StreaMon: a Software-defined Monitoring Platform

Giuseppe Bianchi, Marco Bonola, Giulio Picierro, Salvatore Pontarelli, Marco Monaci
University of Rome Tor Vergata

***Abstract*—The fast evolving nature of modern cyber threats and network monitoring as well as the increasing interest in virtualization approaches for more complex network middlebox functionalities call for new, "software-defined", solutions to virtualize and simplify the programming and deployment of online (stream-based) traffic analysis functions. StreaMon is based on a data-plane abstraction devised to scalably decouple the "programming logic" of a traffic analysis application (tracked states, features, anomaly conditions, etc.) from elementary primitives (counting and metering, matching, events generation, etc), efficiently pre-implemented in the probes, and used as common instruction set for supporting the desired logic. The proposed SDN approach entails platform-independent, portable, multi-tenant online traffic analysis tasks written in a high level language and enables system users to completely virtualize network monitoring functionalities, isolate aggregated traffic flows and run multiple independent applications on a single software instance of the StreaMon platform. We validate our design by developing a prototype and a set of simple (but functionally demanding) use-case applications and by testing them over real traffic traces.**

***Index Terms*—Network monitoring, XFSM, network programmability**

I. Introduction

In this paper we propose StreaMon, a software defined platform for stream-based monitoring tasks directly running over network probes. StreaMon is devised as a pragmatic tradeoff between full programmability and vendors' need to keep their platforms *closed*. StreaMon's strategy closely resembles that pioneered by Openflow [9] in the abstraction of networking functionalities, thus paving the road towards software-defined networking. However, the analogy with Openflow limits to the strategic level; in its technical design, StreaMon significantly departs from Openflow for the very simple reason that (as discussed in Section III-A) the data-plane programmability of monitoring tasks exhibits very different requirements with respect to the data-plane programmability of networking functionalities, and thus mandate for different programming abstractions.
The actual contribution of this paper is threefold.

(1) We identify (and design an execution platform for) an extremely simple abstraction which appears capable of supporting a wide range of monitoring application requirements. The proposed API decouples the monitoring application "logic", externally provided by a third party programmer via (easy to code) eXtended Finite State Machines (XFSM), from the actual "primitives", namely configurable sketch-based measurement modules, d-left hash tables, state management primitives, and export/mitigation actions, hard-coded in the device.

(2) We implement two StreaMon platform prototypes, a full SW and a FPGA/SW integrated implementation. We functionally validate them with five use case examples (P2P traffic classification, Conficker botnet detection, DDos detection), not meant as stand-alone contributions, but rather selected to showcase the StreaMon's adaptability to different requirements.

(3) We assess the performance of the proposed approach: even if the current prototype implementation is not primarily designed with performance requirements in mind, we show that it can already sustain traffic in the multi-gbps range even with several instantiated metrics (e.g. 2.315 Gbps of real world replayed traffic with 16 metrics, see section V-B)

II. Related Work

In the literature, several monitoring platforms have targeted monitoring applications' programmability. A Monitoring API for programmable HW network adapters is proposed in [14]. On top of such probe, network administrators may implement custom C++ monitoring applications. One of the developed applications is Appmon [2]. It uses deep packet inspection to classify observed flows and attribute these flows to an application. Flow *states* are stored in an hash table and retrieved when an *old* flow is observed again. This way to handle states bears some resemblance with that proposed in this work, which however makes usage of (much) more descriptive eXtended Finite State Machines. CoMo [6] is another well known network monitoring platform. We share with CoMo the (for us, side) idea of extensible plug-in metric modules, but besides this we are quite orthogonal to such work, as we rather focus on how to combine metrics with features and states using higher level programming techniques (versus CoMo's low level queries).

Bro [10] provides a monitoring framework relying on event-based programming language for real time statistics and notification. Despite the attempt to define a versatile high level language, Bro is not designed to expose a clear and simple abstraction for monitoring application development and leave full programmability to its users (which we believe results in a more descriptive and yet more complex programming language).

978-0-9836283-9-2 © 2014 ITC

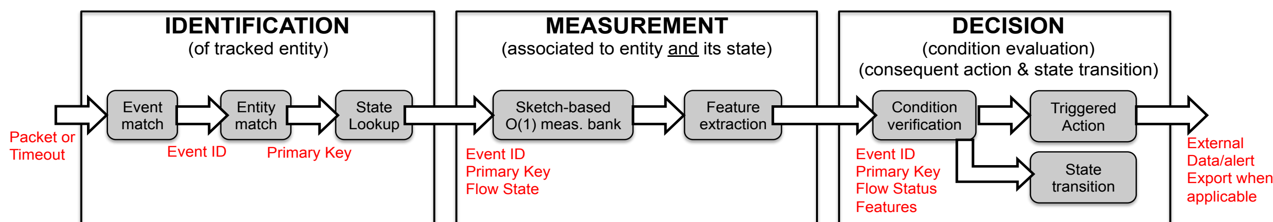

Fig. 1: StreaMon data plane identification/measurement/decision abstraction, and its mapping to implementation-specific workflow tasks

The Real-Time Communications Monitoring (RTCMon) framework [4] permits development of monitoring applications, but again the development language is a low level one (C++), and (unlike us) any feature extraction and state handling must be dealt with inside the custom application logic developed by the programmer. CoralReef [8], FLAME [1] and Blockmon [5] are other frameworks which grant full programmability by permitting the monitoring application developers to "hook" their custom C/C++/Perl traffic analysis function to the platform.

Opensketch [15] is a recent work proposing an efficient generic data plane based on programmable metric sketches. If on the one hand we share with Opensketch the same measurement approach, on the other hand its data plane abstraction delegates any decision stage and logic adaptation the control plane and, with reference to our proposed abstraction, does not go beyond the functionalities of our proposed Measurement Subsystem. On the same line, ProgME [16] is a programmable measurement framework which revolves around the extended and more scalable notion of dynamic flowset composition, for which it provides a novel functional language.

Even though equivalent dynamic tracking strategies might be deployed over Openflow based monitoring tools, by exploiting multiple tables, metadata and by delegating "monitoring intelligence" to external controllers, this approach would require to fully develop the specific application logic and to forward all packets to an external controller, (like in Openflow based monitoring tool Fresco [11]), which will increase complexity and affect performance.

Finally, while our work is, to the best of our knowledge, the first which exploits eXtended Finite State Machines (XFSM) for programming custom monitoring logic, we acknowledge that the idea of using XFSM as programming language for networking purposes was proposed in a completely different field (wireless MAC protocols programmability) by [13].

III. The StreaMon platform

A. Data-plane abstraction

Our strategy in devising an abstraction for deploying stream-based monitoring tasks over a (general-purpose) network monitoring probe is similar in spirit to that brought about by the designers of the Openflow [9] match/action abstraction, for programming networking functionalities over a switching fabric. Indeed, we also aim at identifying a compromise between full programming flexibility, so as to adapt to the very diverse needs of monitoring application developers and permit a broad range of innovation, and consistency with the vendors' need for closed platforms. However, the requirements of monitoring applications appear largely different from that of a networking functionality, and this naturally drives towards a *different* pragmatic abstraction with respect to a match/action table: *first*, the "entity" being monitored is not consistently associated with the same field composition in the packet header (e.g: 2 way communication to/from the same IP address). *Second*, the type of analysis (and possibly the monitoring entity target) entailed by a monitoring application may change over time, dynamically adapting to the knowledge gathered so far. *Third*, activities associated to a monitoring task are not all associated to a matching functionality, but they rather require *triggering conditions* applied to the gathered features.

Our proposed StreaMon abstraction, illustrated in figure 1, appears capable to cope with such requirements (as more extensively shown with the use cases presented in section IV). It comprises of three "stages", programmable by the monitoring application developer via external means (i.e. not accessing the internal probe platform implementation).

(1) The **Identification** stage permits the programmer to specify what is the monitored entity (more precisely, deriving its primary key, i.e., a combination of packet fields that identify the entity) associated to an event triggered by the actual packet under arrival, as well as retrieve an eventually associated state.

(2) The **Measurement** stage permits the programmer to configure which information should be accounted. It integrates *hard-coded and efficiently implemented* hash-based measurement primitives (metric modules), fed by configurable packet fields, with externally programmed *features*, expressed as arbitrary arithmetic operations on the metric modules' output.

(3) The **Decision** stage is the most novel aspect of our abstraction. It permits to define the application logic in the form of eXtended Finite State Machines (XFSM), i.e. check conditions associated to the current state and tested over the currently computed features, and trigger associated actions and/or state transitions.

B. System architecture and API

The previously introduced abstraction can be concretely implemented by a stream processing engine whose archi-

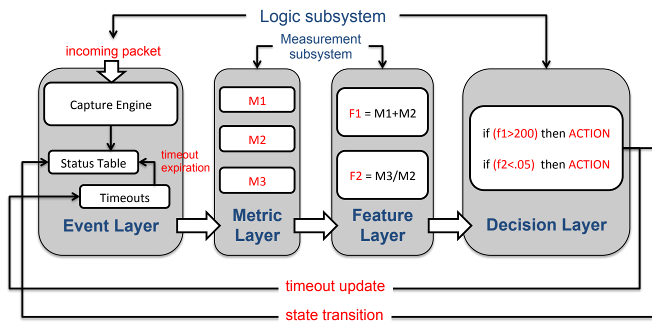

Fig. 2: StreaMon processing engine architecture

tecture is depicted in Figure 2. It consists of four modular layers descriptively organized into two subsystems, namely *Measurement subsystem* and *Logic subsystem*.

Event layer - Such layer is in charge of parsing each raw captured packet, and match an *event* among those user-programmed via the StreaMon API. The matched event identifies a user-programmed *primary key* which permits to retrieve an *eventually* stored state. The event layer is further in charge of supplementary technical tasks, such as handling special timeout events, deriving further secondary keys, etc.

Metric layer - StreaMon operates on a per-packet basis and does *not* store any (raw) traffic in a local database. The application programmer can instantiate a number of *metrics* derived by a basic common structure, implemented as computation/memory efficient multi-hash data structures (i.e., Bloom-type sketches), updated at every packet arrival.

Feature layer - this layer permits to compute user-defined arithmetic functions over (one or more) metric outputs. Whereas metrics carry out the bulky task of accounting *basic* statistics in a scalable and computation/memory efficient manner, the features compute *derived* statistics tailored to the specific application needs, at no (noticeable) extra computational/memory cost.

Decision layer - this final processing stage implements the actual application logic. This layer keeps a list of *conditions* expressed as mathematical/logical functions of the feature vector provided by the previous layer and any other possible secondary status. Each condition will trigger a set of specified and pre-implemented *actions* and a state *transition*.

Application programmers describe their desired monitoring operations through an high-level XML-like language, which permits to specify custom (dynamic) states, configure measurement metrics, formalize when (e.g.in which state and for which event) and how (i.e. by performing which operations over available metrics and state information) to extract features, and under which conditions trigger relevant actions (e.g. send an alert or data to a central controller). We remark that a monitoring application formally specified using our XML description does not require to be *compiled* by application developers, but is run-time installed, thus significantly simplifying on-field deployment. Figure 3 shows an excerpt of a StreaMon application code.

```
<source type="live" name="eth2"/>
<statedef id="normal" />
<statedef id="alert"/>
<statedef id="infected" timeout="60" next_state="alert"/>

<metrics>
    <metric name="dns_qry">
        <v_monitor status="on" type="tewma" window="60"/>
    </metric>
    <metric name="dns_nxd">
        <v_monitor status="on" type="tewma" window="60"/>
    </metric>
    <metric name="syn445">
        <v_monitor status="on" type="tewma" window="60"/>
    </metric>
    <metric name="synack445">
        <v_monitor status="on" type="tewma" window="60"/>
    </metric>
</metrics>
<table name="tset" type="DLeft" nhash="8" shash="20"/>
<features>
    <feature name="NXDRatio" body="dns_nxd/dns_qry"/>
    <feature name="DNSQry" body="dns_qry"/>
    <feature name="DNSNxd" body="dns_nxd"/>
    <feature name="SYNACK445Ratio" body="synack445/syn445"/>
</features>

<event type="timeout" class="t1">
    <state id="alert">
        <use-metric id="dns_qry" vm_get="ip_dst"/>
        <use-metric id="dns_nxd" vm_get="ip_dst"/>
        <use-metric id="syn445" vm_get="ip_dst"/>
        <use-metric id="synack445" vm_get="ip_dst"/>
        <condition expression="SYNACK445Ratio &lt; 0.5"
                              action="" next_state="infected"/>
    </state>
    <state id="infected">
        <use-metric id="dns_nxd" vm_get="ip_dst"/>
        <post-condition-action
            do="print(timeout expired flow: %ip_dst)"/>
    </state>
</event>

<event type="packet" primary-key="ip_dst"
selector="proto udp and src_port 53 and flags dns-nxdomain">
    <state id="normal">
        <use-metric id="dns_qry" vm_get="ip_dst"/>
        <use-metric id="dns_nxd" vm_update="ip_dst"/>
        <condition expression="NXDRatio&gt;0.2"
        action="print(to for %ip_dst)" next_state="alert">
            <timeout_set class="t1" key="ip_dst" value="3"/>
        </condition>
        <post-condition-action do="print(found nxdomain)"/>
    </state>
    <state id="alert">
        <use-metric id="dns_nxd" vm_update="ip_dst"/>
    </state>
    <state id="infected">
        <use-metric id="dns_nxd" vm_update="ip_dst"/>
    </state>
</event>
```

Fig. 3: Excerpt of XML based StreaMon code showing: (i) packet source definition, (ii) state dafinition, (iii) metric element and DLEFT table allocation, (iv) feature compositions, (v) event logic description. In particular this picture shows a timeout event handler, described in terms of metric operations, feature extractions, conditions, actions and state transition. Moreover a packet event is depicted. Note the event selector description (DNS NXDOMAIN reply) and the primary key extraction (destination IP address)

IV. Simple Use case examples

We use the following simple examples to highlight the flexibility of StreaMon in supporting heterogeneous features commonly found in real-world monitoring applications. The input data traces are obtained by properly merging a packet trace gathered from a regional Internet provider with either (i) real malicious traffic extracted from

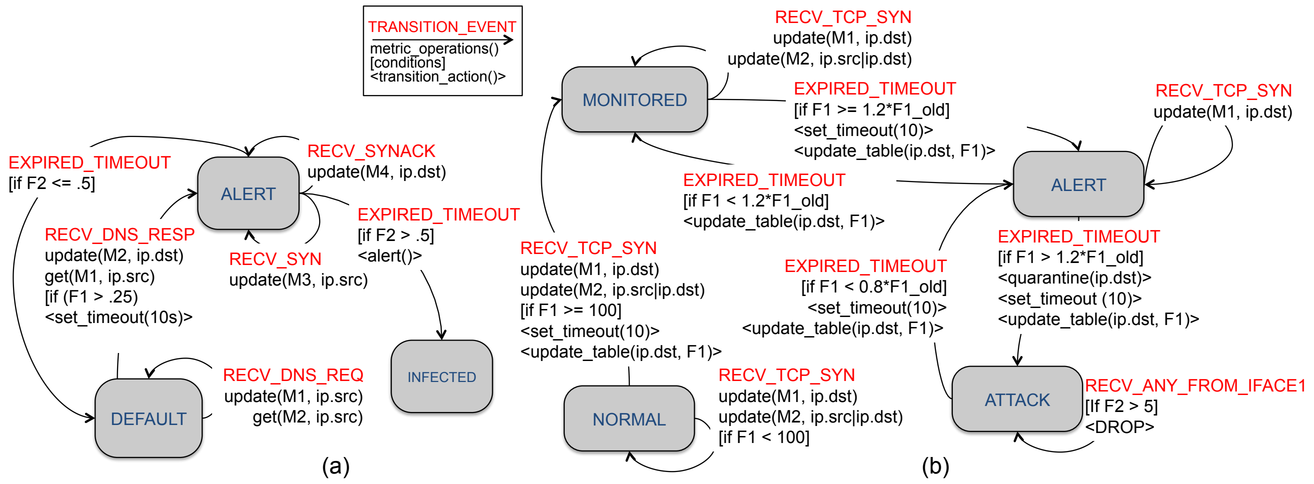

Fig. 4: Application XFSMs: (a) Conficker use case; (b) DDOS use case

traces captured in our campus network (use case IV-A and IV-B) or (ii) *synthetic* traces properly generated in our laboratories (use case IV-C).

A. P2P traffic classification

This example shows how straightforward is the implementation of three transport layer traffic features described in [7], for detecting peer to peer protocols that use UDP and TCP connections. The (stateless) application considers the following packet events (which will ignore well known UDP and TCP ports.):

$E_1 : if(ip.proto == UDP)\&\&(udp.port \neq 25, 53, 110, ...)$

$E_2 : if(ip.proto == TCP)\&\&(tcp.port \neq 25, 53, 110, ...)$

This application extracts the following traffic features $F_1 = M_1\&M_2$; $F_2 = |M_3 - M_4|$ where:

$\{M_1, M_2\}$: VD enabled - return 1 for each IP src/dst pair which previously opened a {UDP, TCP} socket, 0 otherwise;

$\{M_3, M_4\}$: VD enabled and VM type CBF - count the number of different {TCP source ports, hosts} connected to the same destination IP address.

Metrics are read/updated on the basis of the matched event, as follows:

E_1	E_2
$set(M_1, ip.src\|ip.dst)$ $get(M_2, ip.src\|ip.dst)$ $get(M_3, ip.dst)$ $get(M_4, ip.dst)$	$get(M_1, ip.src\|ip.dst);$ $set(M_2, ip.src\|ip.dst)$ $set(M_3, tcp.sport\|ip.dst, ip.dst)$ $set(M_4, ip.src\|ip.dst, ip.dst)$ $get(M_i, ip.src), i = 3, 4$

The application detects a p2p client if the following condition holds after a transitory period: $(F_1 == 1\&\&(F_2 < 10))$.

B. Conficker botnet detection

Conficker is one of the largest Botnets found in recent years [12]. A multi-step detection algorithm can attempt to track the two following phases: (1) a bot tries to contact the C&C Server, and (2) a single bot tries to contact and infect other hosts.

To contact the C&C Server, infected hosts perform a huge number of DNS queries (with a high NXDomain error probability) to resolve randomly generated domains. In the *infection* phase, every host tries to open a TCP connection to the ports 445 of random IPs. Our Conficker detector will use the following metrics (VD disabled and VM of type TBF): number of total DNS queries per host (M_1), number of DNS NXDomain per host (M_2) and number of TCP SYN and SYNACK to/from port 445 (respectively M_3 and M_4). These metrics are combined into the following features: $F_1 = M_2/M_1$, $F_2 = M_4/M_3$.

For a DNS NXDomain response, the condition $F_1 > 0.25$ is checked. If the condition is true, the state of the actual flow changes to *alert* and an *event timeout* is set. In the alert state the application updates M_3 and M_4 and, when this timeout expires, these metrics are used to compute F_2 and to verify the related condition: if the condition is true, then the host is considered *infected* and goes to the next state, otherwise it returns to the default state.

The application XFSM is graphically described (with simplified syntax) in Figure 4.a. Figure 5 shows the trend of features used in this configuration, for a host infected by Conficker (A) and for a *clean* host (B). In the first case, the value of F_1 is relatively high since the beginning of the monitoring (due to the presence of reverse DNS queries, easily filtered by the application); the value increases when the host starts to perform *Conficker queries*. The value of F_2 instead is very low, clearly denoting a port scan. Also for the host B the presence of rDNS queries increases the value of F_1 and this involve a change of state, and the application starts to analyze TCP feature. However the F_2 value (nearly 100% of SYNACKs are received) reveals that this is clearly not a network scan. Testing this configuration in a 90-hours trace with 53 different hosts in idle state, we obtained 100% of detection (8 infected hosts detected) without false positive or false negative.

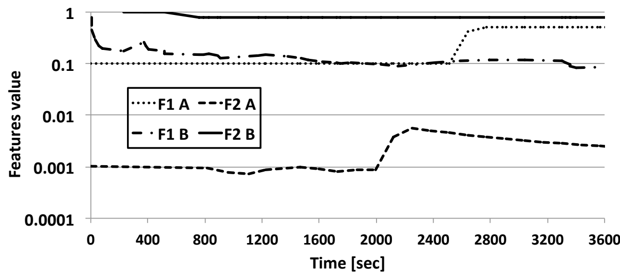

Fig. 5: Conficker features temporal evolution

C. DDOS detection and mitigation

In this section we sketch a simple algorithm which can be used as an initial base for detecting and mitigating DDOS attacks. The algorithm is driven by the number of SYN packets received by possible DDOS targets. The XFSM of this configuration is depicted in 4.b, and is governed by the following two events:

$E_1 : if(ip.proto == TCP)\&\&(tcp.flags == SYN)$

$E_2 : timeoutexpired$

Metric M_1 (VD=off, VM=TBF) tracks the number of TCP SYN addressed to a same target in 60 seconds (with 240s TBF's memory). All external servers for which $F_1 = M_1$ is under a given threshold (100 in this example) are in default state (because they are obviously not under attack and thus do not need an explicit status). When F_1 exceeds this threshold, the target goes in *monitored state*, a timeout is set and the current value of F_1 value is stored into a secondary support table with key ip.dst. As soon as this timeout expires, the difference between the current feature value and the one stored is computed. If the condition $if(F_1(t) > 1.2 * F_1(t-1))$, holds for two consecutive times, i.e. the rate of TCP SYN has increased twice for more than 20%, the flow goes into an *attack* state.

Our use case example mimics this mitigation strategy by considering legitimate all the source hosts that have already shown meaningful activity and contacted the server *before the actual emergence of the attack*, and thus likely dropping most of the TCP SYN having a spoofed source IP address. In our use case example, a second metric M_2 tracks the TCP SYN rate, smoothed over a chosen time window, at which which each host contacts each destination IP addresses (M_2 is configured with VD disabled and VM of type TBF with smoothing window 240s, and it is updated with MFK (IP_{src}, IP_{dst}).

Figure 6 shows the temporal evolution of the state of a server, in an experiment where we performed a DDoS attack, with spoofed TCP SYN packets, after about 140s. The figure also reports the measured TCP SYN rate, as well as the traffic generated by two hosts performing regular queries towards the server: one starting at time 0, and the other starting right in the middle of the attack. When the actual attack is detected, the mitigation strategy starts filtering traffic. Thanks to the second tracked metric, M_2, the user starting activities before the DDoS attack

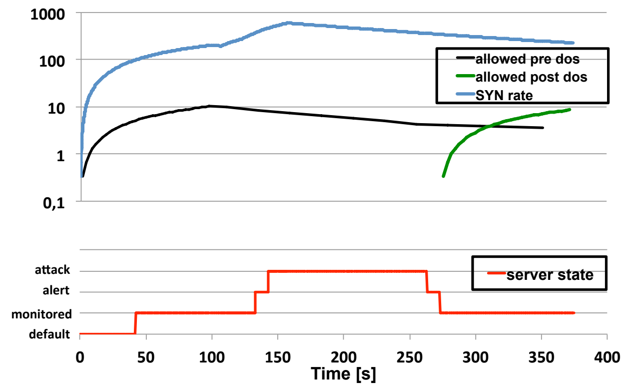

Fig. 6: DDOS temporal representation: (i) DDOS target host status (red curve), (ii) F_1 value for the DDOS target (blu curve) and (iii) F_2 for two different legitimate traffic sources connecting to the DDOS target server

is not filtered; on the contrary, connections from sources not previously detected via the M_2 metric are blocked, as shown by the green curve. New connections can be accepted as soon the server leaves the *attack* state (see the F_2 growth of the second host).

V. Performance evaluation

A. Implementation overview

We experimented StreaMon on a SW-only deployment leverages off-the-shelf NIC drivers, integrating the efficient PFQ packet capturing kernel module [3].

StreaMon start-up sequence is summarized as follows. The application program is given as input to a pre-processor script as a XML formatted textual file. The program is parsed, StreaMon process is configured and executed while in parallel the interpreted feature and condition expressions are transformed them into C++ code and build a "on-demand DLL"[1];

StreaMon implementation takes advantages from multi-core platforms by implementing parallel processing chains as shown in the implementation architecture depicted in figure 7 (for the SW-only setup). The PFQ capture module is used as packet filter and configured with a number of software queues equal to the number of cores. Traffic flows are dispatched from the physical queues of the NIC by the PFQ steering function. For each PFQ, a separate StreaMon chain is executed by a single thread with CPU affinity fixed to one of the available core and share the metric banks, status tables and timeout tables (the concurrent access to the shared data is protected by spinlocks).

The throughput measurement has been performed on a Intel Xeon X5650 (2.67 GHz, 6 cores) Linux server with, 16 GB ram and Intel 82599eb 10Gbit optical interfaces.

[1]Note that this is an optimization step which is devised to on-the-fly compile (transparent to the programmer), the application code, so as to avoid a "feature/condition interpreter", which would have lowered the overall performance and would not have permitted line rate feature computation and condition verification. Since no recompilation of the StreaMon code is ever needed, but user defined features are integrated as a DLL, dynamic deployment of user programs is made possible.

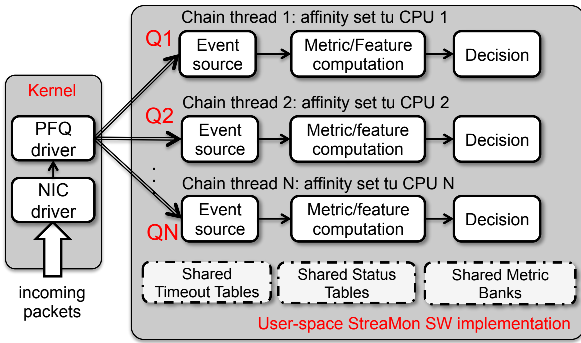

Fig. 7: StreaMon prototype implementation architecture

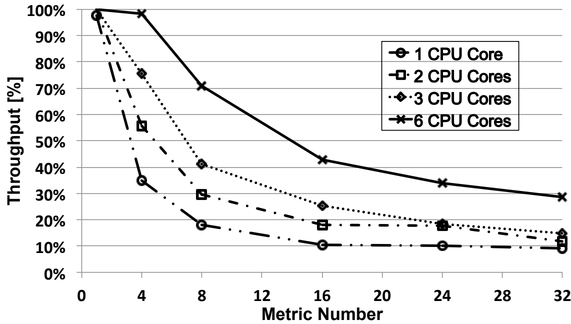

Fig. 8: System throughput evaluation

B. Preliminary performance analysis

Figure 8 shows the processing throughput expressed as percentage of the maximum throughput (6,468 Gbps) obtained with one PFQ source block without the overhead of StreaMon, expressed as function of the number of metrics (1, 4, 8, 16, 24, 32) in case of (1, 2, 3, 6) CPU core parallel processing. The input test data is a small portion of a long trace captured within a local internet provider infrastructure and its parameters are summarized in the following table:

pkt no.	AVG pkt len	Host no.	TX rate
6320928	632.82 bytes	31827	6.47 Gbps

where *pkt len* and *TX* are the average packet length and replayed transmission bitrate respectively.

As expected, the throughput decreases with the number of metrics and grows with the number of cores (even though the growth is lower than expected due to the simple thread concurrency management adopted in this prototype). It is important to underline that such graph shows the worst metric computational scenario in which: (i) all metrics are sequentially updated and retrieved for each packet (single event and stateless logic) and are configured with both the VD (a BF pair) and the MV (DLEFT) enabled; (ii) all flow keys are different (n metrics, n * 2 flow keys). Nevertheless, the results are promising, as for example in case of 16 metrics, for which we observe the following average bitrate (Gbps):

1 core	2 cores	3 cores	6 cores
0.566	0.983	1.367	2.315

VI. Conclusion

The StreaMon programmable monitoring framework described in this paper aims at making the deployment of monitoring applications as fast an easy as configuring a set of pre-established metrics and devising a state machine which orchestrates their operation while following the evolution of attacks and anomalies. Indeed, the major contribution of the paper is the design of a pragmatic application programming interface for developing stream-based monitoring tasks, which does not require programmers to access the monitoring device internals. Despite their simplicity, we believe that the wide range of features accounted in the proposed use cases suggest that StreaMon's flexibility can be exploited to develop and deploy several real world applications.

References

[1] K. G. Anagnostakis, M. Greenwald, S. Ioannidis, and S. Miltchev. Open packet monitoring on flame: Safety, performance, and applications. In *4th Int. Conf. on Active Networks, IWAN '02*, 2002.

[2] D. Antoniades, M. Polychronakis, S. Antonatos, E. P. Markatos, S. Ubik, and A. Oslebo. Appmon: An application for accurate per application network traffic characterisation. In *IST Broadband Europe*, 2006.

[3] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi. On multi-gigabit packet capturing with multi-core commodity hardware. In *Passive and Active Measurement (PAM)*, pages 64–73, 2012.

[4] F. Fusco, F. Huici, L. Deri, S. Niccolini, and T. Ewald. Enabling high-speed and extensible real-time communications monitoring. In *11th IFIP/IEEE Integrated Network Management Symp., IM'09*, 2009.

[5] F. Huici, A. di Pietro, B. Trammell, J. M. Gomez Hidalgo, D. Martinez Ruiz, and N. d'Heureuse. Blockmon: a high-performance composable network traffic measurement system. In *Proc. ACM SIGCOMM 2012, Demo*, pages 79–80, 2012.

[6] G. Iannaccone, C. Diot, D. McAuley, A. Moore, I. Pratt, and L. Rizzo. The como white paper. In *Intel Research Cambridge, Tech. Rep. IRCTR-04-017*, 2004.

[7] T. Karagiannis, A. Broido, M. Faloutsos, and K. Claffy. Transport layer identification of p2p traffic. In *4th ACM Internet Measurement Conference (IMC '04)*, pages 121–134, 2004.

[8] K. Keys, D. Moore, R. Koga, E. Lagache, M. Tesch, and K. Claffy. The architecture of coralreef: an internet traffic monitoring software suite. In *Passive and Active Measurements (PAM)*, 2001.

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[10] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.

[11] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. Fresco: Modular composable security services for software-defined networks. In *ISOC NDSS*, 2013.

[12] B. Stone-gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: Analysis of a botnet takeover, 2009.

[13] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, and F. Gringoli. Wireless mac processors: Programming mac protocols on commodity hardware. In *INFOCOM*, pages 1269–1277, 2012.

[14] P. Trimintzios, M. Polychronakis, A. Papadogiannakis, M. Foukarakis, E. P. Markatos, and A. Oslebo. Dimapi: An application programming interface for distributed network monitoring. In *10th IEEE/IFIP NOMS*, 2006.

[15] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *10th USENIX NSDI 2013*, pages 29–42.

[16] L. Yuan, C.-N. Chuah, and P. Mohapatra. Progme: towards programmable network measurement. *IEEE/ACM Trans. Netw.*, 19(1):115–128, 2011.