any MDU data. In other words, the MDK would reference MDU data rather than a copy thereof. Such variants would reduce the memory size of the data packet objects, however dereferencing the MDK pointer to read the MDU requires more computational steps. In related variants, the MDK can use other forms of reference instead of pointers. For example, the MDK can provide address offsets in the MDU for MDU data. Reading MDU offsets may also require more resources and may not be optimized for cache operation.

[0356] In yet another variant, instead of using MDU metadata, a flow-switch and/or flow classifier (or other kernel entity) can independently generate metadata for the MDK (e.g., the flow-switch may independently determine a TCP/IP header information, etc.). Such variants may require that the kernel entities duplicates work that was done by the user space communications stack (which may be less efficient), however independent operation may provide other benefits (e.g., more security, fewer context switches, etc.)

[0357] At operation **2720**, the user space packet (or a copy thereof) is sanitized by the flow classification process. In one exemplary embodiment, a flow classifier may e.g., ensure that the MDK copy of user space metadata is appropriately formatted, validated, and/or scrubbed for malicious content. In one such variant, the MDK copy of user space metadata can only be used in kernel space after it has been sanitized.

[0358] For example, the user space metadata may include e.g., a TCP/IP header. During sanitization, the TCP/IP header may be checked to ensure that it is properly formatted

(e.g., it contains a source port, a destination port, a sequence number, an acknowledgement number, data offset, checksum, etc.). The TCP/IP header may also be checked to ensure that the values are valid values and that the values have not been tampered with. For example, that the TCP/IP header has not been changed mid-session and does not exhibit potentially malicious traits.

[0359] While the foregoing embodiments describe validating the MDK packet objects associated with the packet, other portions of the packet or operation may also be validated. In some such variants, the MDU packet object may also be checked and/or sanitized so as to e.g., detect malfunctioning behavior and/or ensure that the user space application has not been compromised by malicious activity. In still other examples, the contents of the packet itself and/or the source or destination applications may be verified and/or validated, the user space communication stack(s) may be checked for acceptable operation, and/or the kernel may also verify that its internal processes are operating normally.

[0360] Still other variants may verify that other aspects of the system e.g., network connectivity are behaving correctly. More generally, any number of validations and/or verifications may be substituted by artisans of ordinary skill in the related arts, given the contents of the present disclosure.

[0361] At operation **2730**, the flow classifier, flow-switch, and/or other kernel entity may further augment and/or generate kernel space metadata.

[0362] In one exemplary embodiment, the kernel space classification populates the kernel space metadata. Common examples of kernel space metadata may include without limitation: e.g., transmission status, system wide information, priority information, kernel specific information, and/or any other common classification or categorization. For example, in the context of the present disclosure, the kernel manages and load balances a plurality of different user space communication stacks. Transmission status information (e.g., transmission attempts, retransmission attempts, and/or network congestion, etc.) could be used to infer system activity, and thus may not be provided to the user space communication stack for security and/or privacy reasons. Similarly, each of the individual user space communication stack may be assigned different priorities by the kernel (e.g., a user facing application may be more critical for user experience versus e.g., a daemon or other background tasks). For a variety of privacy considerations, priority information is hidden from the user space communication stacks. More generally, it is appreciated that user space communication stacks should be limited in their exposure to kernel activity.

[0363] In one exemplary variant, the kernel space classification may include a prioritization flag for specific packet types. For example, urgent packets like DNS queries and/or TCP control packets may be expressly flagged within kernel space classification. Various other implementations may use interrupt like processing and/or other priority schemes associated therewith. More generally, the kernel space classification may be based on, without limitation, packet contents, metadata contents, application requirements, source/destination information, device considerations, and/or any number of other factors.

[0364] In one exemplary embodiment, the kernel space metadata may additionally be re-organized and/or re-arranged according to various considerations. Kernel space metadata is only accessible by the kernel, and so data may be arranged in any number of ways that reduces kernel load. In one such variant, kernel space metadata is padded and naturally aligned with natural word boundaries of the processor and/or a local cache. Other common optimizations may seek to e.g., reduce memory footprint, reduce processing complexity, reduce power consumption, improve error rejection, and/or any number of other functions. In some embodiments, a processor system may include multiple different processors and/or multiple different local caches. For example, one common asymmetric multi-processor architecture is a 64-bit central processing unit (CPU) that is paired with one or more 32-bit digital signal processors (DSPs) and/or secure processors (e.g., processors that execute from secure enclaves). Under such asymmetric multi-processor implementations, the natural word boundaries of one processor do not align with the word boundaries of the other processor. In one such variant, kernel space metadata may be aligned according to the processor cache combination that is most likely to access the metadata. In other such implementations, the metadata may be aligned according to the processor cache combination which results in the lowest average penalty. Still other implementations may balance a variety of tradeoffs between the processors; for example, optimizing for a general purpose processor may come at the expense of a hardware accelerator and vice versa.

[0365] At step **2740**, the kernel space metadata may be used directly by the kernel to carry out the user space communication stack operation without requiring further accesses to the user space data. For example, subsequent flow-switch and/or driver accesses for a TCP/IP header during network communications can be made from the kernel space metadata rather than the user space metadata.

[0366] In one exemplary embodiment, any kernel space entity can access an optimized version of the kernel space metadata. In one exemplary variant, an optimized version of kernel space metadata is cache optimized so as to maximize cache hits (where repeated requests for metadata can be served from the local cache memory). Other variants may optimize e.g., memory footprint, processing complexity, power consumption, error rejection, and/or any number of other functions.

[0367] Various embodiments of the present disclosure provide kernel space processes with kernel space metadata and user space processes with user space metadata. This provides a plethora of benefits; for example, rather than parsing the contents out of a packet buffer many times (which could be poorly organized, scattered across protocol headers in the packet buffer, and/or not-naturally aligned), the classifier uses a single access to organize the metadata in a fashion that is optimized based on a variety of system considerations. Subsequent accesses can reap the benefits of the optimized metadata structure throughout the lifetime of the packet.

[0368] Moreover, since user space metadata and kernel space metadata are separate, the various embodiments described herein may optimize each format differently. For example, user space metadata may use index values for fast and secure access within the user space region, whereas kernel space metadata can use pointer values to enable simplified access to data regardless of which user process accesses the kernel space metadata. Additionally, the inherent parallel but distinct data structure ensures that data

transfers between user space and kernel space can be carefully monitored and validated.

[0369] In one exemplary embodiment, the kernel space metadata may be used to forward packets. In one exemplary variant, packet forwarding based on the entries of a flow table allows the system to facilitate optimal forwarding data plane logic. For example, kernel space metadata can be used to switch packets between multiple nexuses. Efficient packet switching between multiple nexuses can effectively "fuse" nexus together; thereby offering a variety of functionalities from different nexuses via a common interface to e.g., a user space application. The techniques described herein allow for the efficient application of traffic rules without separate database lookups, routing tables, or other database reference.

[0370] In one exemplary embodiment, the kernel space metadata may be used to process packets so as to improve overall system operation. Packet processing may include automatic protocol behaviors (forwarding, responding, acknowledgment, etc.) as well as simple data processing (e.g., byte ordering, endian modification, byte masking, concatenation, duplication, etc.). More generally, artisans of ordinary skill in the related arts will readily appreciate that any many processing and/or formatting optimizations may be triggered and/or effectuated on the basis of packet metadata.

[0371] Various embodiments of the present disclosure can passively track flows so as to improve task scheduling and/or resources management. In one exemplary embodiment, flow tracking can be state-ful or state-less. Statistical information can also be gathered and/or other flow metrics. Such information may be used by optimization schemes to e.g., load balance, ensure fairness, optimize device performance, reduce power consumption, improve processing load, and/or any number of other system-wide attributes.

[0372] In still other embodiments, kernel space metadata may be used to a expedite certain special use packets (for low latency). Similarly, kernel space metadata may be used to decimate data to improve network operation, drop packets that are not needed, manage power and/or performance, etc.

[0373] Artisans of ordinary skill in the related arts will readily appreciate the myriad of efficiencies made possible by the techniques described herein. For example, forwarding packets between interfaces can greatly leverage the user space and kernel space metadata, rather than e.g., parsing protocol header at various layers.

[0374] It will be recognized that while certain embodiments of the present disclosure are described in terms of a specific sequence of steps of a method, these descriptions are only illustrative of the broader methods described herein, and may be modified as required by the particular application. Certain steps may be rendered unnecessary or optional under certain circumstances. Additionally, certain steps or functionality may be added to the disclosed embodiments, or the order of performance of two or more steps permuted. All such variations are considered to be encompassed within the disclosure and claimed herein.

[0375] While the above detailed description has shown, described, and pointed out novel features as applied to various embodiments, it will be understood that various omissions, substitutions, and changes in the form and details of the device or process illustrated may be made by those skilled in the art without departing from principles described herein. The foregoing description is of the best mode presently contemplated. This description is in no way meant to be limiting, but rather should be taken as illustrative of the general principles described herein. The scope of the disclosure should be determined with reference to the claims.

What is claimed is:

1. A method for validating user space packet descriptors, comprising:

receiving a plurality of packets from a user space process;

for each packet of the plurality of packets:

extracting a data structure from the packet;

validating that the data structure conforms to a packet descriptor format; and

providing validated packets to a kernel space process.

2. The method of claim **1**, wherein extracting the data structure from the each packet comprises extracting a pattern from a packet header associated with the each packet.

3. The method of claim **2**, wherein validating that the data structure conforms to a packet descriptor format comprises checking the pattern based on a random number and an address associated with the each packet.

4. The method of claim **1**, further comprising terminating the user space process when at least one packet is invalid.

5. The method of claim **1**, wherein validating that the data structure conforms to a packet descriptor format comprises checking that the data structure comprises a valid checksum.

6. The method of claim **1**, wherein validating that the data structure conforms to a packet descriptor format comprises checking that the data structure comprises a base offset from an address.

7. The method of claim **6**, wherein providing validated packets to the kernel space process further comprises translating the base offset from the address to a pointer in kernel virtual address space.

8. A method for tracking user space packet descriptors, comprising:

receiving a plurality of packets from a user space process;

determining a connection state based at least in part on a plurality of packet descriptors associated with a first set of the plurality of packets from the user space process; and

transmitting the first set of the plurality of packets from the user space process when the connection state is valid.

9. The method of claim **8**, further comprising determining a second connection state associated with a second set of the plurality of packets from the user space process; and

rate limiting the second set of the plurality of packets from the user space process when the connection state is invalid.

10. The method of claim **9**, wherein the second set of the plurality of packets from the user space process comprise synchronization packets (SYN) that are sent while one or more acknowledgement packets (ACK) packets have not been received.

11. The method of claim **9**, wherein the second set of the plurality of packets from the user space process comprise reset packets (RST) that are sent without an outstanding socket connection.

12. The method of claim **9**, wherein the second set of the plurality of packets from the user space process comprise reset packets (RST) that are sent without an outstanding synchronization packet (SYN).

13. The method of claim **8**, further comprising generating one or more kernel space statistics based on the first set of the plurality of packets from the user space process when the connection state is valid.

14. The method of claim **8**, further comprising determining a second connection state associated with a second set of the plurality of packets from the user space process; and

ignoring one or more user space statistics based on the second set of the plurality of packets from the user space process when the connection state is valid.

15. A method for propagating sensitive network statistics, comprising:

receiving network statistics from a user space process;

validating the network statistics based at least in part on a plurality of packet descriptors associated with a first set of a plurality of packets from the user space process;

updating kernel space network statistics with the network statistics from the user space process based on the validation; and

generating an obfuscated set of network statistics for a second user space process based on the kernel space network statistics.

16. The method of claim **15**, wherein receiving the network statistics from the user space process includes receiving round trip delay time (RTT) data.

17. The method of claim **16**, wherein validating the network statistics based at least in part on the plurality of packet descriptors associated with the first set of the plurality of packets from the user space process comprises determining an upper and a lower RTT bound.

18. The method of claim **15**, wherein the user space process comprises a user space library that is self-contained.

19. The method of claim **18**, wherein the second user space process comprises a user space application that has entitlements to access the network statistics generated by the user space library.

20. The method of claim **19**, wherein the user space library and the second user space process are operating from a common application space.

* * * * *