ABSTRACT

The fast evolving nature of modern cyber threats and network monitoring needs calls for new, “software-defined”, approaches to simplify and quicken programming and deployment of online (stream-based) traffic analysis functions. StreaMon is a carefully designed 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. Multi-stage multi-step real-time tracking and detection algorithms are supported via the ability to deploy custom states, relevant state transitions, and associated monitoring actions and triggering conditions. Such a separation entails platform-independent, portable, online traffic analysis tasks written in a high level language, without requiring developers to access the monitoring device internals and program their custom monitoring logic via low level compiled languages (e.g., C, assembly, VHDL). 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.

1. INTRODUCTION

The sheer volume of networked information, in conjunction with the complexity and polymorphous nature of modern cyberthreats, calls for scalable, accurate, and, at the same time, flexible and programmable monitoring systems [13][11]. The challenge is to promptly react to fastly mutating needs by deploying custom traffic analyses, capable of tracking event chains and multistage attacks, and efficiently handle the many heterogeneous features, events, and conditions which characterize an operational failure, a network application misbehavior, an anomaly or an incoming attack. Such needed level of flexibility and programmability should address scalability by design, through systematic exploitation of stream-based analysis techniques. And, even more challenging, traffic analyses and mitigation primitives should be ideally brought inside the monitoring probes themselves at data plane, so as to avoid exporting traffic data to central analysis points, an hardly adequate way to cope with the traffic scale and the strict (ideally real time) mitigation delay requirements.

To face this rapidly evolving scenario, in this paper we propose StreaMon, a data-plane programming abstraction 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 [25] in the abstraction of networking functionalities, thus paving the road towards software-defined networking[1]. 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 2), 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. While our handling of sketch-based measurement primitives may recall [34], we radically differentiate from any other work we are aware of, in our pro-

* This work will be submitted for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.
proposa to inject monitoring logic in the form of XFSM devised to locally orchestrate and run-time adapt measurements to the tracked state for each monitored entity (flows, hosts, DNS names, etc), with no need to resort to an external controlling 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, packet entropy HW analysis, DDos detection, Port Knocking), not meant as stand-alone contributions, but rather selected to showcase the StreaMon’s adaptability to different application 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 (for example 2.315 Gbps of real world replayed traffic with 16 metrics, see section 5.2); moreover, we show that scalability can be easily enhanced by offloading the SW implementation with HW accelerated metrics; in essence, it seems fair to say that scalability appears to be an architectural property of our proposed API, rather than a side effect of an efficient implementation.

2. STREAMON 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 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.

At least three important differences do emerge. First, the “entity” being monitored is not consistently associated with the same field (or set of fields) in the packet header. For instance, if the target is to detect whether an IP address (monitored entity) is a bot, and the chosen mechanism is to analyze if the percentage of DNS NXDomain replies (feature) is greater than a given threshold (condition), the flow key to use for accounting is the source IP address when the arriving packet is a DNS query (event), but becomes the destination IP address when the packet is a DNS response (a different event).

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. For instance, if an IP address exhibits a critical percentage of DNS NXDomain replies, hence a bot suspect, we may further track its TCP SYNACK/SYN ratio and determine whether horizontal network scans occur, so as to reinforce our suspicion. And we might then follow up by deriving even more in-depth features, e.g., based on deep packet inspection. But at the same time, we would like to avoid tracking all features for all the possible flows, as this would unnecessarily drain computational resources.

Finally, activities associated to a monitoring task are not all associated to a matching functionality: only measurement and accounting tasks are. Rather, less frequent, but crucial, activities (such as forging and exporting alerts, changing states, setting a mitigation filtering rule, and so on) are based on decisions taken on what we learned so far, and which are hence triggered by 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 4). 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 Our implementation uses d-left hashes for O(1) complexity in managing states. Also, memory consumption can be greatly reduced by opportunistically storing only non-default (e.g. anomalous) states. See the illustrative experiment in Fig. 2 for a rough idea of the attainable saving in the Conficker detection example, use case 1, 2.
(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/protocol 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 take decisions programmed 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.

The obvious compromise in our proposed approach is that (as per the match/action Openflow primitives) new metrics or actions can only be added by directly implementing them over the monitoring devices, thus extending the device capabilities. However, even restricting to our actual StreaMon prototype, the subset of metrics we implemented appears reusable and sufficient to support features proposed in a meaningful set of literature applications - see Table 1 in Section 3.2.

3. STREAMON PROCESSING ENGINE

3.1 System components

The previously introduced abstraction can be concretely implemented by a stream processing engine whose architecture is depicted in Figure 3. It consists of four modular layers descriptively organized into two subsystems, namely Measurement subsystem and Logic subsystem, detailed in sections 3.2 and 3.3, respectively.

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 (see section 3.3), 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.

StreaMon’s programming language

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 de-
Figure 4: Excerpt of XML based StreaMon code showing: (i) metric element allocation, (ii) feature compositions, (iii) an 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.

HW acceleration

StreaMon allows the seamless integration of HW accelerated metrics, e.g. mandated by stringent performance requirements. This remains transparent to the application programmer, which can thus port the same application from a SW based probe to a dedicated HW platform with no changes in the application program. Seamless HW integration is technically accomplished by performing all metrics, parsing, and event matching HW-accelerated computations in a front-end (in our specific case, an FPGA), and by bringing the relevant results up to the user plane through a HW/SW interface, by appending the meta-data generated by the HW tasks to the packet.

3.2 Measurement Subsystem

The StreaMon Measurement subsystem provides a fast, computation/memory efficient, set of n highly configurable built-in modules that allows a programmer to deploy and compute a wide range of traffic features in stream mode.

Multi-Hash Metric module (MH) - From a high level point of view, StreaMon metric modules are functional blocks exporting a simple interface to update and retrieve metric values associated to a given key. Even though in principle such modules can be implemented with any compact data structure compatible to our proposed stream mode approach (and indeed the current architecture support pluggable user defined metric modules), several metrics of practical interest can be derived from a basic structure depicted in Figure 5 and extending the construction proposed in [5]. It permits to count (or time-average, see below) distinct values (called variations in [5]) associated to a same key: for example: the number of distinct IP destinations contacted by a same IP address. The MH module is implemented using Bloom filter extensions [9, 5, 6]. Processing occurs in three stages: (i) extract from the packet two binary strings, Detector FlowKey (DFK) and Monitor FlowKey (MFK), that will be used in to query the subsequent filters; (ii) detect whether the DFK has already appeared in a past time window (Variation Detector filter, VD), and if this is the case, iii) account to the MFK key some packet-related quantity (e.g. number of bytes, or simply add 1 if packet count is targeted) to the third stage’s Variation Monitor (VM) filter. The reader interested in the rationale behind such construction and in further design details may refer to [5]. Finally note that, as a special case, the MH module can be configured to either i) provide an ordinary count, by disabling the VD stage, or ii) perform a match for checking whether some information associated to the incoming packet is first seen (in a given time window), by disabling the VM stage.

Programming the Measurement subsystem -

StreaMon programmers can deploy two types of “counting” data structures: (i) an ordinary Count Sketch that sums a value associated to the MFK (CBF) or (ii) a Time decaying Bloom Filter (TBF) [6] that performs a smoothed (exponentially decaying) average of a value associated to the MFK. At deployment time, an MH module is configured by: (i) enabling/disabling the VD or the VM; (ii) specifying the counting data structure in the VM; (iii) setting the MH parameters, i.e., num-

Figure 5: Metric module structure

Even if, for simplicity of explanation, we account such an extraction to the MH module, for obvious performance reasons we perform packet parsing and the consequent extraction of the flowkeys DFK and MFK once for all, for all metrics, in the Event Layer. Indeed, different metrics may make usage of a same packet field.
| Paper | Description | Brief description of reference features supported by StreaMon |
|-------|-------------|---------------------------------------------------------------|
| 17    | Passive DNS anal. x malicious domain detection | (i) Short life and Access Ratio per domain; (ii) Multi-homing/multi-address per domain; (iii) AVG, STD, total number, CDF of the TTL for a given domain |
| 32    | Traffic charact. via Complex Network modeling | CDF, STD, Max/Min of: (i) total number of endpoints and correspondent host per endpoint; (ii) bytes exchanged between two endpoints |
| 17    | DNS analysis for fastflux domain detection | Arithmetic functions of: (i) number of unique A records returned in all DNS lookups, (ii) number of A records in a DNS response, (iii) number of name server (NS) records in one single lookup, (iv) number of unique ASNs for each A record |
| 33    | Stateful VOIP attack detection | Stateful rules: (i) RTP traffic after SIP BYE? (ii) source IP address changed in time window? (iii) RTP sequence numbers evolve correctly? |
| 26    | DDOS detection through ICMP and L4 analysis | Sent/received TCP packet ratio; Sent/received for different ICMP messages; UDP bitrate; No. different UDP connections per destination; Connection packet count |
| 12    | SVM-based method for P2P traffic classification | Per distinct host and port: (i) ratio between TCP and UDP traffic; (ii) AVG traffic speed; (iii) ratio between TCP and UDP AVG packet length; Per distinct IP: (i) ARP request count, (ii) ratio between sent/received TCP, UDP packet, (iii) ratio between {TCP, UDP} and total traffic; Per distinct port: traffic duration |
| 16    | Mail spammer detection via Network level analysis | (i) AVG/STD msg len in past 24h; (ii) AVG distance to 20 nearest IP neighbors of the sender; (iii) AVG/STD of geodesic distance between sender and recipient; (iv) Message body len; (v) AVG, STD and total no. different recipients |

Table 1

What practical features can be supported? - We believe that the above metrics fulfill the needs of a non negligible fraction of real world monitoring applications. Indeed, 6 shows a concrete example of a real world application reimplemented by just using modules derived from our general MH module described above. Indeed, limiting to the StreaMon’s measurement subsystem (we will significantly extend the framework in the next section), table 1 shows features considered in a number of works taken from a somewhat heterogeneous set (different targets, operating at different network layers, different classification approaches) which, according to our analysis, are readily deployed through suitable configuration of the above metrics/features. Indeed, most applications either require to track/match (1) features that are directly retrievable from a single packet and do not have memory of past values (e.g.: short life of a DNS domain, message body length, all the features in 17 except the first one); and/or (2) require counting or averaging over a various set of parameters (eventually uniquely accounted - e.g. variations), which are readily instantiated with an MH module and/or (3) require logical/mathematical combinations of different statistics, which is the goal of our Feature Layer. Traffic features not covered by the families listed above are those which require a stateful correlation of different flows status. Such metrics (like the ones exploited in 33) are supported by StreaMon but require the stateful framework described in Section 3.3.

3.3 Logic subsystem

StreaMon’s Logic Subsystem is the result of the interwork between the Event Layer and the Decision Layer. It provides the application developer with the ability to customize the tracking logic associated to a monitored entity subject to specific user-defined conditions, so as to provide a verdict and/or perform analyses beyond those provided by the Monitoring Subsystem.

Event Layer - The Event Layer generates the events triggering the StreaMon processing chain, identifies the monitored entity (primary key), and retrieves the specific event context (in particular the flow status). This layer is further composed of three functional component: (i) the capture engine responsible for “capturing” the triggering event, either a “packet arrival” or a “timeout expiration”; (ii) the timeout manager in charge of keeping track of all registered timeouts and manage their expiration; (iii) the status table, a (dleft hash) table storing the status associated to the processed flows - the primary key status - and the so-called secondary

---

4 For instance, the number of bytes exchanged between two hosts can be obtained by setting as VMK is the concatenation of source and destination IP address and as updating quantity the length field of the packet; similarly, unique counts such as the total number of distinct TTL for a given DNS domain is obtained by setting as VDK the concatenation of Domain Name and TTL in the DNS response and as VMK is the Domain Name.
support table, a table storing states not associated to the primary key but required by the application state machine to take some decision.

The triggering event is associated to a primary key, i.e. the monitored entity (flow, host, etc) under investigation for which the monitoring application will provide a final verdict, like “infected” or “legitimate”. For intercepted packets, the primary key is a combination of packet fields parsed by the (extensible) protocol dissector implemented in this layer. For locally generated timeout expiration events, the primary key is retrieved from the timeout context. The primary key is used to retrieve the current status of the flow: no match in the state table is considered to be a default state.

If the primary key is not directly retrievable from the packet, and instead the flow status is related to some other flow previously processed, a secondary support table storing the references to the entries in the status table is used. Such secondary support table is called related status table. For example, this can be the case of an application keeping track of SIP URIs, (i.e.: the primary key is the SIP user ID in SIP INVITE messages) and consider as suspicious all UDP packets related to a data connection with a suspicious SIP user. In case of UDP packets, neither of the packet fields can be used to retrieve the flow status. Instead, a secondary support table is used to keep a reference between the socket 5-tuple and the status entry of the associated SIP initiator user. If the application does not take into account the flow status, the primary extraction is skipped.

**Decision Layer** - The Decision Layer is the last processing unit of the StreaMon architecture and receives the event context carrying an indication of the current flow status and a set of traffic features container. This layer keeps a list of Decision Entries (DEs) defined as the 3-tuple (enabling Condition (C), Output actions (O), State Transition (ST)). For each triggering event, and according to the current flow status, the decision layer verifies the enabling conditions and executes the actions and the state transition associated to the matched condition. Since secondary support tables can be updated, StreaMon support variable conditions, i.e. in which the comparison operands may change during time. The first matched condition will trigger the execution of a action set (like DROP, ALERT, SET_TIMEOUT etc.).

*Programming the Logic Subsystem* - Programmers describe an XFSM specifying (i) states and triggering events; (ii) for each state, which metrics and features are updated and which auxiliary information is collected/processed (secondary table); (iii) which conditions trigger a state transition and the associated actions.

To identify the triggering event, StreaMon keeps a list of user-defined “event descriptors”, expressed as a 3-tuple (type, descriptor, primary key). For example, in case of intercepted packets, an event can be defined as: \((\text{packet;} \{\text{udp.dport} == 53; \text{ip.src}\})\).

Moreover, programmers are given primitives to define for each event and state: (i) a set of metric operations (MOP) as described in section 3.2; (ii) a set of conditions expressed as an arithmetic function of features and secondary support table values. For each condition, programmers define a set of built-in actions and a state transition. Table 2 summarizes the logic subsystem supported condition operators and actions.

### 4. 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 traces captured in our campus network (use case 4.1 and 4.2) or (ii) synthetic traces properly generated in our laboratories (use case 4.3).

#### 4.1 P2P traffic classification

This example shows how straightforward is the implementation of three transport layer traffic features described in [20], 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): $E_1 : i f (i p . p r o t o == U D P ) \& \& (u d p . p o r t \neq 25, 53, 110, \ldots)$ $E_2 : i f (i p . p r o t o == T C P ) \& \& (t c p . p o r t \neq 25, 53, 110, \ldots)$

This application extracts the following traffic features $F_1 = M_1 \&\& M_2; \quad F_2 = |M_3 - M_4|$ 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\((M_1, i p . s r c | i p . d s t)\); | get\((M_1, i p . s r c | i p . d s t)\); |
| get\((M_2, i p . s r c | i p . d s t)\); | set\((M_2, i p . s r c | i p . d s t)\); |
| get\((M_3, i p . d s t)\); | set\((M_3, t c p . s p o r t | i p . d s t, i p . d s t)\); |
| get\((M_4, i p . d s t)\) | set\((M_4, i p . s r c | i p . d s t, i p . d s t)\); |
| get\((M_i, i p . s r c)\), $i = 3, 4$ | get\((M_i, i p . s r c)\), $i = 3, 4$ |
The application detects a p2p client if the following condition holds after a transitory period: \( F_1 = 1 \&\& (F_2 < 10) \).

### 4.2 Conficker botnet detection

Conficker is one of the largest Botnets found in recent years \cite{29}. A multi-step detection algorithm can attempt to track the two following (separated) phase: (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 NNIDomain 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 NNIDomain 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 NNIDomain 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 6a. Figure 7 shows the trend of features used in this configuration, for an 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.

### 4.3 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 Figure 6b, and is governed by the following two events:

\[ E_1 : \text{if } (ip.proto == TCP) \&\& (tcp.flags == SYN) \]

\[ E_2 : \text{timeoutexpired} \]

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 \( f(F_1(t) > 1.2 \cdot F_1(t-1)) \), 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 each host contacts each destination IP addresses (\( M_2 \) is configured with UD disabled and VM of type TBF with smoothing window 240s, and it is updated with MFK \((IP_{src}, IP_{dst})\)).

Figure 8 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 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).

### 4.4 HW accelerated detection of non standard encrypted traffic

This example shows how HW metrics can be integrated in StreaMon. Since one of the task performed during deep packet inspection is the collection of statistics on byte frequencies, offsets for common byte-values, packet information entropy, we illustrate an use case that is a simplified version of the approach described in [14], in which encrypted flows are detected by combining two traffic features: (i) the bit information entropy of a packet; (ii) the percentage of printable characters, i.e. ASCII characters in the range \([32 \ldots 127]\). Both features are computationally demanding for a software implementation, whereas they are best delegated to an FPGA implementation (just 250 logic cells for our implementation).

This use case implements a simple stateless StreaMon application with the following characteristics: (i) it considers as event the reception of a UDP or TCP packet with length greater than 100 bytes; (ii) the primary key is the socket 5-tuple of the packet; (iii) it makes use of two FPGA precomputed metrics: popcount (number of bit set to 1) \( M_1 \) and printable chars percentage \( M_2 \) (which is directly mapped into a StreaMon feature \( F_1 = M_1 \)); (v) the packet entropy Feature \( F_2 \) is obtained as \( -\frac{1}{N} \sum_{i=0}^{N} n_i \cdot (\log(n_i) - \log(N)) \), where \( N = \text{len}, n_0 = N - M_1, n_1 = M_1 \). An encrypted flow is detected if: \( (F_2 < 0.75) \&\&( -3\sigma < F_1 - H_U(l) < 3\sigma) \)

where \( H_U(l) \) is the entropy of a packet with uniform distribution of 1 in the payload and length \( l \) and \( \sigma \) is the standard deviation. Figure 9 shows \( M_1 \) and \( M_2 \) to the simple case of HTTP (unencrypted flow) and SSL (encrypted flow) traffic, and as in [14] justify the condition expressed above.

### 4.5 Port knocking

![Figure 8: 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](image)

![Figure 9: (a) percentage of printable character as function of the percentage of packets (b) percentage of bit set to 1 as function of the percentage of packets](image)
This use case shows how a stateful firewall application can be easily implemented with StreaMon. In particular, this use case implements a port knocking mechanism to enable SSH access. The application’s XFSM is depicted in figure 10. SSH access will be granted only to those clients guessing the correct port sequence 5000, 4000, 7000 (a short sequence only for presentation convenience).

The application considers four events (TCP SYN received respectively on port 5000, 4000, 7000, other ports) and four states (default, 5000 contacted, 4000 contacted, allowed). A state transition is triggered only if the client (identified by the IP source address) contacts the expected port in the sequence. If after a state transition the next expected port is not contacted in five seconds, the flow status is rolled back to default.

In addition, to avoid random port scanning for guessing the correct sequence, a metric $M_1$ (VD=ON, VM=TFB with smoothing window 5s) counting the SYN rate is used. If a port scan is detected ($if (F_1 = M_1 > 40)$) the flow status is updated to "attack", the host is blocked for 2 minutes and an alert is generated. $M_1$ is updated with $DFK = ip.src|ip.dst|tcp.dport$ and $MKF = ip.src$ when a SYN to an unexpected port is contacted.

5. PERFORMANCE EVALUATION

5.1 Implementation Overview

We implemented StreaMon on two platforms. A SW-only deployment leverages off-the-shelf NIC drivers, integrating the efficient PFQ packet capturing kernel module [8]. To test HW acceleration, we also deployed StreaMon using a 10 gbps Combo FPGA card as packet source. In what follows, unless otherwise specified (specifically, section 5.2.3 which deals with HW accelerated metrics implemented over the Combo FPGA card) we focus on the SW-only configuration.

StreaMon start-up sequence is summarized as follows. The application program is given as input to a preprocessor 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".

StreaMon implementation takes advantages from multicore platforms by implementing parallel processing chains as shown in the implementation architecture depicted in figure 11 (for the SW-only setup). The PFQ driver 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.

5.2 Performance Analysis

5.2.1 Measurement subsystem performance

Figure 12 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:

| Metric Number | CPU Core | Throughput |
|---------------|----------|------------|
| 1             | 1        | 6,345 Gbps |
| 4             | 2        | 6,254 Gbps |
| 8             | 3        | 6,123 Gbps |
| 16            | 4        | 6,002 Gbps |
| 24            | 6        | 5,881 Gbps |

Note that this is an optimization step which is devised 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.
Table 3 shows the details of the test traces used for three among the use cases presented in Section 4 and their performance evaluation, where \( \text{pkt. no} \) is the total number of packets, \( \text{AVG plen} \) is the average packet length, \( M \) is the total number of metrics, \( FK \) is that the transmission bitrate is fixed by the injector probe and depends on the average packet length.

As already underlined, the use case applications experience a better throughput with respect to the bitrate obtained with the same number of metrics in the worst case measurement showed in [12]. Note that due to the different performances of the COMBO card processor and the limitation to single core processing, the encrypted flow detection use case performance is omitted to avoid confusion, as it would present results not directly comparable with the remaining ones.

### 5.2.3 FPGA accelerated primitives

As already discussed, our current StreaMon prototype supports seamless integration of HW precomputed primitives, and in particular we have implemented packet entropy computation (see section 4.4) and event parsing on an INVEA-TECH Combo FPGA card [1]. The interface between the Combo card and StreaMon core software is realized via a footer appended to the ethernet frame. The Combo card capture the packet traveling in the network under inspection, performs the required operation and transfer the resulting metrics and signals as a list of Type Len Value (TLV) to the PC hosting the capture card. This list of TLV is parsed by the Event Layer and will be available to the remaining layers.

Our next step is to implement a full deep packet inspection module. To confirm its necessity, we have so far implemented a simplified content matching packet payload inspection primitive in both SW and HW. Figure 13 compares the two implementations for a toy example of just 32 content strings, showing the severe performance degradation of the SW implementation even in such small scale example. At the same time, the figure shows that, by offloading content matching primitives to the HW front-end, our prototype achieves almost optimal throughput performance.

### 6. RELATED WORK

In the literature, several monitoring platforms have targeted monitoring applications’ programmability. A Monitoring API for programmable HW network adapters is proposed in [31]. On top of such probe, network administrators may implement custom C++ monitoring applications. One of the developed applications is AppMon [4]. It uses deep packet inspection to classify observed flows and attribute these flows to an application.

---

**Table 3:**

| pkt no. | AVG pkt len | Host no. | TX rate |
|---------|-------------|----------|---------|
| 63209928 | 632.82 bytes | 31827 | 6.47 Gbps |

where \( \text{pkt len} \) and \( \text{TX} \) 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:

1. 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;
2. all flow keys are different (\( n \) metrics, \( n \times 2 \) 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   |

Without discussing here the advantages of a stateful pre-filtering (which will be shown in section 4.2.2), we want to underline that, in the typical case in which not all the metrics are updated for each packet and require distinct flow keys, we can experience a better relation between the processing throughput and the number of metrics. For this reason, we compared the average throughput obtained by executing the traffic classification algorithm described in [32] (the most computational demanding algorithm cited in [1]) in terms of number of distinct metrics and flow keys, and portion of processed traffic. The following table shows the average bitrate (in Gbps) obtained by [32] in case of TCP traffic classification ([32]: 6 metrics, 3 flowkeys) and the average bitrate of the worst case configuration (6 metrics and 12 flow keys).

|       | 1 core | 2 cores | 3 cores |
|-------|--------|---------|---------|
| worst case | 1.243  | 2.812   | 2.008   |

---

5.2.2 **Use case performance evaluation**
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 [19] 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 [27] 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).

The Real-Time Communications Monitoring (RTC-Mon) framework [15] 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 [21], FLAME [3] and Blockmon [18] 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. On a different line, a number of monitoring frameworks are based on suitable extensions of Data Stream Management Systems (DSMS). PaQueT [23], and more recently BackStreamDB [24], are programmable monitoring frameworks developed as an extension of the Borealis DSMS [2]. Ease of programming and high flexibility is provided by permitting users to define new traffic metrics by simply performing queries to the DSMS. The DSMS is configured through an XML file that is processed to obtain a C++ application code. Gigascope [10] is another stream database for network monitoring that provide an architecture programmable via SQL-like queries.

Opensketch [34] 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 [35] 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 [28]), 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 [30].

7. 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

| case | pkts.     | AVG plen [bytes] | Ms, FKs. | TX [Gbps] | RX [# core: Gbps] |
|------|-----------|------------------|----------|-----------|-------------------|
| 4.1  | 7676958   | 404.97           | 8, 4     | 3.658     | 1: 1.956, 2: 2.954, 3: 3.654 |
| 4.2  | 11852112  | 329.77           | 4, 2     | 2.462     | 1: 2.462          |
| 4.3  | 9387240   | 215.38           | 2, 2     | 2.067     | 1: 2.003, 2: 2.067 |

Table 3: Use case trace parameters and throughput
of features accounted in the proposed use cases suggest that StreamMon’s flexibility can be exploited to develop and deploy several real world applications.

8. REFERENCES

[1] COMBO product brief. http://www.inve-tech.com/data/combo/combo_pb_en.pdf.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In In CIDR, pages 277–289, 2005.

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

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

[5] G. Bianchi, E. Boschi, S. Teofili, and B. Trammell. Measurement data reduction through variation rate metering. In INFOCOM, pages 2187–2195, 2010.

[6] G. Bianchi, N. d’Heureuse, and S. Niccolini. On-demand time-decaying bloom filters for telemarketer detection. ACM SIGCOMM Comput. Commun. Rev., 41(5):5–12, 2011.

[7] L. Bilge, E. Kirda, C. Kruegel, and M. Balducci. EXPOSURE: Finding malicious domains using passive DNS analysis. In NDSS 2011, 2011.

[8] 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.

[9] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In Internet Mathematics, pages 636–646, 2002.

[10] C. Cranor, T. Johnson, and O. Spatasek. Gigascope: a stream database for network applications. In SIGMOD, pages 647–651, 2003.

[11] A. Dainotti, A. Pescape, and K. Claffy. Issues and future directions in traffic classification. IEEE Network, 26(1):35–40, 2012.

[12] H. Deng, A. M. Yang, and Y. D. Liu. P2p traffic classification method based on svn. Comp. Eng. and Applications, 44(14):122, 2008.

[13] L. Deri, F. Fusco, and J. Gasparakis. Towards monitoring programmability in future internet: Challenges and solutions. In Trustworthy Internet, pages 249–259. Springer, 2011.

[14] P. Dorfinger, G. Panholzer, B. Trammell, and T. Pepe. Entropy-based traffic filtering to support real-time skype detection. In Proc. 6th Int. Wireless Commun. and Mobile Computing Conf. (IWCMC ‘10), 2010.

[15] 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.

[16] S. Hao, N. A. Syed, N. Feamster, A. G. Gray, and S. Krasser. Detecting spammers with snare: Spatio-temporal network-level automatic reputation engine. In USENIX Sec. Symp., pages 101–118, 2009.

[17] T. Holz, C. G orecki, K. Rieck, and F. C. Freiling. Measuring and detecting fast-flux service networks. In NDSS, 2008.

[18] 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.

[19] G. Iannaccone, C. Diet, D. McAuley, A. Moore, I. Pratt, and L. Rizzo. The como white paper. 2004.

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

[21] 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.

[22] F. Leder and T. Werner. Know Your Enemy: Containing Conficker, To Tame a Malware. Technical report, The Honeynet Project, Apr. 2009.

[23] N. Ligocki and C. H. C. Lyra. A flexible network monitoring tool based on a data stream management system. In ISCC 2008, pages 800–805, 2008.

[24] C. Lyra, C. Harai, and J. Duarte, EliasP. Backstreamdb: A distributed system for backbone traffic monitoring providing arbitrary measurements in real-time. In Passive and Active Measurement (PAM), pages 42–52, 2012.

[25] 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.

[26] J. Mirkovic, G. Prier, and P. L. Reiher. Attacking ddos at the source. In ICNP, pages 312–321, 2002.

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

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

[29] 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.

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

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

[32] X. Wu, K. Yu, and X. Wang. On the growth of internet application flows: A complex network perspective. In INFOCOM, 2011.

[33] Y.-S. Wu, S. Bagchi, S. Garg, N. Singh, and T. K. Tsai. Scidive: A stateful and cross protocol intrusion detection architecture for voice-over-ip environments. In DSN, pages 433–442, 2004.

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

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