Combined stateful classification and session splicing for high-speed NFV service chaining

Tom Barbette[†1], Cyril Soldani[2] and Laurent Mathy[2]

[1]KTH Royal Institute of Technology
[2]University of Liege

***Abstract*—Network functions such as firewalls, NAT, DPI, content-aware optimizers, and load-balancers are increasingly realized as software to reduce costs and enable outsourcing. To meet performance requirements these *virtual* network functions (VNFs) often bypass the kernel and use their own user-space networking stack. A naïve realization of a chain of VNFs will exchange raw packets, leading to many redundant operations, wasting resources.**

In this work, we design a system to execute a pipeline of VNFs. We provide the user facilities to define (i) a traffic class of interest for the VNF, (ii) a session to group the packets (such as the TCP 4-tuple), and (iii) the amount of space per session. The system synthesizes a classifier and builds an efficient flow table that when possible will automatically be partially offloaded and accelerated by the network interface. We utilize an abstract view of flows to support seamless inspection and modification of the content of any flow (such as TCP or HTTP). By applying only surgical modifications to the protocol headers, we avoid the need for a complex, hard-to-maintain user-space TCP stack and can chain multiple VNFs *without re-constructing the stream multiple times*, allowing up to 5x improvement over standard approaches.

This paper is the accepted version published by IEEE/ACM Transactions on Networking. The final version is available at https://ieeexplore.ieee.org/document/9503101, DOI: 10.1109/TNET.2021.3099240

I. Introduction

According to J. Sherry, *et al.* [1] and V. Sekar, *et al.* [2], there are roughly as many middleboxes as routers in enterprise networks. Additionally, a myriad of middleboxes are deployed in mobile networks[3]. These network functions provide network security and performance enhancements. This middlebox functionality can be implemented in software on commodity hardware using Network Functions Virtualization (NFV), rendering these middleboxes less opaque and supporting the outsourcing of middlebox functionality to the cloud. In NFV settings, as in traditional networks, packets often pass through several middleboxes, leading to the need to chain several Virtual Network Functions (VNFs) together.

Unfortunately, these VNFs perform redundant tasks, such as session identification or other network stack processing, thus wasting cycles & memory and increasing latency. However, avoiding these redundant operations is difficult, as each VNF has its own requirements, such as inspecting specific traffic classes, be stateless, tracking statistics at different level of aggregations (*e.g.* group packets per IP pair, 4-tuples, ...), or modifying a TCP or QUIC bytestream. The proposed unification must preserve isolation and the ability to identify & shut off a misbehaving VNF.

† Includes work done while at the University of Liege

Previous work such as SNF[4] and OpenBox[5] synthesize traffic classifiers. At best, systems such as μNF[6] remove redundant and identical processing modules, but an HTTP classification module cannot benefit from, *e.g.* a prior NAT classification. E2[7] and microboxes[8] allow some VNFs to pass a bytestream of TCP payload between VNFs, therefore avoiding redundant TCP management, but cannot modify the stream without terminating the connection.

In this paper, we present the design, implementation, and evaluation of a system prototype where packets begin their journey via a unified flow manager responsible for classification. This classification is reused by all subsequent VNFs.

The unified flow manager combines the traffic class classification from all VNF components. When the Network Interface Controller (NIC) has flow classification capabilities this classification is**offloaded to the NIC**. In contrast, traditional service chains involve software routing between VNFs, preventing such offloading. Additionally, the flow manager handles the sessions for each VNF component, avoiding the need for multiple, often identical, hash tables along the way to find each packet's session. By unifying the traffic class classification and session mapping of all VNFs of a service chain: (i) each field of a packet is only looked at once for the a given value and (ii) the classification and session mapping are reused across the entire chain. To our knowledge, we are the first to propose an algorithm to **automatically combine redundant**

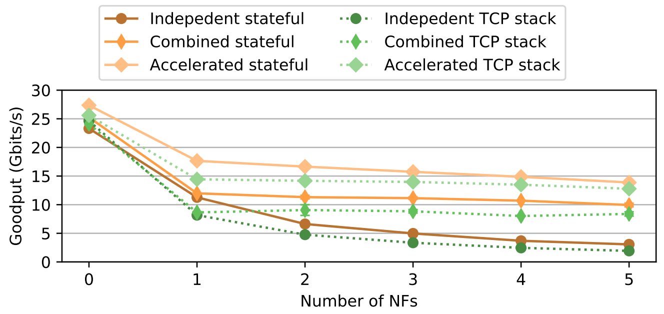


Figure 1: Impact of re-doing classification on a chain of stateful Network Functions (NFs), and TCP-based NFs. State management has a high performance cost, which is better paid only once. Our light in-the-middle stack built on top of the session manager has a very low overhead.

and overlapping session mapping, in a single flow table that accommodates every VNF in a chain. The orange and continuous lines in Figure 1 show the advantage of combining the state of a series of stateful NFs (requiring a few bytes per 5-tuple) and accelerating the (unique) session classification using the NIC *(more details about the experiment can be found in Section V-D)*. Combining the session classification of the NFs improves performance by 5.2×. While previous work (such as [7] and [8]) only combined TCP state, in this work, we do not differentiate based on what is done on top of the session management. As a result, our proposal combines and manages sessions of stateful firewalls, NATs, or full TCP applications – as will be demonstrated.

The framework provides a zero-copy *stream abstraction* and easily supports new protocols on top of other protocols, thus enabling modification of packets of a given session *without* knowledge of the underlying protocols. For example, when an HTTP payload is modified, the content-length must be corrected. A layered approach must propagate the effect of stream modification across the relevant lower layers; therefore, we provide a TCP-in-the-middle stack which can on-the-fly modify the sequence and acknowledgement numbers on both sides of the stream when the upper layer makes changes. This **enables modification of the number of bytes in the stream *without* terminating the connection and** without requiring a full TCP stack. In contrast, other proposals still need a full TCP stack - even for small modifications of the stream. Figure 1 shows the cost of our in-the-middle TCP stack is very low, and like the session classification, can be shared by multiple VNFs to accelerate the processing by 7X compared to using individual copies of our stack.

Moreover, our approach enables future innovations, such as new TCP extensions since the protocol stack only needs to understand how to modify the flow and implement only a limited amount of TCP semantics, as the stack behaves as transparently as possible, being agnostic to changes in congestion or flow control algorithms and retransmission techniques. Support for greater semantic changes (such as TCP FastOpen[9]), only require a few lines of code. In contrast, all components written to modify streams, such as TCP/HTTP payload would directly work on a new QUIC[10] component.

Our system combines the advantages of efficient, end-to-end & non-cooperative systems, such as DPDK[11], Netmap[12], Arrakis[13], IX[14], and specific user-level stacks [15], [16], [17], [18], with contrasting approaches that build upon reusing components, such as CoMb[2], OpenBox[5] and μNF[6]. Therefore, our design combines efficient consolidation with tailored services. If no TCP reconstruction needs to occur along the service chain of the VNFs, then reconstruction does **not** happen; while if multiple VNFs need it, it is done only once. In contrast to prior work, we offer a practical, low-level approach to build a high-speed NFV dataplane.

Section II explains how we unify classification and combine the VNFs along the service chain. Section III discusses implementations details, such as how, in practice, a VNF can expose classification details together with possible execution models. This induces a highly parallel and non-redundant stream architecture that can be used to support multiple protocols, as explained in Section IV. Finally, we evaluate the performance of the prototype in Section V. Section VI reviews the state of the art and states our specific contributions.

II. Traffic class and session unification

Figure 2 shows an example of a simple service chain of three VNFs. It realizes a per-session (stateful) load-balancer for both UDP and TCP traffic, but passes HTTP traffic (*i.e.* TCP packets with destination port=80) through a HTTP filter (*e.g.* a parental filter or ad-remover), while dropping other TCP traffic. UDP traffic to 10.0.0.27 with destination port 6970 goes through a NetFlow monitor (*i.e.* a per-flow tracker that count bytes, packets, *etc.*) that allows operator to monitor that specific traffic class (*i.e.* RTP flows).

In this paper, **we decouple flow management** from the VNFs – as shown in Figure 3. Each VNF declares the kind of packets it accepts (HTTP packets, all TCP packets, . . .) and, if needed, the definition of the session it requires (*e.g.* packets grouped by IP pair, by TCP 4 tuples, . . .) and a number of bytes per-session (shown in the hexagons in Figure 3), *i.e.* the per-session scratchpad. In our example, the flow monitor, the load-balancer, and the HTTP filter need a per 5-tuple scratchpad to write some per-session metadata. The flow manager assigns (for each session), a *Flow Control Block (FCB)* containing a sufficient number of bytes for each of the VNFs (potentially laid out as shown at the bottom of Figure 3. Traffic class information given by all VNFs is used to reduce the overhead of session classification and remove static values from the tuples. Static classification can be offloaded to a modern NIC, thus accelerating session classification.

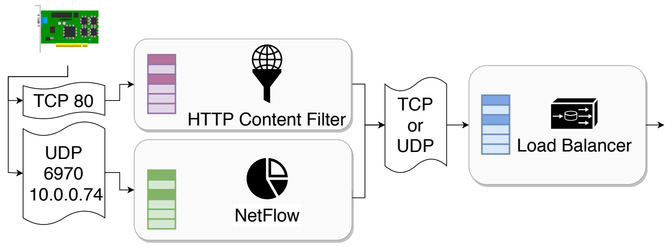


Figure 2: A small NFV chain monitors some UDP traffic, filters HTTP content, and then load-balances all packets.

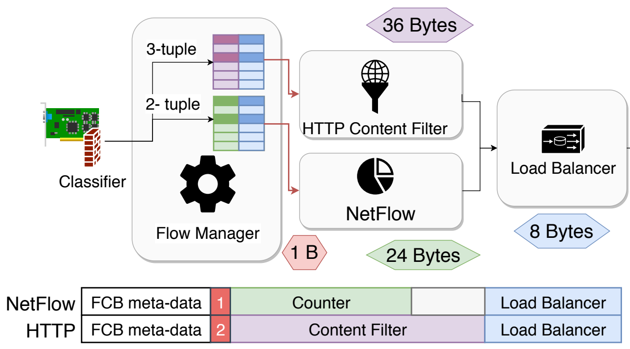


Figure 3: Computation of the size and offsets needed for the FCBs. Nodes contain the size they require in the FCB.

A. *Traffic classes*

In Figure 2, each flag is a classification step usually handled by reading the corresponding packet field(s) to determine the next VNF for the packet. Following the various paths of the graph (as proposed by OpenBox[5] and SNF[4]), one can build a flow table encoding each encountered traffic class as a rule. Table I, shows such a flow table for the example in Figure 2. As in prior work, the classifier can be minimized by removing redundant classification operations. However, these earlier systems either return the packet to the controller after each VNF (inducing extra latency) or duplicate VNFs for each possible traversal (which is not always suitable for state reconciliation.) In contrast, each of our rules corresponds to a FCB. Instead of classifying packets at each step of the processing chain all the necessary information will be included in the FCB, as a list of next hops that describe the paths between the VNFs. Note that traffic classes are not limited to being defined based upon packet fields. *E.g.*, an ISP could define a traffic class as packets from one ingress to a specified egress, the combination determining the processing to apply.

Table I: Rules for each possible path in Figure 2 and their corresponding FCBs after receiving some packets.

Rules							Flow Control Block	
Ethertype	ARP Type	Proto	Dport	Sport	Dst	Src	Next hops	Session space
IP	*	UDP	6970	21567	172.16.29.47	10.0.0.27	1	0x9edcc200, 1
IP	*	UDP	6970	PFP	PFP	10.0.0.27	1	NetFlowState, int
IP	*	UDP	*	*	*	*	DROP(1)	-
IP	*	TCP	80	52100	10.0.0.1	89.18.17.216	2	0x1a234579, 0
IP	*	TCP	80	52100	10.0.0.17	120.12.17.12	2	0xab38977d, 1
IP	*	TCP	80	PFP	PFP	PFP	2	HTTPSession, int
IP	*	TCP	*	*	*	*	DROP(2)	-

B. *Sessions*

In addition to the traffic class, we allow each component to describe the sessions they require and an amount of space they need per-session for their metadata (to be stored in the scratchpad). To define sessions, *Populate From Packet (PFP)* rules allow special header wildcards. These entries require that when the rule is matched the rule be duplicated with the exact values of the fields. For example, an HTTP content filter will define its session as $proto = TCP; dstport = 80; srcipaddr = PFP; dstipaddr = PFP; dstipport = PFP;$. In this example, the space per session required by the HTTP content filter is 36 bytes and this will be allocated in the FCB's session space (its placement is described in Section II-C). When a rule containing a PFP field is matched, the rule is duplicated and the PFP fields are replaced with their actual values from the packet. The FCB will be duplicated along the way, with the pre-allocated space that each VNF asked for (as showcased by highlighted lines in Table I). As our example only considers HTTP packets with the destination port 80 while dropping others, the session mapping will only be based on the last 3 tuples. For the flow monitor, as the destination is known the session is only based on the 2 tuples.

C. *FCB size*

All components of the service chain expose the size of the per-session scratchpad they require. Figure 3 showed an example of those bytes needed for each component of the earlier NFV chain as an ordered layout for some of the FCBs. As packets of the same flow usually take a unique path through the components, the same amount of space can be assigned to parallel paths thus optimizing the allocation of space. To avoid indirection, we waste some space by using an offset *independent* of the input path, thus each component has one and only one offset within all potential FCBs assigned to packets passing by, hence leading to some unused space along some path.

The offset in the FCB for each component is the amount of bytes needed for the **greediest** path that leads to it, *i.e.* the maximum number of bytes needed for every encountered component for any path. We call this the maximal offset. Simply setting the maximal offset as each component's allocation in the FCB is sufficient - if packets cannot follow parallel paths - as in the case in Figure 3, where a packet cannot be both a TCP and UDP packet at the same time. In Figure 3, one needs 1 byte to reach the flow monitor or the HTTP filter. Therefore, the maximal offset to the load balancer would be $1+37 = 37$, as the path through the HTTP filter requires more bytes than the flow monitor ($1+24 = 25$). When packets from the same flow can only take one path, the algorithm achieves the best possible ordered placement, as the maximal offset is the first position that accommodates enough bytes for each prior component in any path and does not lead to collision (as by construction each downstream component accounts for each upstream component).

However, to take advantage of parallelism or when packets of the same flow can take different paths (*e.g.* counters that count different events), some components cannot share the same scratchpad space. To account for this case, before setting the maximal offset as the final position of the component in the FCB, we compute the set of reachable components from the component and build a bitvector with a bit set to 1 for each corresponding byte that is already assigned in the reachable components. Then, starting at the maximal offset, we search for sufficient place in the bitvector to accommodate the component's required space.

To create compact layouts while allowing non-ordered access, we implemented a variant of the algorithm which places the most frequent components (in terms of the number of appearances across paths) first, by trying placements starting from the beginning of the FCB and increasing the offset when a collision occurs, therefore filling holes when possible while placing the most-seen element first. Thus, in Figure 3, keeping the Load Balancer data for the UDP traffic at the beginning of the block minimizes the space needed for UDP FCBs. As FCBs are pool-allocated (see Section II-D) and offsets must be unique, non-ordered access would only occur for HTTP flows. We leave the performance assessment of non-ordered compact placement for further study.

D. *Classification tree*

The classification algorithm we propose in this section is an example of how to implement a flow table that allows dynamic duplication of some rules. Traffic class and session classification are both implemented using the *same* tree. Each level of the tree corresponds to one field. Figure 4 shows the classifier for the example shown in Figure 2.

In the tree of Figure 4, each node (shown as a rounded square) is linked to a “level” object (shown as a call-out) that defines the programmable field’s implementation, *i.e.* how to extract data from the packet for header fields. The pseudo-code for matching is given in figure 5. Most levels are header classifiers that will read a field of the packet, apply a mask, and then return the value so the node’s implementation can access a child based on this value.

Matching consists of descending the tree until a leaf is reached. Leaves are FCBs, *i.e.* not node objects. All nodes also have a default branch. Level objects indicate whether the field is a PFP field, in which case the default node and the child FCB are duplicated upon a miss, producing a per-session FCB. An FCB can be marked to apply early drop directly in the flow manager (if the VNF operator allows it). This is also useful for debugging purposes as it can identify which VNFs drop a certain class of traffic. However, this is not always desirable as an operator may want to keep statistics about dropped packets.

After the tree is synthesized, it undergoes an optimization to tailor the implementation for each level according to the number of children. Depending on the associated field or user-provided hints about the best underlying implementation, each level can have different implementations: a condition - if there are two possible outcomes, a heap - if there is only a few static cases, a table - if the range of possible values is small (such as with VLAN numbers or 1-byte fields), or a hash table in all other cases. The tree creation process warns the user when a path is impossible, such as placing a UDP-based VNF after a TCP-based VNF. Potential future improvements would be to replace the implementation of connected static levels of the tree (without PFP fields) by an efficient static classification algorithm (*i.e.* with fast lookup time, but slow update time), such as EffiCuts[19] or HyperCuts[20]. With our tree design, one only needs to register a new implementation for a given level. In practice, we preferred to offload all static classification to the NIC, making the software implementation performance irrelevant as explained in Section II-F.

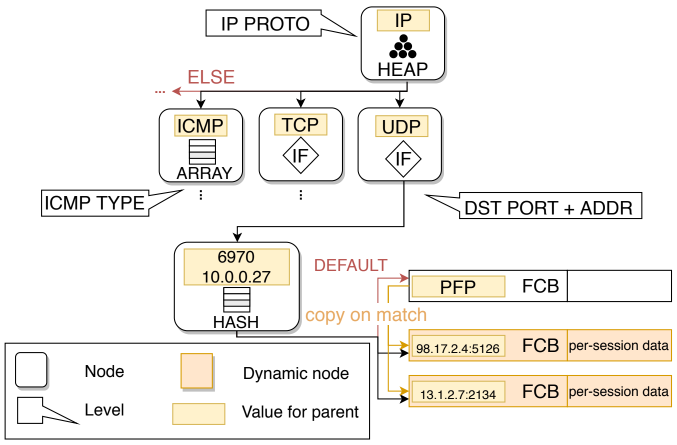


Figure 4: Part of the classification tree for the example in Figure 2 (omitting TCP and system’s classification such as ICMP, ARP, *etc.*).

```
(1) fcb* match(packet) {
    node = root;
    while (node is not leaf) {
        field = node->level->get_field(packet);
        child = node->find(data);
        if (!child)
            if (node->level is dynamic)
                child = node->default->copy();
            else
                child = node->default;
        node = child;
    }
    return (fcb*)leaf;
}
(2) bool verify(packet, fcb) {
    node = fcb->parent;
    field = fcb->field;
    do {
        if field != node->level->get_field(packet)
            return false;
        node = node->parent;
    } while (node is not root);
}
```

Figure 5: Pseudo code for (1) the traversal of the classification tree to find the FCB matching a packet and (2) verify a given FCB is matching a specific packet.

FCBs are managed in per-thread pools for efficient allocation and recycling. Both nodes and FCBs have their first bytes reserved for the value of the parent level. If the flow manager is to be traversed by multiple threads, a special dynamic “thread” level will be inserted before the first dynamic level. When a new thread pass through the thread level, it will duplicate its default children nodes for the current thread, under the assumption that packets of the same session are handled by the same core, a feature allowed by RSS hashing.

E. Classification tree expansion

When the default implementation of a dynamic node is insufficient (*e.g.* it reaches its maximal capability, it produces a lot of collisions for a hash table, . . .) the node starts to grow, which is marked by a flag in the node. In this case, the default path of the node is replaced by a new empty node that uses the same level but a more appropriate implementation, *e.g.* a bigger hash table, or when the hash table size is close to the number of possible values (*i.e.* the 65536 possible values for the 16-bit TCP/UDP port fields) then use a vector with one path for every possible value. When a node grows such that no child can be added, the classification only passes through it looking for existing flows. When all children of a growing node are removed (*i.e.* all flows timed out or finished), the growing node is removed and only the new node remains. This scheme avoids the jitter caused by growing hash tables, which normally requires a full re-allocation of the buckets. It allows transformation of a badly performing implementation into another one, *e.g.* changing the hash function of a hash table, moving from linear probing to open addressing, *etc.* The level tracks the node’s growth, therefore it can propose an appropriate implementation for new node(s).

F. Hardware acceleration

As individual software elements in a service chain cannot access the NIC, combining the classification upfront enables offloading of all or a part of the classification to the NIC. As a first step towards hardware offloading, we use the RSS hash given by the NIC as an index into a cache to directly find the FCB, a technique already used by OpenVSwitch[21] to classify a few most-seen rules. As multiple sessions can map to the same RSS hash, one still has to verify the fields' values. This is done by traversing the tree from the leaf to the root (see pseudo code shown in Figure 5). The bottom-to-top traversal verifies each field is equal to the node's child value and has a complexity proportional to the depth of the tree (*i.e.* the number of fields minus potential optimizations). This is much faster than a top-down descent where each node performs a lookup (in *e.g.* a hashtable) to find the child node for the given field's value, as this is proportional to the *sum* of each node's own lookup method's complexity.

With the increasing classification capacity of commodity NICs, the traffic class classification (of non-PFP fields) can be offloaded to some recent NICs, such as our Mellanox ConnectX 5[22]. Therefore, we propose an extended version of our flow manager that offloads to the NIC the static part of the tree. One rule is installed in the NIC for each path to a FCB or a dynamic node. The rule uses a "mark" action to associate an id with the packet. This is subsequently used by software to directly access the final object of the path via an indirection table. We did not implement dynamic classification offloading. In an NFV context, we target millions of flows per second, but the NIC is only able to install a few thousands rules per second. Techniques such as offloading only elephant flows[23] fall outside of the scope of this article.

We can use the RSS-based cache and offload the static classification at the same time. For instance, the hardware will mark TCP packets with destination 80, and then the cache will find the correct dynamic FCB. However, the flow manager still needs to ascend in the tree from the leaf (the FCB) to the first non-PFP field (*i.e.* the last static node referred to by the mark) to verify the FCB is not from a colliding flow for the 3-tuple. Both techniques and their combined performance improvement are evaluated in Section V.

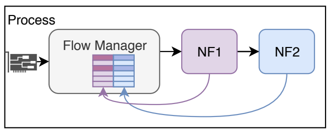
G. FCB release

Each FCB has a usage counter. When a packet matches a FCB, the FCB usage counter is incremented, and when the packet is released, the usage counter is decremented. VNFs can declare an amount of time the FCB will stay alive after its usage counter reaches zero (as packets of the same session might soon arrive). The FCB is released when the usage counter reaches 0 **and** the timeout has passed. Upon release, an optional chain of user-defined functions is called to clean up any FCB state (as needed). The FCB is returned to the pool but removed from the tree. All nodes and leaves (FCBs) have a parent pointer so all dynamic (PFP-duplicated) parent nodes having no remaining children can be removed to prune the tree.

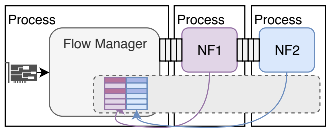
The main mechanism for ageing is lazy deletion. Upon collision when looking up for FCBs in the tree, the FCB encountered during the traversal will be verified for expiration. If it is expired, the FCB will be renewed by the classifier (*i.e.* the release function is called and the FCB's space reset to its initial value). As collision may never happen, garbage collection must still be ensured. If the timeout has not passed when the usage counter reaches 0, the FCB is added to a list of pending timeouts. The list of pending timeouts is traversed when the system is idle (*i.e.* polling all input devices gave no packet) or the length of the list of pending timeouts exceeds a threshold. As growth of the list is normal, the threshold is set using exponential back-off. A timer triggers a list pruning every 15 seconds, thus providing a fixed upper bound on the release time.

H. Isolation between VNFs

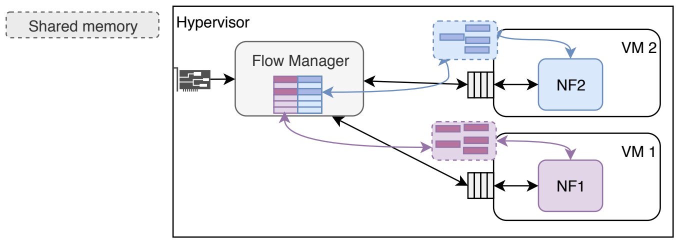
Figure 6a shows a single-process deployment of the service chain, where each VNF is a distinct object that can directly access the flow table. Our prototype (see section III) follows this approach, because passing packets between functions only involves functions calls and the flow table is directly accessible from each VNF.



(a) Deployment of a chain of VNFs in a single process



(b) Deployment of a chain of VNFs with one process per VNF



(c) Deployment of a chain of VNFs using virtual machines

Figure 6: Access to the shared flow table under various isolation levels

To integrate third-party applications, one must use shared memory to first, pass packets between processes, and secondly, allow access to the flow table from the other processes as shown in Figure 6b. The flow manager allocates FCBs from a shared memory zone to allows every process to access it. Pointers to FCBs are passed with the packets, *e.g.* as metadata. This model allows access of one VNF to the scratchpads of other VNFs. However, isolation between VNFs can be enforced by using indirect memory allocation, where the FCB contains only pointers to memory allocated from pools shared between the flow manager and each NF. This scheme can also be used with virtual machines (VMs), as shown in Figure 6c. For each VNF, the hypervisor reserves enough space in the FCBs to keep a pointer rather that the session data itself. When a “PFP” rule is matched, the FCB will be duplicated. Thus, when a flow is seen for the first time, the hypervisor allocates the space needed per-session for each VM in guest memory. This enables fast allocation of per-session space in the guest system without traversing per-VMs flow tables, while ensuring memory isolation. Unlike our prototype, a practical implementation needs a new API to pass the allocated memory pointer together with batches of packets from the same session.

Indirect memory allocation can also be used if the data needed by a VNF component has variable size, as the VNF simply ask for space to fit its static data and a pointer. The pointer can then be used to keep a reference to memory allocated when the flow is first seen using another dynamic mechanism, such as an efficient pool-allocation system.

I. Flow manager placement

The flow manager does not necessarily need to run as the first VNF, hence it may run after other VNFs. In the end, it is a VNF like any other VNF; however, it provides certain services for the downstream VNFs. For example, consider an anti-DDOS system, such as a heavy hitter detector, followed by stateful VNFs (e.g. a load balancer). It would be counter-productive to run a stateful classification *before* the anti-DDOS module. In this case, the flow manager is placed (i) before the anti-DDOS module, so it will offload all traffic class classification (in practice there will always be some ARP handling, IP header parsing, etc – independently of the first VNF) **and** (ii) after the anti-DDOS module. The first instance defines a few "static" FCBs, doing almost nothing in software, while the second flow manager's tree will be probably be limited to a single level of stateful classification and will be pruned according to the traffic classes of the first flow manager. Only packets passing the heavy hitter detector will be processed by the second flow manager, which will replace the current FCB pointer with a new dynamic one.

III. Prototype implementation

We modified FastClick[24], an extension of the Click Modular Router[25] to allow its basic building blocks (called elements) to expose their traffic class and session needs to the flow manager. Click allows piping simple networking elements together to build a more complex NF using a simple language. The elements themselves are written in C++, and implement a few virtual functions to handle packets and events. Many believe that Click is limited to its native elements in C++. However, FastClick can exchange packets with applications through various I/O systems (such as Kernel sockets), but also much faster means such as DPDK rings (allowing it to also pass metadata needed to pass the pointer to the FCB along with the packets) or netmap pipes (allowing to exchange packets between applications running in different VMs[26]). Therefore, we picked FastClick as a very efficient service-chaining tool while avoiding re-writing of every applications. The flow manager could be designed as a socket extension, allowing applications to open a socket with a new option to expose the traffic classes and the session needs. The socket would allow receiving packets with a link to the space in the FCB. In contrast, we prefer a userlevel approach because Kernel I/O would reduce performance of inter-VNF communication.

In our prototype, called MiddleClick, the flow manager is implemented by the `FlowManager` element that must be placed before any element that would access the FCB. This is enforced at configuration time. The `FlowManager` traverses the graph, using the traffic class and session definitions exposed by the elements of the graph to build a flow table as explained above.

A. Traffic class and session definition

Figure 7 shows a Click element that defines a rule to verify the IP protocol field is TCP (10/06), a standard 4-tuple for a TCP session, and asks for some space to store a structure in the FCB by extending a specific C++ class. The classification rule is directly used by the flow manager to build the table. The *FLOW_IP* constant (line 14) (described in Section IV-A is used by the previous context (Ethernet in this case) to spawn a rule for IP packets themselves. This supports tunneling, as each layer has a way to define the next protocol. The inheritance class (line 7) is a higher level abstraction avoiding registering some virtual functions to expose the size of the *NATEntry* structure and directly passes to the *push_flow* function a pointer to where that structure is in the FCB.

```
1  //Per-session data kept for the NAT
2  struct NATEntry {
3      PortRef* ref;
4      bool fin_seen;
5  };
6
7  class FlowNAT:
8      public FlowStateElement<FlowNAT,NATEntry> {
9  public:
10
11    [...]
12
13    FLOW_ELEMENT_DEFINE_SESSION_CONTEXT(
14    "10/06 12/0/ffffffff "
15    "16/0/ffffffff 22/0/ffff 20/0/ffff", FLOW_IP);
16
17    void push_flow(int, NATEntry*, PacketBatch *);
18 }
```

Figure 7: Example of session definition for a self-contained NAT element. The novel API is written in purple.

B. Service chain definition

In Click, a service chain is defined as a set of elements piped together. Dispatching traffic according to header fields - *the traffic class classification* - is done using a `Classifier` element (or a variant such as `IPClassifier` that provides more convenience), to dispatch traffic to subsequent elements according to a given set of rules, as shown in Figure 8(a). In MiddleClick, the `Classifier` is modified to expose its rules as a set of traffic classes. Therefore, the `FlowManager` includes the classifier's rules in the flow table and sets the next hop number according to the Click output path in the FCB. Thus, the `Classifier` simply reads the next hop number in the FCB to decide the output, without classifying in place or even touching the packet.

Alternatively, Figure 8(b) illustrates a new link syntax called a context link, `~>`, which will automatically place a `Classifier` element according to the traffic classes exported by all elements to the right of the arrow. Context links remove the needs for obvious classification. In our example, by using the context link the input can directly be tied to all ARP elements, the flow defined by the ARP elements will be used pass ARP requests to the ARPResponder, replies to the ARPQuerier, and other packets to the remaining paths. In many cases, the element will always ask for the same packets and an explicit `Classifier` is unneeded.

```
FromDevice(...)
  -> Classifier(12/0800) // Accept only IP packets
  -> cp :: IPClassifier(proto udp, proto tcp);
cp[0]
  -> IPClassifier(dst udp port 6970 &&
                  dst ip addr 10.0.0.27);
  -> t :: NetFlow();
cp[1]
  -> IPClassifier(dst tcp port 80)
  -> h :: HTTPProcessor();
```

(a)

```
fd :: FromDevice(...);
fd
  ~> FlowDispatcher(dst udp port 6970 &&
                    dst ip addr 10.0.0.27)
  -> t :: NetFlow();
fd
  ~> h :: HTTPProcessor();
```

(b)

Figure 8: (a) Click configuration for the example of Figure 2. (b) Corresponding MiddleClick configuration.

If some elements rewrite headers that have previously been classified, such as a NAT, no re-classification is done, as in most cases the packets belong to the same session. If the session must change (*e.g.* dividing a flow into multiple subflows in a dynamic way), a new `FlowManager` placed in the chain will assign new FCBs, therefore new sessions. It is up to the NF operator to ensure the service chain is still correct after rewriting, *e.g.* a firewall placed after a NAT does not classify on the original addresses. Alternatively, SymNet[27] or another similar system may allow verification of correctness for similar configurations.

IV. Stream Abstraction

At this point in our design, a VNF developer can easily receive a batch of raw packets matching a given traffic class along with their FCB. The developer knows the traffic class of the packets, as this is marked in the FCB. If the component asked for some per-session scratchpad and the VNF component has space for its own use in the FCB, thus the FCB is duplicated per-session and this space is shared by all packets of the same session. Most of the time, a VNF developer expects a seamless stream of data of a given protocol, rather than simply packets matching a given set of tuples. The developer also wants a way to touch the data without caring about the protocol's details. Therefore, we introduce the stream context concept.

A. Contexts

VNF components exchange batches of packets of the same session. A stream context allows *requests* to act on or modify the stream. Within in a given context, components can use a *content offset* to access metadata associated with each packet or to directly access the payload. Additionally, we offer multiple abstractions to act on the data as a stream, *without* the need to copy the packet's payload, similar to an iterator that can iterate across packets. This enables zero-copy inspection of a stream and allows a VNF component using this higher-level stream abstraction to access headers as needed (a feature that all intrusion detection systems (IDSs) need as attacks may be based on header fields).

In the case of TCP, the context offset follows the TCP header of each packet. Each context independently handles a request and then passes the request to the previous context. Modificationa that impact the size of a stream are via requests to *add bytes* or *remove bytes*. An example, of adding bytes is showed by the green boxes and lines in Figure 9. Modification of the number of bytes in a TCP stream implies adjustment of acknowledgement and sequence numbers (see Section IV-C). If the packet length changes, the IP header also needs to change so the request is passed to the lower layer.

In addition to functions to modify packets, the context can *determine if a given packet is the last useful packet* for the current context, *i.e.* the packet has set the FIN flag for a TCP connection. In an HTTP context, the value of *Content-Length* is passed to the previous context or TCP if the previous context is unknown. This context *closes the current connection* when the context supports a stateful protocol, such as TCP and *registers a function* to be called when the connection terminates (to cleanup the session scratchpad).

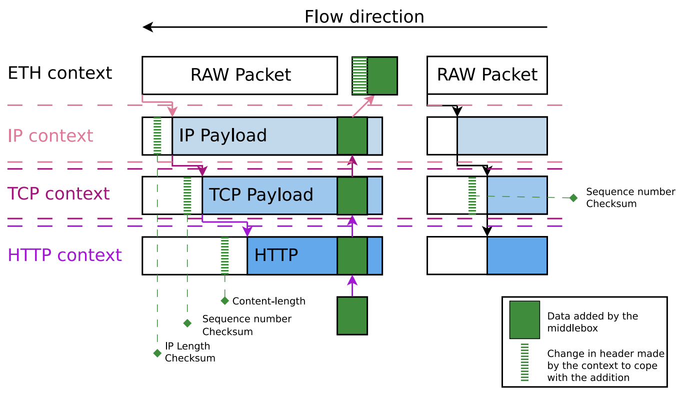


Figure 9: Context approach. Upon entry into a context, the payload offset is moved forward. When the stream is modified, prior layers take care of the implications (see Section IV-C).

Figure 10 shows a checksum computation element using a per-chunk stream iterator. The *process_data* function is called when a batch of chunks of payload is available. When the stream closes, the *release_stream* function is called to do something with the final checksum. The checksum could be checked against a database of known dangerous payloads or the *process_data* could return a negative value to terminate the connection by triggering the close connection request.

B. TCP Flow stalling

One may want to buffer data before forwarding it, *e.g.* execute a pattern match and avoid any part of the pattern being forwarded, a feature especially important for network-based parental control, DPI, and ad-removal. While our platform is protocol-agnostic, the TCP case shows it could operate. As a TCP source may wait for an ACK from the destination before sending more packets, buffering data may prevent the

```
struct fcb_cksum {
    unsigned int fcb_cksum = 0xffffffff;
};

class FlowChecksum:
  public ContextChunkBufferElement<FlowChecksum,
               fcb_cksum> {
  public:
    [...]
    int process_data(fcb_cksum*,
                     FlowBufferChunkIter&);

    inline void release_stream(fcb_cksum*) {
        //Do something with fcb_cksum->cksum;
    }
};

int
FlowChecksum::process_data(fcb_cksum* fcb,
        FlowBufferChunkIter& iterator) {
    while (iterator) {
        auto chunk = *iterator;
        fcb->cksum = update_cksum(fcb->cksum,
                (char *) chunk.bytes, chunk.length);
        ++iterator;
    }
    return 0;
}
```

Figure 10: Code for a stream checksum computation element.

destination from sending that ACK, thus leading to a deadlock. The TCP context provides functionality to pro-actively ACK.

If enabled, when the TCP context receives a *request for more packet*, it sends an ACK for the given packet to the source with an acknowledgement number that the destination would have sent. Therefore, we maintain outgoing pre-ACKed TCP packets in a buffer until the destination ACKs them. Buffering is done when a VNF component specifies that it may stall or modify packets, or when the component wants to protect against overlapping TCP segment attacks[28]. Functions that do not need to see a stream of data, such as NATs or load-balancers, do not need to keep outgoing packets in buffers, as processing retransmissions does not pose any problems. Buffering uses reference counting to avoid copying packets. Packets leaving the TCP context are remembered (using a linked list of pointers) and have their usage counters incremented by one, most likely to 2. When the packets are sent, their usage counter is decremented, most likely to 1. When the ACK is received, the list will be pruned and the packet's counter decremented. Finally, the packet is recycled when the counter reaches zero.

Another traditional approach supported by our platform when the flow is *not* modified (such as for analysis purposes) is to forward the packet even if it may be part of malicious content. The ACK is sent by the destination as expected, and we send a RST to both sides of the connection when the connection needs to be closed when a further packet is received, *e.g.* when a pattern has been matched by an IDS. However some protocols operating on top of TCP may already have handled the payload and most of the attack may have been executed or unfiltered content could have been displayed. A realistic example of such approach is the case of HTTP file downloads, where *generally* the file will be dropped by the browser if the connection is reset before the last packet is received.

C. TCP Flow resizing

Many applications need to modify the content of a stream. For the web, examples include rewriting HTTP traffic to replace URLs with a corresponding CDN-based URL, ad-insertion and removal, and potential new uses enabled by the lightweight in-the-middle stack we propose, such as per-user targeted HTTP page modification or a proxy cache that includes image content in the page itself. Additionally, pages could be translated on the fly to the requesting user's language. Other non-web uses include TLS termination, protocol translation, video transcoding and audio enhancement. Therefore, the ability to modifying a stream is an essential NF and this requires stalling (delaying) and re-ordering packets.

When a VNF removes data from or adds data in a TCP stream, the sequence number must be modified lest the destination think the data has been lost or is a duplicate. However, when the destination sends the corresponding ACK, the value must be also mapped to the correct value. For example. Figure 11 shows an example in which the following modifications are made to the original flow by components in the TCP context: 2 bytes are removed at position 2, removing *CD* from the stream, and 3 bytes are inserted at the position 5, adding *XYZ* in the stream. We keep a list of modifications, represented by a position (the position at which the modification occurred) and an offset that corresponds to the number of bytes modified. This offset is negative if bytes are removed and positive if bytes are added. Note that cannot maintain a cumulative offset because an ACK may induce a retransmission of previously sent data before the cumulative offset and we would not know which bytes had been removed or added from the original flow. Once an ACK arrives for the data, then the list can be pruned.

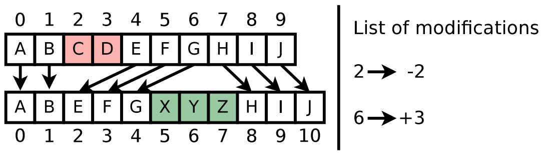


Figure 11: Example of the mapping between an original flow and the corresponding modified flow. Red cells correspond to data removed from the original flow and green cells correspond to data added in the modified flow.

It is worth mentioning that the book-keeping is only instantiated if one of the VNF components specify it may resize a flow. When the size of the flow is unchanged, there is no need for book-keeping. From a practical point of view, adding too many bytes will eventually break congestion algorithms on a host, or fill-up the receive window. Fortunately, many applications only modify slightly the size of the TCP flow, such as encryption and TLS termination, content filtering, split proxies, . . . ; however, evaluation of how many bytes can be removed or added without negatively impacting congestion control is left for future work.

D. Matching both directions of the flow

Within MiddleClick some data must be shared between both directions of a session, such as TCP sessions. One option would be to re-parsing the classification tree but with inverted destination and port, but if both directions are not served by the same core, then the flow table would need a complex locking solution. Alternatively, a symmetric hash key[29] could be used, but if a VNF is a NAT, then the return direction will have a different tuple and is unlikely to hash to the same core. As shown in mOS[18], one approach is to loop through multiple NAT source ports and compute the 4-tuple hash in software in the same way the hardware would until the hash leads to the current core as RSS would. Unfortunately, on average as many hashes as cores have to be computed in software, with an unbounded worst case. The capabilities of recent NICs might be used to allocate NAT ports in such a way that the return packets will be served by the correct core, by using specific receive filters to assign chosen ranges of ports to cores. However, not all NICs are capable of masked classification, and NICs often lack support for newer protocols. Therefore, we developed an efficient, lockless solution that allows each side of the connection to be served by different cores while proposing an efficient way to reconcile common data.

When a new TCP stream is seen (a SYN packet), the TCP context entry allocates a new common data structure for data common to both directions using a per-thread memory pool. The pointer to this common data is saved in the FCB's session scratchpad. The TCP context entry component adds a pointer to this common space in a thread-safe hash table where each bucket is protected by a *Readers-Writer (RW)* lock. The RW lock is based on a usage counter. The counter will be negative while there is one writer and positive when there are readers in the bucket list. When the other direction sees the corresponding stream, it looks for the inverted 4-tuple in the hash table, adds the pointer to the common data to its own session scratchpad. As the session scratchpad is returned with each packet of the stream, the hash table is never read again for this session, thus contention between cores is minimal as it only occurs if two sessions are created at the same time and in the same bucket. We verified this in experiments with millions of new connections per second (see Section V).

E. Context implementation

Entry and exit of contexts are done through pairs of IN and OUT elements, such as `TCPIn` and `TCPOut`. IN elements traverse the graph to find their corresponding OUT elements and announce themselves to them so one can access the other. Along the way, they announce themselves to the next flow elements, including other "downstream" IN elements. A VNF element will have a *previous context* pointer pointing to the last IN element, which has its own pointer to the previous one, *etc.* up to the first IN element.

Each context entry element implements the set of known requests described in Section IV-A. Access to the context is done through function calls to the *previous context* element. The previous context element handles its protocol specifics and passes the request to the previous one and so on, until the first entry element (`IPIn` in most cases) finds no other context entry. Combined with the context link, the usually complex Click manual wiring is actually very minimal as shown in Figure 12. When the flow classifier traverse the graph and resolves the $\sim>$ context links, it remembers the last IN element that was traversed.

The context link already allows spawning a classification rule for a traffic class & session definition according to the elements on the right of the $\sim>$ symbol. We implemented a way to automatically define protocol classification when IN elements are inserted. For instance, the IPIn element could not expose a traffic rule such as *ethertype 0800* because the IP header may be encapsulated by another protocol, *e.g.* a GTP tunnel, rather than an Ethernet frame. Therefore, the last IN element is interrogated to spawn a traffic class rule based on the next IN element. Thus the IPIn element can spawn a *ip proto/06* traffic class rule when it is followed by a TCPIn element. While if TCPIn was preceded by an ATMIn element to implement TCP-over-ATM, a different traffic class rule would be used if ATMIn was programmed to return a rule when interrogated about how to classify fora TCP context. By default, the `FlowManager` act as a first "EthernetIn", returning a rule (*i.e. eth type/0800*) when a context link is inserted before an IPIn element. Therefore, the example of Figure 12 will have a flow table with one rule "eth type/0800 ip proto/06 ip src/ffffffff ip dst/ffffffff udp src port/ffff udp dst port/ffff" that will drop non-IP/UDP packets and duplicate itself and the session scratchpad for each UDP 4-tuple.

In this way, we retain Click's modularity but have a more streamlined default case. If the user wants to exercise finer control of classification using a more refined `Classifier`, the context links can be omitted. Using an `IPIn` context entry after a `UDPIn` supports IP in UDP encapsulation – out of the box. A pattern matcher can be used on top of an application layer context, that would be put after *e.g.* the `HTTPIn` block as many protocols are now deployed on top of HTTP - with HTTP accounting for a majority of the internet traffic[30]. The same pattern matcher could be used on top of `QUICIn` (a still to-be-implemented context block to support QUIC[10] – itself on top of `UDPIn`.

```
FromDevice(...) -> FlowManager
  ~> IPIn
    ~> UDPIn(TIMEOUT 300)
      ~> WordMatcher(ATTACK, MODE REMOVE)
    -> UDPOut
  -> IPOut
-> ToDevice(1);
```

Figure 12: Configuration for a transparent VNF that removes the word "ATTACK" from the UDP flow, even across packets. As UDP does not implement connection semantics, the UDPIn element sets the session timeout to some value, here 300 s.

F. Socket-like abstraction

As the elements of a FastClick configuration manipulate batches of packets, it is inconvenient for a developer to perform some operations, such as searching a specific pattern in the flow. To address this, we provide an iterator-like object reminiscent of FlowOS[31]. Initially, the iterator points to the first byte of the current level in the batch of packets.

When calling `iterator++`, it may cross a packet border seamlessly according to the current context. If the processing function returns with the iterator in the middle of a packet, all packets *before* the iterator will be forwarded to the rest of the graph. A request for more data will be propagated through the context for packets after the iterator. If the iterator was at the last packet, all packets are forwarded. This abstraction allows implementation in only a few line of new VNF components that would appear complex. Additionally, we provide multiple generic VNF elements that act on the current context, such as regular expression matcher, packet counter, load balancer, and an accelerated NAT.

Figure 13 shows the code for a simple intrusion prevention system (IPS) using the byte iterator. Each byte of the payload is fed to a deterministic finite automaton (DFA) (on line 8) with its state kept inside the FCB. If the DFA finds a match, the connection is closed (line 11). If the iterator is at the end of the available payload, but the DFA is in the middle of a potential match (line 18), the iterator will be left at the last point known to be safe in the flow (saved in line 14). Packets up to that point will be processed, others will be kept in the FCB and a *request for more data* will be made. In contrast to most IPS, such as Snort[32] or Suricata[33], this IPS is not subject to eviction attacks, as the matching state is kept between windows. Moreover, packet buffering is minimal – as only the data part of a potential match will be kept.

```
 1  int FlowIDSMatcher::process_data(fcb_ids* fcb,
 2                         FlowBufferContentIter& iterator) {
 3  //Position in the flow where there is no pattern for sure
 4      FlowBufferContentIter good_packets = iterator;
 5
 6      while (iterator) {
 7          unsigned char c = *iterator;
 8          _program.next(c, fcb->state); //Advance the DFA
 9          if (unlikely(fcb->state == SimpleDFA::MATCHED)) {
10              _matched++;
11              return CLOSE;
12          } else if (fcb->state == 0) {
13              //No possible match up to this point
14              good_packets = iterator;
15          }
16          ++iterator;
17      }
18      if (fcb->state != 0) {
19  //No more data, but in the middle of a potential match
20          iterator = ++good_packets;
21      }
22      return 0;
23  }
```

Figure 13: Code for a DFA-based IPS that is not subject to eviction. It only buffers data when the payload is missing the next bytes in a state that might detect a pattern.

V. Performance Evaluation

In this section we discuss MiddleClick' performance under various test cases. We start by demonstrating the performance of the flow manager for a single stateful function in §V-B. We then study the performance of the stream abstraction built on top of the flow manager in §V-C. In §V-D we analyze the benefit of avoiding multiple re-classifications and multiple stacks. In §V-E, we study how MiddleClick behaves with a chain of light and heavy NFs to assess if is fit-for-purpose in a realistic environment. We finish, in §V-F, with a quantitative assessment of the benefits for a developer.

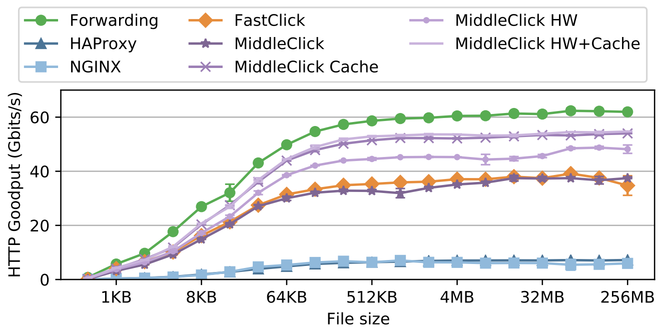


Figure 14: HTTP Goodput when downloading many objects of various size through a load-balancer using 2048 concurrent connections. The proxy ran on a single CPU core.

A. Testbed

Tests were done using a *client* machine, with 192 GB of RAM and two 16-core Xeon Gold 6246R @ 3.4 GHz processors that generate HTTP requests using WRK[34]. These requests are sent towards a *Device Under Test (DUT)* with 192 GB of RAM and an 18-core Xeon Gold 6140 processor, fixed at its nominal frequency of 2.3 GHz using the Linux Kernel *cpupower* facility. The *DUT*'s first processor is isolated with *isolcpus* for running the VNFs for the tests, while the second processor run the experiment scripts. The *DUT* running the various solutions being tested forwards packets to the *sink*, a machine similar to the *client* that runs the NGINX[35] web server on all 32 cores. All machines are interconnected using Mellanox ConnectX 5 2*100GbE NICs. All test scripts used NPF[36] to run each test 10 times for 20 s (after a 2 s period to avoid measuring cold-start) and output a graph of the average and standard deviation for the selected metric. All of the test scripts are available online[1].

B. Performance of the flow manager for a single function

The goal of the testing described in this section is to ensure that decoupling state management from the VNF is beneficial – even for a single function. We evaluate the session classification performance of our system against state-of-the art solutions, using a TCP load-balancing reverse proxy. The proxy balances in a round-robin way the upcoming HTTP connections from the *clients* to multiple IP addresses assigned to the *sink*, ensuring that packets of the same session go to the same IP destination. This is a typical datacenter application. Redirections is done by changing the destination IP address of the request. The requests traversing the proxy are NATed, to ensure that the packets go back through the box such that the source IP address sends them to the original destination.

We compared a load-balancer implemented using Middle-Click to FastClick, HAProxy[37] in TCP mode and the NGINX[35] reverse proxy. For the laters, we tuned the Linux TCP stack to allow enough connections and enabled various common optimizations, such as using SO_REUSEPORT. While OpenBox, E2, and Microboxes[8] are probably closest

[1]https://github.com/tbarbette/middleclick-experiments/

to MiddleClick, their implementations are not fully available[2] In this experiment, the VNFs require only a partial TCP state reconstruction (similar to FastClick's one) and not a full (in-the-middle) TCP stack. The script request files ranging in size from 0 kbyte to 256 MB. HTTP keep-alive is disabled, so each HTTP request is made using a new TCP connection. Figure 14 shows the goodput as a function of file size and implementation. The "Forwarding" line shows the performance of the testbed when the DUT is only forwarding packets using a forwarding configuration in FastClick (as an upper bound). The MiddleClick proxy has similar performance to the FastClick one. Indeed, the advantage of the minimization of the classification is balanced by the flexibility of the flow manager. Additionally, the offloading features were enabled one by one to decouple the individual improvements. Activating the cache (as described in Section II-F) improved performances up to 40% over FastClick. Activating the traffic class offloading leads to a lower improvement, as the flow classification is the heavy work in this testcase. All MiddleClick solutions were an order of magnitude faster than HAProxy and NGINX, because of their unnecessary TCP termination and inefficient network I/O.

C. *How fast is the stream abstraction?*

Figure 15 exposes the cost of using heavier context features compared to other software doing similar modifications. The simplest context feature is a TCP state tracking function. This allows context components to react to TCP state changes (*e.g.* connection established, closed, ...) and be called when some data is available, but not reordered. It also calls a function on each block of payload available, but in this experiment, does nothing with it. mOS[18] is one of the rare fully available NFV frameworks and it proposes an implementation that we compare against. We made an mOS application which uses mOS's flow monitoring, but also performing no action. As can be seen in Figure 15, MiddleClick performs 2.3 times better than mOS for state analysis. mOS reproduces the state of both sides of the connection, therefore spending a lot of time in timeout management. Next, we compared multiple solutions that fully reorder and reconstruct the stream, allowing the VNF operator to act on the bytestream. Utilizing the full

[2]Specifically, E2 and Microboxes are unavailable. OpenBox[5] is partially available, but its per-flow metadata mentioned in §4.5.3 is not present, therefore we would end up using the same NAT elements as FastClick, only using slower I/O.

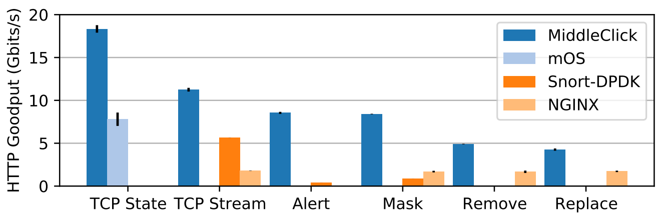


Figure 15: Throughput for various action on streams of 8K HTTP requests

TCP in-the-middle context block of MiddleClick for stream processing, while adding reordering and more book-keeping incurs a 54% performance hit. We also ran Snort IDS version 2.9.11.1 with a DPDK Data Acquisition Module (DAQ) for I/O[38], configured without any rule or inspection preprocessor other than the TCP stream engine (*i.e.* Stream5), hence doing no matching, only TCP reconstruction. We compared this solution to NGINX in proxy mode, as it similarly passes bytes untouched. Our system performs at least twice as fast a the other engines.

The alert function builds on top of the TCP context to detect the presence of a single word ("ATTACK" in the experiment) in the stream and raises an alert when found. In MiddleClick, we use the *WordMatcher* element that uses the chunk payload iterator presented in Section IV-A to run an AVX2-based string matcher taken from [39]. We also integrated HyperScan[40], a full-featured pattern matching library for IDS, that allows the developer to register the state of the DFA in its own memory location. We therefore allocate and keep the DFA directly in the FCB, avoiding eviction attacks in contrast to window-based implementations such as Snort – as the matching algorithm always continues in the state it was left in. The *HyperScan* element incurs only a performance drop of 2.6% when compared to the AVX2 implementation of the alert function; therefore, it was not shown in Figure 15. Snort configured with the HTTP preprocessor and a single similar matching rule performs 20 times slower than the other configurations.

The mask function replaces the word with some stars (*) characters. NGINX's substitution module can be used to implement the same replacement function, but its througput is 4.6 times lower than MiddleClick. Snort can perform exact-length replacement too but is unable to add or remove content. Stream size modification is showcased by the remove function that will remove the word, while the replacement function replaces the string by a longer HTML string, explaining in color that some harmful content has been removed. Those two functions modify the payload and utilize the TCP sequence tracking explained in Section IV-A. Still, MiddleClick performs more than 2.2 times better than NGINX for both removal and replacement. After multiple fixes, ClickNF[41] could only achieves 128 Mbps in a configuration similar to TCP Stream and was kept out of Figure 15. This showcases well the difficulty of building a full TCP stack compliant with the standards and shows the relevance of our lighter approach.

D. *What is the benefit of avoiding multiple re-classifications?*

Section I already showed in Figure 1 the impact of adding multiple VNFs in the chain without consolidating both the classification and the stack. In this experiment, a flow monitoring module is repeated, from zero to five times. Each blocks require 24 bytes for tracking, that will be accommodated in a single flow table, for the "combined", or in repeated flow managers, one per NF for the "Independent" lines. The combined table allows for offloading, which gives an improvement of around 1.5X as shown by the "Accelerated" lines. In case of the TCP stack, we use the monitoring "TCPIn" element presented above. Figure 16 shows the latency of packets for the same experiment. The latency is computed as the time between the reception and the transmission of the packet. With 5 NFs, the latency is 10 times lower when combining the TCP stack (using hardware acceleration) compared to using independent NFs. The flow completion time (average file download time) follows the same trend, divided by 4 when using the combined version.

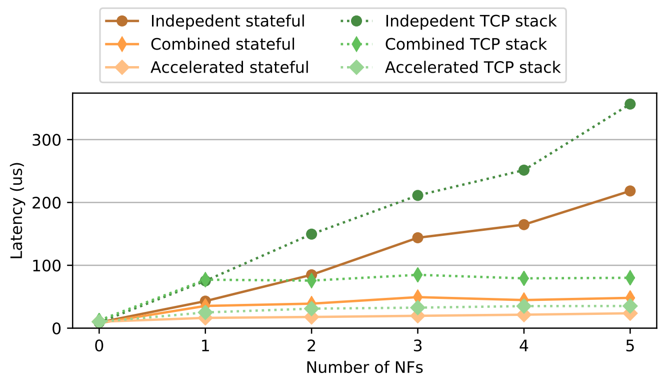


Figure 16: Latency of a chain of stateful NFs, and TCP-based NFs.

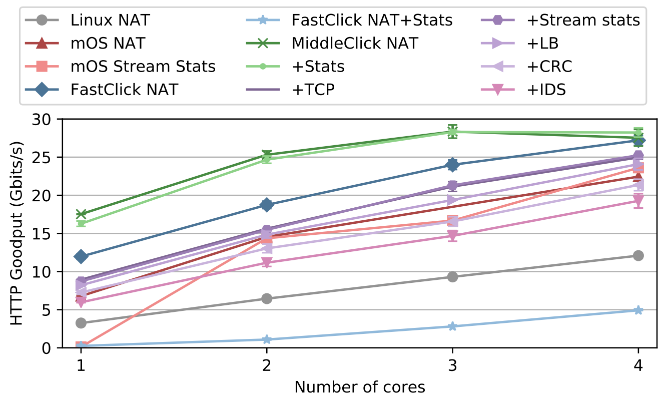
E. How does MiddleClick perform with a chain of light and heavy NFs?

Figure 17 shows the performance of several service chains running on 1 to 4 cores for 8K HTTP requests. First, we compare a chain of a single function, a NAT, using MiddleClick, FastClick, mOS, and the Linux NAT. As established in Section V-B, MiddleClick outperforms both FastClick and mOS, and the Linux NAT too. Additionally, mOS faces serious limitations as its model prevents using L2/L3 features such as learning bridge or ARP queries, therefore it only supports a bump-in-the-wire configuration (mandatory for a NAT) when used in inline[3]. In its current state, mOS cannot be used either in a service chain when using DPDK, which is mandatory to have acceptable userlevel performance (a shortcoming solved by Microboxes[8] which is still not available at the time of writing). The slowness of mOS can be explained by it much heavier TCP stack – most of which is unnecessary for a NAT.

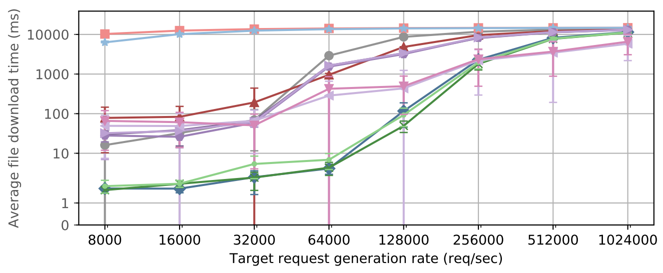
When adding to the FastClick and MiddleClick solutions a statistics VNF to simply count all bytes per-sessions, FastClick performance drop considerably because of the second session classification. In contrast, MiddleClick is barley slowed down by the second function, as adding this function only extends the flow table by a few bytes. To further highlight the advantage of using MiddleClick, we introduce a few more functionalities to the chain. When adding TCP reconstruction to the MiddleClick chain, performance decreases to $7.6\,\text{Gbps}$, due to the cost of TCP state management and reordering of TCP packets. Adding VNFs for flow statistics (per-session byte count but only for the useful payload), a load balancer, and a computation of a checksum (similar to the checksum computation element presented in Figure 10) has very little impact on performance as they all have their own space in the FCB as automatically extended to fit all VNFs of the service chain by MiddleClick. *i.e.* without any manual tuning. In comparison, mOS with only stream statistics performs worst than the MiddleClick chain running 5 more functions. We note with a single core, the mOS application enters a receive livelock situation, barely handling any packets, while the NAT does not work with 3 cores as the RSS computation to find a port that will lead to packets returning to the same core is broken for our NIC.

Figure 17b shows the average flow completion latency for 8K requests for the same experiments, under increasing offered load. For a single function (NAT), MiddleClick performs similarly to FastClick. However, once the price of the classification has been paid, adding more functions does **not** increase the latency very much, even when executing the heavier DPI function.

In this example, the FCB layout for the full chain is 282 bytes for the forward path and 278 for the reverse path. The full TCP tracking accounts for 112 bytes, static classification (marking which path should be selected according to the traffic class) for around 24 bytes (depending on the path), and the 7 NFs share the rest more or less equally.



(a) Throughput using one to four CPU cores running at $2.3\,\text{GHz}$



(b) Latency for increasing offered request rate, using 1 CPU

Figure 17: MiddleClick advantage when chaining multiple VNFs compared to two functionally identical FastClick setups for NAT and stream statistics, Linux NAT, mOS NAT and mOS stream statistics.

F. What are the quantitative benefits for a developer?

Decoupling the state management from the network function itself removes a lot of burden for developers. As an

[3] *i.e.* when packets pass through the box, as opposed to a monitoring mode where the box receives copies of packets to perform some analysis on but does not pass them through

example, the datapath code for the FastClick NAT contains more than 400 lines of code, handling classification, ageing, state tracking, etc. The code of the UDP path is mostly copy-pasted from the TCP part, forcing the developer to maintain both code bases. In comparison, the MiddleClick datapath code is only around 20 lines. Figures 7, 10 and 13 already show examples of MiddleClick pseudo-code. As stated in Section II, the flow manager can utilize multiple classification algorithms – selecting them according to the characteristics of the tree and the performance of the nodes. The classification algorithm accounts for more than 6000 lines of code allowing to fit many use cases. During the development phase, we built many components such as a pattern matcher, data replacement blocks, statistic tracker, ... However, thanks to the context system, one can re-use those blocks on top of existing protocols, without necessarily writing C++. Avoiding a full TCP reconstruction also makes MiddleClick's context elements much more maintainable. We did not need to include any congestion control mechanism, as we let both parties handle re-transmission, but only translate SEQs and ACKs sent in the session. Additionally, the session manager would be compatible with an implementation of a full TCP stack, *e.g.* to implement a web server; hence, naturally merging such a server with firewalls, flow trackers, or other stateful NFs. The TCP stack elements constitue around 3000 lines of code.

VI. Related works

In addition to the state of the art discussed in the previous sections, this section discusses some related prior work.

A. Operating Systems

One could argue that efficient service chaining is the problem of the Operating System (OS), but generic OSs have proven to be far too slow for raw I/O [24] and middlebox implementations[18]. Prior work, such as IX[14] or Arrakis[13], modified or re-designed the OS to improve performance and increase isolation between middlebox components. Mirage[42], NetVM[43], and ClickOS[44] try to make the components themselves faster using fast-deployable and efficient unikernels or light VMs, but all three lack support for cooperation between VNF components inside the OS and cooperation between multiple instances of the dataplane.

B. Userlevel TCP stacks

CliMB[16] and ClickNF[41] introduce a full event-driven TCP-stack inside the Click Modular Router[25] which has proven to be a good platform for middlebox and NFV implementations and helps to bridge the gaps, along with other user-space stacks[15], [17], [45], towards a full userlevel service chain by bypassing the kernel. StackMap[46] observed most userlevel TCP stacks were actually not maintained and preferred to keep the kernel stack, which is well maintained and benefits from all protocols extension, but to use the kernel bypass techniques of netmap[12] to accelerate the fast path I/O. Yet, they are not in the scope of cooperation between instances of those stacks. Those userlevel stacks all receive and send raw packets, that would need full re-classification and protocol specific management for each VNF along the chain while a full TCP stack is often not even needed.

C. NFV Dataplanes

xOMB[47] provides better programmability by decoupling middlebox functionalities, allowing simple pipelines of middleboxes functions to. CoMb[2] explores consolidation of middleboxes for better resource management and reduces the need for over-provisioning. As in our design, CoMb proposes memory sharing with older applications that cannot be modified to take advantage of the facilities provided. But both xOMB and CoMb lack consolidation inside the low-level components, *e.g.* passing flows between applications and providing support to build functions on top of a flow abstraction that can be unified, likely leading to limited throughput or higher latency when the service chain is long. xOMB relies on heavy message buffers passed between components and does not achieve high-speed. For a function similar to the one evaluated in V-B their performance is less than 1% of ours. This difference is unlikely to be solely due to their older testbed. Moreover, xOMB does not provide a way to decompose and recompose multiple VNFs to consolidate the components of multiple middleboxes together, thus their performance is likely to drop even further with more VNFs. As xOMB is not publicly available, it is impossible to evaluate what it current performance might be.

NFP[48] automatically builds a parallel graph based on the order and dependency between VNFs and takes care of efficiently copying packets and merging them back. It unifies the static service chain classification, but not the intra-VNF classification nor the dynamic flow tables & flow abstraction. While we utilize a run-to-completion model, their pipelining technique could be used in conjunction with our proposal.

OpenBox[5] consolidates low-level functions across the service chain, using a controller to manage Click-based low-level components (although this is only a proof-of-concept and they allow for other implementations including hardware ones), extracting packet header classification to a unified parser but lacks stream abstraction and re-use of session parsing. The OpenBox protocol defines a per-session key/value store but this per-flow metadata is only conceptual, and the authors do not explain how they would build and manage a data structure (such as the FCBs) to handle millions of flows per second, which is a major contribution of our work. Nor do they address recycling of entries in the store or how to handle possibly multiple levels of sessions (per destination, per IP pair, per TCP session, ...). In contrast, we address all these issues.

D. Flow tempering

Both NetBricks[49] and mOS[18] implement a TCP stack with similar abstractions, but do not provide any factorization and acceleration of the full service chain. Their flow abstraction is limited to a less flexible window system and do not provide a generic non-TCP stream abstraction nor the session scratchpad facility, likely losing a lot of performances when the running many different VNFs. mOS offers a nice, high-performance event-based TCP stack. Most of the events they offer are available through MiddleClick context with the advantage that MiddleClick can work for any protocol. More importantly, neither proposes a way to modify flows (*i.e.* more than simple rewriting) without fully terminating the connection. In contrast, we lets endpoints handle the TCP semantics, making it future proof whereas most of the state of the art TCP stacks only support part of the TCP protocol, strip options, or represent so much maintenance work that they are already abandoned. TCP Splice[50] avoids full termination by mapping sequence and acknowledgement numbers with a constant offset, but does not allow modifications. In contrast with this paper, they do not focus on service chaining, use shared flow tables, or describe classifications problems. Moreover, our proposal is protocol agnostic and allows a context-based system that support *e.g.* HTTP modifications on top of TCP modifications. Microboxes[8] essentially extends mOS to offer a publish/subscribe API to either a monitoring socket or a full stack. In contrast, we offer an in between solution to modify flows *without* running two full TCP stacks. Moreover, MiddleClick combines classifications and components as a lower-level combination to achieve very high-speed and enables hardware offloading of the classification.

VII. Conclusion

In this work, we have developed a high-speed framework to build service chains of VNFs. Our system has better throughput and lower latency than other approaches, thanks to the avoidance of multiple reclassification of packets as they pass through the various VNFs in a chain.

Our framework also eases the handling of per-flow *and* per-session state. Thus an VNF developer can specify, in a flexible way, which flows or sessions the VNF is interested in, and the size of the state the VNF needs for each traffic class or session. Then, the system automatically provides and manages the associated per-session storage, which is directly available to the VNFs.

Finally, our framework exposes simple stream abstractions, providing easy inspection and modification of flow content at any protocol level. Thus the developer simply focuses on the VNF functionality at the desired protocol level, while the framework adjusts the lower-level protocol headers as needed, even creating new packets if necessary. Our framework can also act as a man-in-the-middle for stateful protocols such as TCP, greatly simplifying high-level VNF development, while avoiding the overhead of a full TCP stack.

Our open-source implementation of MiddleClick[4], shows significant performance improvements over traditional approaches on a few test cases. We believe the ease of use demonstrated through the various examples in this paper show its potential for broad adoption.

Acknowledgment

This work has been funded by the *Fond National de la Recherche Scientifique (*FNRS*)* through the PDR ePi project and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 770889). The authors would like to thank Gerald Q. Maguire Jr. for his help reviewing the paper, as well as the anonymous reviewers of ToN for their valuable input that greatly improved the quality of this paper.

[4]Available at: https://github.com/tbarbette/fastclick/tree/middleclick

References

[1] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," in *Proc. ACM SIGCOMM*, August 2012.

[2] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proc. USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[3] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An untold story of middleboxes in cellular networks," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM'11. New York, NY, USA: ACM, 2011, pp. 374–385. [Online]. Available: http://doi.acm.org/10.1145/2018436.2018479

[4] G. P. Katsikas, M. Enguehard, M. Kuźniar, G. Q. Maguire Jr, and D. Kostić, "Snf: synthesizing high performance nfv service chains," *PeerJ Computer Science*, vol. 2, p. e98, 2016.

[5] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: a software-defined framework for developing, deploying, and managing network functions," in *Proc. ACM SIGCOMM*, 2016.

[6] S. R. Chowdhury, H. Bian, T. Bai, R. Boutaba *et al.*, "μnf: A disaggregated packet processing architecture," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 342–350.

[7] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for nfv applications," in *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[8] G. Liu, Y. Ren, M. Yurchenko, K. Ramakrishnan, and T. Wood, "Microboxes: high performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 504–517.

[9] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan, "Tcp fast open," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. ACM, 2011, p. 21.

[10] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 183–196.

[11] L. Foundation, "Data plane development kit (DPDK)," 2015, http://www.dpdk.org. [Online]. Available: http://www.dpdk.org

[12] L. Rizzo, "netmap: A novel framework for fast packet i/o." in *Proc. USENIX Annual Technical Conference (ATC)*, 2012. [Online]. Available: http://info.iet.unipi.it/~luigi/netmap/

[13] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2014. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[14] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "Ix: A protected dataplane operating system for high throughput and low latency," in *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2014. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[15] I. Marinos, R. N. Watson, and M. Handley, "Network stack specialization for performance," in *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2013.

[16] R. Laufer, M. Gallo, D. Perino, and A. Nandugudi, "Climb: enabling network function composition with click middleboxes," *Proc. ACM SIGCOMM*, 2016.

[17] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: a highly scalable user-level tcp stack for multicore systems." in *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[18] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "mos: A reusable networking stack for flow monitoring middleboxes." in *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[19] B. Vamanan, G. Voskuilen, and T. Vijaykumar, "Efficuts: optimizing packet classification for memory and throughput," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 207–218, 2011.

[20] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. ACM SIGCOMM*, 2003.

[21] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and

implementation of open vswitch." in *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[22] Mellanox, "Mellanox ethernet cards," http://www.mellanox.com/page/ethernet_cards_overview. [Online]. Available: http://www.mellanox.com/page/ethernet_cards_overview

[23] V. Tanyingyong, M. Hidell, and P. Sjödin, "Using hardware classification to improve pc-based openflow switching," in *Proc. IEEE High Performance Switching and Routing (HPSR)*, 2011.

[24] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Proc. ACM/IEEE Symposium on Architectures for networking and communications systems (ANCS)*, May 2015.

[25] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000. [Online]. Available: http://doi.acm.org/10.1145/354871.354874

[26] S. Garzarella, G. Lettieri, and L. Rizzo, "Virtual device passthrough for high speed vm networking," in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems*. IEEE Computer Society, 2015, pp. 99–110.

[27] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Static checking for stateful networks," in *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization*. ACM, 2013, pp. 31–36.

[28] J. Novak, S. Sturges, and I. Sourcefire, "Target-based tcp stream reassembly," *Aug*, vol. 3, pp. 1–23, 2007.

[29] S. Woo and K. Park, "Scalable tcp session monitoring with symmetric receive-side scaling," *KAIST, Daejeon, Korea, Tech. Rep*, 2012.

[30] CAIDA, "The caida ucsd passive nyc dataset." [Online]. Available: http://www.caida.org/data/passive/trace_stats/

[31] M. Bezahaf, A. Alim, and L. Mathy, "FlowOS: A flow-based platform for middleboxes," in *Proc. ACM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2013. [Online]. Available: http://doi.acm.org/10.1145/2535828.2535836

[32] Cisco, "Snort - Network Intrusion Detection & Prevention System," 2017, http://www.snort.org/. [Online]. Available: http://www.snort.org/

[33] Open Information Security Foundation, "Suricata | Open source IDS / IPS / NSM engine," 2017, https://suricata-ids.org/. [Online]. Available: https://suricata-ids.org/

[34] W. Glozer, "Wrk," https://github.com/wg/wrk. [Online]. Available: https://github.com/wg/wrk

[35] NGINX Inc., "NGINX | High performance load balancer, web server & reverse proxy," 2017, https://www.nginx.com/. [Online]. Available: https://www.nginx.com/

[36] T. Barbette, "Npf," 2017, https://github.com/tbarbette/npf. [Online]. Available: https://github.com/tbarbette/npf

[37] W. Tarreau, "HAProxy: The reliable, high performance TCP/HTTP load balancer," 2017, http://www.haproxy.org/. [Online]. Available: http://www.haproxy.org/

[38] J. Iurman *et al.*, "Master thesis: Fast service chaining," 2017.

[39] W. Muła, "Simd-friendly algorithms for substring searching." [Online]. Available: http://0x80.pl/articles/simd-strfind.html#sse-avx2

[40] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu, "Hyperscan: a fast multi-pattern regex matcher for modern cpus," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 631–648.

[41] M. Gallo and R. Laufer, "Clicknf: a modular stack for custom network functions," in *2018 USENIX Annual Technical Conference (ATC)*, 2018, pp. 745–757.

[42] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451167

[43] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.

[44] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proc. USENIX Networked Systems Design and Implementation (NSDI)*, April 2014.

[45] Solarflare, "Openonload," http://www.openonload.org/. [Online]. Available: http://www.openonload.org/

[46] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "Stackmap: Low-latency networking with the os stack and dedicated nics." in *USENIX Annual Technical Conference*, 2016, pp. 43–56.

[47] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xomb: extensible open middleboxes with commodity servers," in *Proc. ACM/IEEE symposium on Architectures for networking and communications systems (ANCS)*, 2012.

[48] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "Nfp: Enabling network function parallelism in nfv," in *Proc. ACM Special Interest Group on Data Communication*. ACM, 2017.

[49] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of nfv." in *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[50] D. A. Maltz and P. Bhagwat, "Tcp splice for application layer proxy performance," *Journal of High Speed Networks*, vol. 8, no. 3, pp. 225–240, 1999.