Building a chain of high-speed VNFs in no time

(Invited Paper)

Tom Barbette, Cyril Soldani, Romain Gaillard and Laurent Mathy

University of Liege
firstname.lastname@uliege.be

***Abstract*—To cope with the growing performance needs of appliances in datacenters or the network edge, current middlebox functionalities such as firewalls, NAT, DPI, content-aware optimizers or load-balancers are often implemented on multiple (perhaps virtual) machines.**

In this work, we design a system able to run a pipeline of VNFs with a high level of parallelism to handle many flows. We provide the user facilities to define the traffic class of interest for the VNF, a definition of session to group the packets such as the TCP 4-tuples, and the amount of space per sessions. The system will then synthesize the classification and build a unique, efficient flow table. We build an abstract view of flows and use it to implement support for seamless inspection and modification of the content of any flow (such as TCP or HTTP), automatically reflecting a consistent view, across layers, of flows modified on-the-fly.

Our prototype gives rise to a user-space software NFV dataplane enabling easy implementation of middlebox functionalities, as well as the deployment of complex scenarios. Our prototype implementation is able to handle our testbed limit of ~ 34 Gbps of HTTP requests (for 8-KB files) through a service chain of multiples stateful VNFs, on a single Xeon core.

I. Introduction

According to [1] and [2], there are roughly as many middleboxes as routers in enterprise networks. A myriad of middleboxes is also deployed in mobile networks [3]. These middleboxes are there for good reasons, as they provide network security as well as performance enhancements. However, the various middleboxes often complicate network management and operations, as well as slow down innovation, because they are often independent systems working like complete black boxes.

Network Function Virtualization (NFV) is an approach where middlebox functionality is implemented in software on commodity hardware, not only rendering the implementation of middleboxes less opaque, but also supporting the outsourcing of middlebox functionality to the cloud. In NFV settings, just like in more traditional networks, network packets must often be treated by several middleboxes, which leads to the need of chaining several virtual network functions (VNFs) together.

As exemplified in figure 1, such chains, if built in a naive way, can easily lead to redundant work, such as packet classification being performed in each VNF of the chain. And once a packet has been classified, VNFs will also often perform another (redundant) lookup to retrieve the state associated with the corresponding flow, before the packet can be processed. Those redundant operations, of course, reduce performance.

We therefore propose MiddleClick, a software flow processing plateform that explicitly recognizes NFV chains and automatically provides consolidation to avoid repeated (redundant) work. More precisely, MiddleClick analyzes the various classification needs of the VNFs in the chain, and synthesizes a single, non-redundant classifier (see figure 2). Also, once classified, packets are associated with a Flow Control Block (FCB), which contains the aggregate state associated with the flow, by each VNF. By passing the packets with a reference to their FCB, the VNFs along the chain can easily, and quickly, retrieve the associated state.

While such consolidation would already prove very useful in terms of performance, MiddleClick goes further in terms of facilitating the development of middlebox functionalities.

Indeed, building packet processing systems is often the realm of specialist "guru" developers, who must juggle the nitty-gritty details of protocol implementations, as well as low-level system programming. To facilitate VNF development, MiddleClick introduces a high-level stream abstraction, which allows the programmer to concentrate on the functionality of the VNF, by exposing iterator-like operators to specific payloads of a flow of packets. MiddleClick then liberates the programmer from the consequences of stream modifications, by automatically rippling the effect of those changes to all protocol layers in the system. In other words, a programmer working, for instance, on a VNF that modifies web pages no longer needs to understand the details of the TCP/IP protocol stack. While most transmission channels are built on top of HTTPS nowadays, datacenter owners usually decrypt the traffic at the datacenter entry, exchanging unencrypted data inside the datacenter itself. Middleboxes can therefore inspect and potentially modify unencrypted streams. Modifications of a stream is done without terminating the connection by applying surgical modifications to the packets, leading to a very efficient flow modification. This enables future innovations, such as new TCP extensions as the protocol stack only needs to understand how to modify the flow and only implements a few TCP semantics. *I.e.* the stack behaves as transparently as possible, being agnostic to changes in congestion or flow control methods and retransmissions techniques used on any ends of the connection. Support for heavier semantical changes such as TCP FastOpen [4] would only need a few lines of code.

MiddleClick exploits user-space I/O frameworks (DPDK or netmap) so that all VNFs run in user space, freeing the programmer from the more hostile kernel development environment.

The hardware infrastructure where VNFs are deployed is

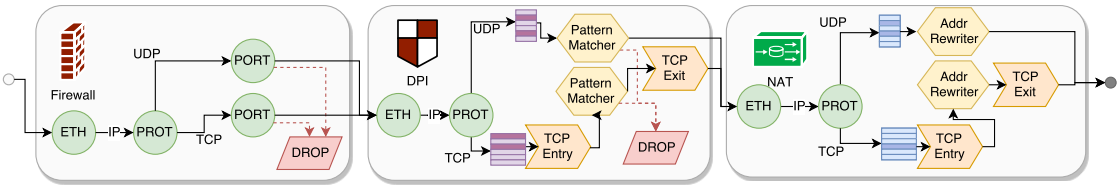

Figure 1: Overall schematic of usual service chain of VNFs broken into basic components.

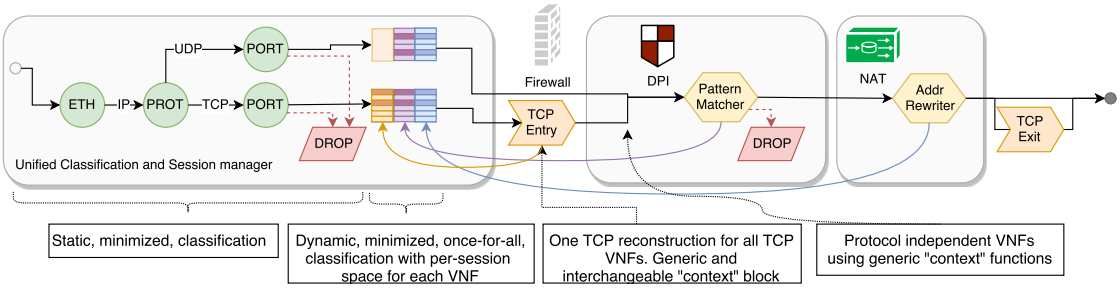

Figure 2: Overall schematic of the architecture we propose to unify the service chain

inherently parallel (*e.g.* multicore servers). Exploiting such parallel compute power is notoriously difficult, so MiddleClick also provides automated parallelism management, in order to further lower the bar to VNF programming.

In section II, we explain how the VNFs among the service chain are combined. From there stems a highly parallelisable and non-redundant stream architecture that can be used as a basis to support multiple protocols, as explained in section III. Section IV reviews the state of the art and our specific contributions. Finally, we evaluate the performance of the prototype in section V, before concluding.

This invited paper provides a high-level view of our proposed solution and does not focus on the implementation details of MiddleClick.

II. Traffic class and session unification

Each VNF component declares the kind of packets it wants (HTTP packets, all TCP packets, ...) and, if needed, the definition of session they want to see (packets grouped by IP pair, by TCP 4 tuples, ...) and an amount of bytes needed per-session, called the per-session scratchpad. The information provided by each component is used to derive a unique classification table that will avoid further re-classification on the packet header. The classification runs before all VNF components, as shown in figure 2. The classification can therefore potentially be offloaded to some specific hardware or use classification functionalities of the NIC.

Consider figure 3 as a simplified example. This system handles ARP requests and replies in a packet-based fashion. It runs a per-session load-balancer for UDP and TCP traffic, but before that passes HTTP (TCP packets with port destination 80) traffic through some HTTP filter (*e.g.* a parental filter or ad-remover). Other TCP traffic is dropped.

The ARP subsystem wants to receive ARP requests and replies, and needs no session management. The HTTP filter needs TCP packets directed to port 80. The load balancer receives any HTTP or UDP traffic. Both the load-balancer and the HTTP filter need a per-5-tuple scratchpad to write some per-session metadata. Note that traffic classes are not limited to

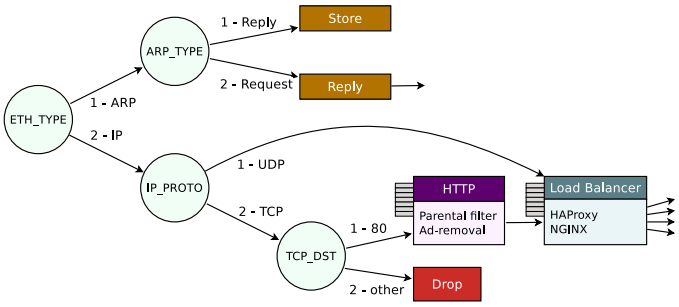

Figure 3: A small system handling ARP packets and applying some processing on HTTP traffic, dropping other TCP streams. It then load-balances UDP and HTTP traffic to some servers.

Rules							Flow Control Block	
Ethertype	ARP Type	Proto	Dport	Sport	Dst	Src	Next hops	Session space
ARP	Request	*	*	*	*	*	1, 2	-
ARP	Reply	*	*	*	*	*	1, 1	-
IP	*	TCP	80	52100	10.0.0.1	89.18.17.216	2, 2, 1	0x1a234579, 0
IP	*	TCP	80	32100	10.0.0.1	18.17.62.29	2, 2, 1	0x9e5cc632, 1
IP	*	TCP	80	52100	10.0.0.17	120.12.17.12	2, 2, 1	0xab38977d, 0
IP	*	TCP	80	PFP	PFP	PFP	2, 2, 1	TCPSession, int
IP	*	UDP	32100	32100	10.0.0.78	129.251.324.118	2, 1	0
IP	*	UDP	PFP	PFP	PFP	PFP	2, 1	int

Table I: Unified flow table

packet fields. *E.g.*, for an ISP a traffic class could be defined as packets from one ingress to a specified egress, the combination of the two determining the processing to apply.

A. Traffic class definition

In figure 3, each circle is one step of the classification which would usually be handled by reading the corresponding packet fields to decide the next step. The first level has 2 outputs, ARP or IP packets that could be identified as 1 and 2. Following this idea, reaching a leaf of the classification path can be considered as following a list of *next hop* numbers. White lines in table I shows the resulting flow table.

To each rule will correspond a *Flow Control Block* (FCB). Instead of classifying packets at all steps of the processing chain as with the circles in figure 3, all the necessary information is included in the FCB, which starts with the list of next hops.

We based our prototype on FastClick [5], an extension of the Click Modular Router [6]. Click allows to pipe simple

```
class ARPResponder : public FlowElement { public:

    ARPResponder() CLICK_COLD;
    ~ARPResponder() CLICK_COLD;

    const char *class_name() const { return "ARPResponder"; }
    const char *port_count() const { return PORTS_1_1X2; }
    const char *processing() const { return PROCESSING_A_AH; }

    FLOW_ELEMENT_DEFINE_SESSION_CONTEXT("20/0001", FLOW_ARP);

    [...]
}
```

Figure 4: Example of session definition for the ARP responder element. Only the coloured text is specific to our proposal

networking elements together to build a more complex network function using a simple language. The elements themselves are written in C++, and implement a few virtual functions to handle packets and events as defined by the common ancestor of all elements, the `Element` class.

Figure 4 shows the changes (in purple) made to the original ARPResponder element of Click to ask for the traffic class of ARP packets that are queries (offset 20, value 0001). The classification rule will be directly used to build the table, while the usage of the *FLOW_ARP* constant will be described in section III-A, in short it will allow the previous context (Ethernet in this case) to spawn a rule for ARP packets, that is the equivalent of adding the traffic class *12/0806* but in a more programmable way, allowing to support tunnelling of protocols inside others if the previous context was not Ethernet.

B. Session definition

We allow each component to describe, on top of flows, the sessions they want to see, and an amount of space they need per-session to write their metadata, the scratchpad. That space will be assigned in the FCB session space, as in table I. To define sessions, rules allow special header wildcards that are *Populate From Packet (PFP)*. The PFP entries mean that the rule must be duplicated with the exact values of the fields when the rule is matched. The difference between defining a rule such as $proto = TCP; dstport = *$ and $proto = TCP; dstport = PFP$ is that the second one will lead to one session and therefore one scratchpad per TCP destination port.

Coloured lines in table I show the flow table after receiving multiple packets that hit each of the PFP rules. Only HTTP and UDP rules contain PFP field, spawning new rules and duplicating the session space.

```
//Per-session data kept for the NAT
struct NATEntryIN {
    PortRef* ref;
    bool fin_seen;
};

class FlowNAT : public FlowStateElement<FlowNAT,NATEntryIN> {

public:

  [...]

  FLOW_ELEMENT_DEFINE_SESSION_CONTEXT("12/0/ffffffff "
                  "16/0/ffffffff 22/0/ffff 20/0/ffff", FLOW_TCP);

  void push_batch(int, NATEntryIN*, PacketBatch *);
}
```

Figure 5: Example of session definition for a self-contained NAT element

We added a few virtual functions to FastClick, allowing elements to ask for some traffic class or per-session scratchpad. Figure 5 shows a more convenient, higher-level abstraction of a Click element that defines a standard 4-tuple for TCP session and asks for some space to fit a given structure in the FCB.

C. 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), which dispatches traffic to following elements according to a given set of rules, as shown in figure 6 (a).

```
ct :: Classifier(12/0800,
                 12/0806 20/0001,
                 12/0806 20/0002);
cp :: IPClassifier(proto tcp, proto udp, -);
cd :: IPClassifier(dst tcp port 80, -);
td :: ToDevice(...)
arp_querier :: ARPQuerier(...) -> td;
lb :: LoadBalancer() -> arp_querier;
FromDevice(...) -> ct;
ct[0] -> cp;
ct[1] -> ARPResponder(...)[0];
ct[2] -> [1]arp_querier;
cp[0] -> cd;
cp[1] -> lb;
cd[0] -> HTTPProcessor() -> lb;
```

(a)

```
td ::ToDevice(...);
arp_querier :: ARPQuerier(...) -> td;
lb :: LoadBalancer() -> arp_querier;
fc :: FlowClassifier();
FromDevice(...) -> fc;
fc ~> ARPResponder(...)[0] -> td;
fc ~> [1]arp_querier;
fc ~> HTTPProcessor() -> lb;
fc ~> lb;
```

(b)

Figure 6: (a) Click configuration for the example of figure 3. (b) Corresponding MiddleClick configuration.

In MiddleClick, a `FlowClassifier` element must be placed at the beginning of each input path. The `FlowClassifier` will traverse the graph, using the traffic class and session definition to build the flow table as explained above. In MiddleClick, the `Classifier` is modified to expose its rules as a set of traffic classes. The `FlowClassifier` will therefore include the classifier's rules in the flow table, but also set the next hop number according to the Click output path in the FCB. Therefore, the `Classifier` just has to read the next hop number in the FCB to decide the output, without classifying in place or even touching the packet.

Alternatively, figure 6 (b) illustrates a new link syntax called the 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 allow to remove the needs for obvious classification. In our example, the input can directly be tied using the context link to all ARP elements, the flow defined by the ARP elements will be used to actually give 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 not needed.

If some elements rewrite headers that have been classified upon, such as a NAT, no re-classification is done. Indeed, in most cases the packets still belong to the same session.

If the session must change (*e.g.* divide a flow in multiple sub-flows in a dynamic way), the operator can place a new `FlowClassifier` in the chain that will assign new FCBs and therefore new sessions. It is up to the 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.

D. FCB size

All components of the service chain must be visited to compute the total FCB size. Starting from each input, the `FlowClassifiers` visit the downstream components and computes the total size needed for all of them. When a packet can traverse parallel paths but never both at the same time, the same space can be assigned to both paths to optimize space. The components are then informed of an offset in the FCB where they will be able to find their requested space. To avoid needing an indirection table, we prefer to waste some space and have an offset independent of the input path so each element has one and only one offset inside all potential FCBs assigned to packets passing by, eventually leading to some unused space. In figure 7 it would seem better to keep the Load

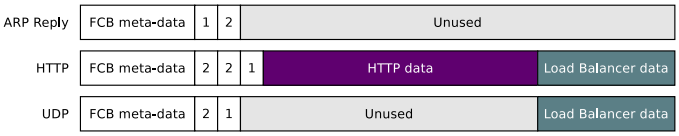

Figure 7: Computation of the size and offsets needed for the FCBs. Nodes contain the size they require in the FCB, while edges indicate the cumulative needed size (in bits).

Balancer data for UDP at the beginning of the block. But given that FCBs are pool-allocated, leading to constant memory allocation and a much easier recycling, and that offsets must be unique, it would only cause un-ordered access for HTTP flows.

If the data needed by a middlebox has variable size, the middlebox may simply ask for space for 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 memory allocation mechanism, like an efficient pool-allocator. FCBs are managed in per-thread pools for efficient allocation and recycling.

III. Stream abstraction

At this point of our design, a middlebox developer can easily receive a bunch of raw packets matching a given flow along with their FCB. The developer knows directly to which kind of traffic the given packets belong, as this is marked in the FCB. If the component asked for some per-session scratchpad, the middlebox component will also have some space for its own use in the FCB.

FastClick already implements batching using linked lists to pass lists of packets between elements, instead of single packets. In MiddleClick, batches of packets are always packets of the same session. To avoid having batches that are too small, the flow classifier has a builder mode to put packets into an internal ring of batches. For each packet, it searches the ring for packets of the same session, and append the packet to the end of the batch if found. When all packets are classified, the flow classifier sends the session batches to the next element one by one. Packets are reordered but the relative order inside the same session is kept.

Hence, any middlebox element knows that it can work on the payload of the packets of a batch as a single stream of data.

A. Contexts

Most of the time, a middlebox developer expects a seamless stream of data, not packets matching a given set of tuples, but from a given protocol. The developer also wants a way to touch the data without caring about the protocol details. Instead of letting each VNFs handle flow reconstruction and more generically protocol-dependent bookkeeping, we propose a context-based approach.

We introduce the concept of stream context. Entry and exit of contexts are done through pairs of IN and OUT elements, such as `TCPIn` and `TCPOut`. Call to the context is done through function calls to the *previous context* element. The previous context element will handle its protocol specifics and pass the request to the previous one and so on, until the first entry element (`IPIn` in most cases) finds no other context entry.

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

Figure 8: Configuration for a transparent middlebox that removes the word "ATTACK" of UDP flow passing by, even across packets. As UDP does not implement connection semantics, the UDPIn element can set the session timeout to some value, here 300 seconds.

Combined with context links, the usually complex Click manual wiring actually becomes minimal. Figure 8 shows a very simple example to implement a middlebox that will remove the word “ATTACK” from UDP flows, even across packet boundaries. Changing `UDPIn/Out` with `TCPIn/Out` in this example would work, as the `WordMatcher` element uses the abstract context system allowing to work on the “seamless stream” of data and modify it, no matter the protocol. We keep Click’s modularity but have a much more streamlined default case. Context links can always be omitted to use a more refined `Classifier` if the user wants finer control on classification.

The context allows to issue *requests* to act on or modify the stream. When in a given context, components can use a *content offset*, a metadata associated with each packets, to access the payload directly. On top of these facilities, we offer multiple abstractions that allow to act on the data as a stream, without the need to copy the packets payload, like an iterator that can iterate across packets per bytes or per chunks. This allows zero-copy inspection of a stream, but still allowing the middlebox component using the higher-level stream abstraction to access the headers if need be, a feature all IDS need as some attacks may be based on headers fields. This abstraction allows to implement new VNF components in a few lines, that would otherwise appear much more complex.

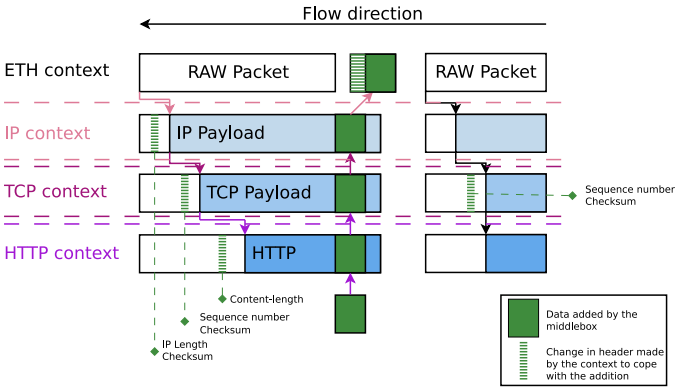

Figure 9: Context approach. Upon entry in a context, the payload offset is moved forward. When the stream is modified, higher layers of context take care of the implications, such as changing the TCP ACK number of the current packet and the following ones.

We already provide multiple generic VNF elements that act on the current context such as regular expression matcher, packet counter, load balancer and an accelerated NAT.

When a middlebox is in IP context, the content offset is set just after the IP header. If the middlebox modifies data in such a way that the IP packet length changes, the length in the IP header will be changed when the packet leaves the IP context, and the checksum will be updated accordingly. This is showed by the downwards arrows in figure 9.

In TCP context, the offset is moved forward after the TCP header of each packet. Each context does its own work when handling a request and then passes the request to the previous context.

On top of the functions to modify the packets, the context also allows to *determine if a given packet is the last useful one* for the current context. In TCP context, the request will simply check the TCP session state while HTTP context will use the value of *Content-Length* or pass it to the previous context if unknown. The context also allows to *close the current connection* when the context implements support for a stateful protocol such as TCP and *registers a function* to be called when that connection terminates, usually to clean the session scratchpad.

Modifications that impact the size of a stream are done through requests to *add bytes* and *remove bytes*. The first one is showed by the green boxes and lines in figure 9. Modification of the number of bytes in a TCP stream implies a lot of accounting around acknowledgement and sequence numbers detailed in section III-C, for the current packet but also the following ones. The IP header will also need to change if the packet length changes, so the request is passed to the previous context.

Figure 10 shows the code for a CRC computation element using the 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, allowing to do something with the final CRC. If the CRC was computed as a checksum to check against a database of known dangerous payload, the *process_data* could return a negative value to terminate the connection right away.

```
struct fcb_crc {
    unsigned int crc = 0xffffffff;
};

class FlowCRC : public StackChunkBufferElement<FlowCRC,fcb_crc> {
  public:

    [...]

    int process_data(fcb_crc*, FlowBufferChunkIter&);

    inline void release_stream(fcb_crc*) {
        //Do something with fcb_crc->crc;
    }
};

int
FlowCRC::process_data(fcb_crc* fcb, FlowBufferChunkIter& iterator)
{
    unsigned crc = fcb->crc;
    while (iterator) {
        auto chunk = *iterator;
        crc = update_crc(crc, (char *) chunk.bytes, chunk.length);
        ++iterator;
    }
    fcb->crc = crc;
    return 0;
}
```

Figure 10: Code for a CRC stream computation element.

B. TCP Flow stalling

One may want to buffer data before letting it go through. This is, for instance, the caser if the last received packet for the current session starts with the payload *AT*, and we're searching for (or replacing) occurence of the *ATTACK* pattern. Therefore a decision cannot be made before the next chunk of data is received. While our platform is protocol-agnostic, the TCP case is chosen to show it is fit for purpose. As a TCP source may wait for an ACK from the destination before sending more packets, and buffering data may prevent the destination from sending that ACK, this can lead to a deadlock. To solve this issue, the TCP context provides an optional functionality to do pro-active ACKing.

If enabled, when it receives a *request for more packet*, the TCP context sends an ACK for the given packet to the source with an acknowledgement number corresponding to the sequence that the destination would have sent. We therefore keep outgoing pre-ACKed TCP packets in a buffer until the destination acknowledge them. Buffering is done when a middlebox component specifies that it may stall or modify packets, or when the component wants to protect against TCP overlapping segment attacks [7]. 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 is done by using reference counting, avoiding any packet copy.

Figure 11 shows the code for a simple IPS using the byte iterator. It is using a DFA (assumed to be already built) with its state kept inside the FCB. If the iterator is at the end of the available payload, but the DFA is in the middle of a potential match, the iterator will be left at the last point known to be safe in the flow. Packets up to that point will be processed, others will be kept in the FCB and a *request for more data* will be made. This IPS, contrary to most IPS such as Snort [8] or Suricata [9] is not subject to eviction attacks, as the state of the matching is kept between windows. Moreover, the buffering of packets is minimal as only the data part of a potential match will be kept.

C. TCP Flow resizing

Stalling and re-ordering are requirements for modifying a stream. Many applications need to modify the stream content.

```
int FlowIDSMatcher::process_data(fcb_ids* fcb,
                    FlowBufferContentIter& iterator) {
    SimpleDFA::state_t state = fcb->state;

    //Position in the flow where there is no pattern for sure
    FlowBufferContentIter good_packets = iterator;

    while (iterator) {
        unsigned char c = *iterator;
        _program.next(c, state); //Advance the DFA
        if (unlikely(state == SimpleDFA::MATCHED)) {
            _matched++;
            return 1;
        } else if (state == 0) {
            //No possible match up to this point
            good_packets = iterator;
        }
        ++iterator;
    }
    if (state != 0) {
        //No more data, but left in the middle of a potential match
        iterator = ++good_packets;
    }
    fcb->state = state;
    return 0;
}
```

Figure 11: Code for a DFA-based IPS that is not subject to eviction, and only buffers data when the payload is missing the next bytes in a state that may lead to the detection of a pattern

For the specific web case, examples include rewriting HTTP traffic to change URLs to CDN-based URLs, ad-insertion or removal, along with potential new uses enabled by the novel performance of the lightweight in-the-middle stack we propose such as per-user targeted HTTP page modification, or a proxy cache that would include image content in the page itself. Pages could be translated on the fly to target the user language. Other non-web usages include some protocol translator, video transcoding or audio enhancement.

When the middlebox removes or adds data in a TCP stream, the sequence number must be set accordingly so the destination does not think the data has been lost or is a duplicate. However when the destination sends the corresponding ACK, the number must be mapped back to its original value. The TCP entry and exit components take care of modifying sequence and acknowledgement number on the fly by keeping track of the amount of removed and added bytes on both direction of the flow.

IV. State of the art

[10]–[12] implement user-level TCP stacks. In an NFV context, there is no need to fully terminate the TCP connection, packets pass through mostly untouched. Moreover, each instance of those stacks will bring up a session classification and TCP state management, which are factorized in our system. MiddleClick is able to automatically collect informations from the middlebox components and provide just enough, tailored services.

E2 [13], NetBricks [14] and mOS [15] implement in-the-middle TCP stacks with some similar abstractions, but do not provide any factorization and acceleration of the full service chain. Their flow abstraction for stream is limited to a less flexible window system, and do not provide a generic non-TCP specific stream abstraction nor the session scratchpad facility, likely losing a lot of performances when the box actually run many different VNFs. xOMB [16] provides some protocol independent modules through heavier message buffers, that eventually leads to performances more than two order of magnitude below ours (though, their paper is slightly older) for comparable functionality. Moreover none of those frameworks

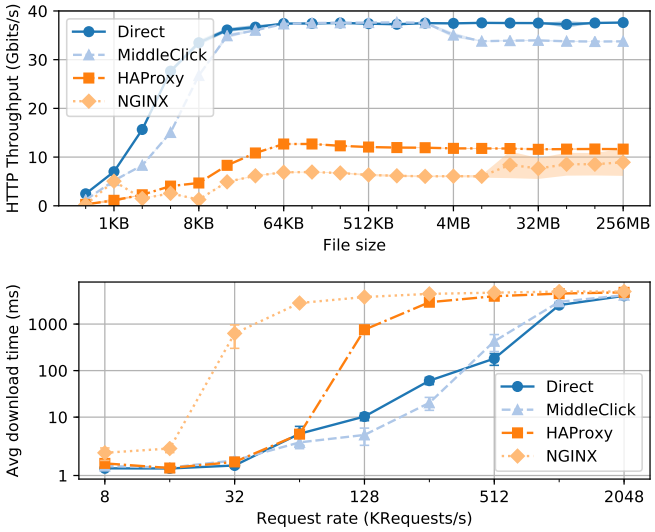

Figure 12: Data downloaded through a load-balancer using 128 concurrent connections. The proxy solution run on one core to constraint the performance by the CPU and not the test-bench capacity.

support efficient stalling and in-the-middle resizing of flows. Only the needed protocols layers are invoked according the context of each components, and when possible the layer is not left across middleboxes like chaining multiple standard socket-based application would, *i.e.* the stream is kept and checksums, protocols specifics such as ACKs numbers are only recomputed once for all middleboxes.

OpenBox [17], SNF [18], xOMB [16], NFP [19] and CoMb [2] optimize the graph to reuse some basic components or provide better CPU placement. CoMb decouples functions into common parts to allow to re-use some bricks. OpenBox and SNF goes a step further by differentiating kinds of bricks and re-order then - *when allowed* - to allow further merging of some bricks. While those works remove some redundant classification, none of them combine the session classification or provide an equivalent to the FCB facility.

V. Performance evaluation

Tests are always done using 3 machines. One *client* that generates traffic, a *Device Under Test (DUT)* and a traffic *sink*. The *client* and *dut* have a 8-core Xeon E5-2630 v3, 32 GB of RAM, while the *sink* has a 16-core E5-2683 v4, 64 GB of RAM. All machines are interconnected using Intel XL710 2*40G NICs. All experiments are executed 3 time each for 20 seconds.

A. TCP load-balancing reverse proxy

To evaluate the flow performances of our system against state-of-the art industrial solutions, we built a TCP load-balancing reverse proxy. The proxy balances in a round-robin way the incoming HTTP connections to multiple destinations, making sure that packets of the same session go to the same destination. It is an application typical of datacenters. The requests traversing the proxy are NATed, to ensure that the packets go back through the box so the source IP address can be set back to the original destination address.

We compared our solution to HAProxy in TCP mode, and NGINX in load-balancing mode. The proxy load-balances in a round-robin way the connections towards another NGINX server running on the *sink* that listens on 4 IP addresses for this test. We made requests for file sizes from 0 KB to 256 MB. The achieved throughput can be seen in figure 12, along with the average latency to download 8K files under an increasing request rate. The *direct* line shows the performance that can be achieved without the *DUT*, that is the baseline of the testbed.

Our solution outperforms HA Proxy and NGINX in term of throughput for every object sizes, ranging from a $\sim 5X$ improvement for small file sizes to a $\sim 3X$ improvement with bigger file sizes. When the *DUT* becomes the bottleneck, the latency increases for the others solutions, while ours achieves the limit of the testbed.

B. Service chaining

Figure 13 shows the performance of running some service chains on 1 to 4 cores for 8K HTTP requests. First, we compare NAT implementations using MiddleClick, FastClick, mOS and the Linux NAT. Both FastClick and MiddleClick actually achieve the limit of the testbed. That is $\sim$ 34 Gbps, using a single core of the *dut*. Linux and mOS solutions fall far behind. When adding a statistics VNF simply counting all bytes per-sessions to the FastClick and MiddleClick solutions, FastClick performances drop considerably because of the second session classification. MiddleClick, however, still achieves the limit of the testbed, as adding this function only extends the flow table per 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, a hit is introduced that lowers performances to 20 Gbps, a cost paid for TCP state management and reordering of TCP packets. Adding VNFs for flow statistics (byte count per-session but only for the useful payload), a load balancer, and a computation of a checksum (similar to the CRC computation element presented in figure 10 except it does a 4-bytes checksum) induce very little impact as they will all have their space in the FCB at the same cost, extended to fit all VNFs of the service chain by MiddleClick without any manual tuning. As a point of comparison, mOS with only stream statistics is performing worst than the MiddleClick chain running 4 more functions. Adding an IPS (simple string matcher) induces a bigger hit because of the pattern matching algorithm that must sill be improved. When using 2 cores, the chains up to the load balancer achieves ~ 34 Gbps, while 3 cores are enough to run the chain up to the checksum.

VI. Conclusion

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

Our framework also eases the handling of per-flow or per-session state. The middlebox developer can specify, in a flexible way, which flows or sessions the middlebox is interested in, and the size of the state it needs for each flow/session. Then, the system automatically provides and manages the associated per-flow/session storage, which is directly available to the middlebox components.

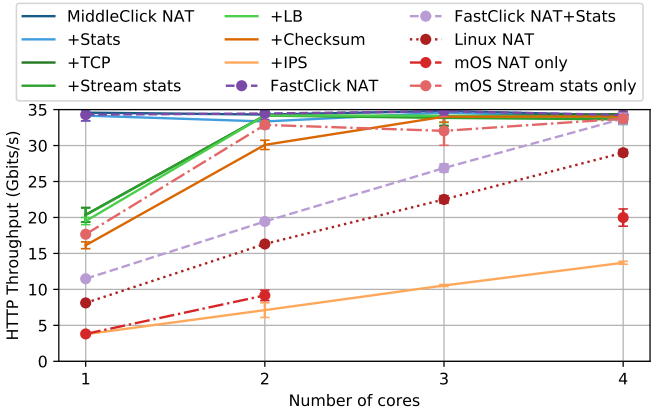

Figure 13: MiddleClick advantage when chaining multiple NFV components compared to two functionally identical FastClick setups for NAT and Statistics

Finally, our framework exposes simple stream abstractions, providing easy inspection of flow content at any protocol level. The developer only needs to focus on the middlebox functionality at the desired protocol level, and the framework will adjust 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 middleboxes development, while avoiding the overhead of a full TCP stack.

Our open-source implementation is available[1].

Acknowledgment

This work has been funded by the *Fond National de la Recherche Scientifique (*FNRS*)* through the PDR ePi project.

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," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[2] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 24–24.

[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] 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.

[5] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems*. IEEE Computer Society, 2015, pp. 5–16.

[1] https://github.com/tbarbette/fastclick/tree/middleclick

[6] 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

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

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

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

[10] I. Marinos, R. N. Watson, and M. Handley, "Network stack specialization for performance," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 2013, p. 9.

[11] 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 *NSDI*, 2014, pp. 489–502.

[12] R. Laufer, M. Gallo, D. Perino, and A. Nandugudi, "Climb: enabling network function composition with click middleboxes," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 4, pp. 17–22, 2016.

[13] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for nfv applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 121–136.

[14] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of nfv." in *OSDI*, 2016, pp. 203–216.

[15] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "mos: A reusable networking stack for flow monitoring middleboxes." in *NSDI*, 2017, pp. 113–129.

[16] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xomb: extensible open middleboxes with commodity servers," in *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*. ACM, 2012, pp. 49–60.

[17] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: a software-defined framework for developing, deploying, and managing network functions," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 511–524.

[18] 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.

[19] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "Nfp: Enabling network function parallelism in nfv," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 43–56.