Stream-Oriented Network Traffic Capture and Analysis for High-Speed Networks

Antonis Papadogiannakis*, Michalis Polychronakis†, Evangelos P. Markatos*

*Institute of Computer Science, Foundation for Research and Technology – Hellas
†Computer Science Department, Columbia University
papadog@ics.forth.gr, mikepo@cs.columbia.edu, markatos@ics.forth.gr

***Abstract*—Intrusion detection, traffic classification, and other network monitoring applications need to analyze the captured traffic beyond the network layer to allow for connection-oriented analysis, and achieve resilience to evasion attempts based on TCP segmentation. Existing network traffic capture frameworks, however, provide applications with raw packets and leave complex operations like flow tracking and TCP stream reassembly to application developers. This gap, between what applications need and what systems provide, leads to increased application complexity, longer development time, and most importantly, reduced performance due to excessive data copies between the packet capture subsystem and the stream processing module.**

This paper presents the *Stream capture library (Scap)*, a network monitoring framework built from the ground up for stream-oriented traffic processing. Based on a kernel module that directly handles flow tracking and TCP stream reassembly, Scap delivers to user-level applications flow-level statistics and reassembled streams by minimizing data movement operations and discarding uninteresting traffic at early stages, while it inherently supports parallel processing on multi-core architectures, and uses advanced capabilities of modern network cards. Our experimental evaluation shows that Scap can capture all streams for traffic rates two times higher than other stream reassembly libraries. Finally, we present the implementation and performance evaluation of four popular network traffic monitoring applications built on top of Scap.

***Index Terms*—Traffic Monitoring; Stream Reassembly; Packet Capture; Packet Filtering; Overload Control; Performance**

I. Introduction

Passive network monitoring is an indispensable mechanism for increasing the security and understanding the performance of modern networks. For example, Network-level Intrusion Detection Systems (NIDS) inspect network traffic to detect attacks [1], [2] and pinpoint compromised computers [3], [4]. Similarly, traffic classification tools inspect network traffic to identify different communication patterns and spot potentially undesirable traffic [5], [6]. To make meaningful decisions, these monitoring applications need to analyze network traffic at the transport layer and above. For instance, NIDS reconstruct transport-layer streams to detect attack vectors that span multiple packets, and avoid evasion attacks [7]–[10].

Unfortunately, there is a *gap between monitoring applications and underlying traffic capture tools*: Applications increasingly need to reason about higher-level entities and constructs such as TCP flows, HTTP headers, SQL arguments, email messages, and so on, while traffic capture frameworks still operate at the lowest possible level: they provide the raw—possibly duplicate, out-of-order, or overlapping—and in some cases even irrelevant packets that reach the monitoring interface [11]–[13]. Upon receiving the captured packets at user space, monitoring applications usually perform TCP stream reassembly using existing libraries [14] or custom stream reconstruction engines [1], [2]. This results in additional memory copy operations for extracting the payloads of TCP segments and merging them into larger "chunks" in contiguous memory. Moreover, this misses several optimization opportunities, such as the early discarding of uninteresting packets before system resources are spent to move them to user level, and assigning different priorities to transport-layer flows so that they can be handled appropriately at lower system layers.

To bridge this gap and address the above concerns, we present the *Stream capture library (Scap)*: a unified passive network monitoring framework built around the abstraction of the *Stream*, which is elevated into a first-class object handled by user applications. Designed from the beginning for stream-oriented network monitoring, Scap (i) provides the high-level functionality needed by network monitoring applications, and (ii) implements this functionality at the most appropriate place: at user level, at kernel level, or even at the network interface card. On the contrary, existing TCP stream reassembly implementations are confined, by design, to operate at user level and, therefore, are deprived from a rich variety of efficient implementation options.

To enable aggressive optimizations, we introduce the notion of *stream capture*: that is, we elevate the *Stream* into a first-class object that is captured by Scap and handled by user applications. Although previous work treats TCP stream reassembly as a necessary evil [15], used mostly to avoid evasion attacks against intrusion detection and other monitoring systems, we view streams—not packets—as the fundamental abstraction that should be exported to network monitoring applications, and as the right vehicle for the monitoring system to implement aggressive optimizations all the way down to the operating system kernel and network interface card.

To reduce the overhead of unneeded packets, Scap introduces the notion of *subzero packet copy*. Inspired by zero-copy approaches that avoid copying packets from one main memory location to another, Scap not only avoids redundant packet copies, but also avoids bringing some packets in main memory

Manuscript received December 31, 2013; revised May 27, 2014.
This paper was presented in part at the ACM SIGCOMM Internet Measurement Conference (IMC), Barcelona, Spain, 2013.

in the first place. We show several cases of applications that are simply not interested in some packets, such as the tails of large flows [16]–[20]. Subzero packet copy identifies these packets and does not bring them in main memory at all: they are dropped by the network interface card (NIC) *before* reaching the main memory.

To accommodate heavy loads, Scap introduces the notion of *prioritized packet loss* (PPL). Under heavy load, traditional monitoring systems usually drop arriving packets in a random way, severely affecting any following stream reassembly process. However, these dropped packets and affected streams may be important for the monitoring application, as they may contain an attack or other critical information. Even carefully provisioned systems that are capable of handling full line-rate traffic can be overloaded, e.g., by a sophisticated attacker that sends adversarial traffic to exploit an algorithmic complexity vulnerability and intentionally overload the system [21], [22]. Scap allows applications to (i) define different priorities for different streams and (ii) configure threshold mechanisms that give priority to new and small streams, as opposed to heavy tails of long-running data transfers.

Scap provides a flexible and expressive Application Programming Interface (API) that allows programmers to configure all aspects of the stream capture process, perform complex per-stream processing, and gather per-flow statistics with a few lines of code. Our design introduces two novel features: (i) it enables the early discarding of uninteresting traffic, such as the tails of long-lived connections that belong to large file transfers, and (ii) it offers more control for tolerating packet loss under high load through stream priorities and best-effort reassembly. Scap also avoids the overhead of extra memory copies by optimally placing TCP segments into stream-specific memory regions, and supports multi-core systems and network adapters with receive-side scaling [23] for transparent parallelization of stream processing.

We have evaluated Scap in a 10GbE environment using real traffic and show that it outperforms existing alternatives like Libnids [14] and Snort's stream reassembly [1]. Our results demonstrate that Scap can capture and deliver at user level all streams with low CPU utilization for rates up to 5.5 Gbit/s using a single core, while Libnids and Snort start dropping packets at 2.5 Gbit/s due to high CPU utilization for stream reassembly at user level. A more detailed evaluation of Scap can be found in our previous work [24].

We have implemented four popular network monitoring applications on top of Scap: (i) flow export, (ii) NIDS signature matching, (ii) Layer-7 traffic classification, and (iv) HTTP protocol parsing and analysis. All these applications require flow tracking, and most of them also require stream reassembly and protocol normalization to be accurate. Thus, they can significantly benefit from the features and performance optimizations offered by Scap. We present in detail the implementation and performance characteristics of these applications using Scap.

In summary, the main contributions of this paper are:

- We identify a semantic gap: modern network monitoring applications need to operate at the transport layer and beyond, while existing monitoring systems operate at the network layer. To bridge this gap and enable aggressive optimizations, we introduce the notion of *stream capture* based on the fundamental abstraction of the *Stream*.
- We introduce *subzero packet copy*, a technique that takes advantage of filtering capabilities of commodity NICs to not only avoid copying uninteresting packets across different memory areas, but to avoid bringing them in main memory altogether.
- We introduce *prioritized packet loss*, a technique that enables graceful adaptation to overload conditions by dropping packets of lower priority streams, and favoring packets that belong to recent and shorter streams.
- We describe the design and implementation of Scap, a framework that incorporates the above features in a kernel-level, multicore-aware subsystem, and provides a flexible and expressive API for building stream-oriented network monitoring applications.
- We experimentally evaluate Scap and demonstrate that it can capture and deliver transport-layer streams for traffic rates two times higher than previous approaches.
- We present the implementation and evaluation of four real-world network monitoring applications using Scap.

The rest of the paper is organized as follows: in Section II we present the design and basic features of Scap, while in Section III we outline the main Scap API calls. Then, Section IV describes the high-level architecture of Scap, and Section V discusses implementation details. In Section VI we experimentally evaluate the performance of Scap, comparing with existing libraries while replaying real network traffic captured in the wild. In Section VII we present the implementation and performance of four real-world network traffic analysis applications using the Scap framework. Finally, Section VIII summarizes related work, and Section IX concludes the paper.

II. Design and Features

A. Subzero-Copy Packet Transfer

Several network monitoring applications [16]–[20] need to analyze only the first bytes of each connection, especially under high traffic load. In this way, they analyze the more useful (for them) part of each stream and discard a significant percentage of the total traffic [17]. For such applications, Scap has incorporated the use of a *cutoff* threshold that truncates streams to a user-specified size, and discards the rest of the stream (and the respective packets) within the OS kernel or even the NIC, avoiding unnecessary data transfers to user space. Applications can dynamically adjust the cutoff size *per stream*, allowing for greater flexibility.

Besides a stream cutoff size, monitoring applications may need to efficiently discard other types of less interesting traffic. Many applications often use BPF filters [12] to define which streams they want to process, while discarding the rest. In case of an overload, applications may want to discard traffic from low priority streams or define a stream *overload cutoff* [18], [19]. Also, depending on the application, packets belonging to non-established TCP connections or duplicate packets may be discarded. In all such cases, Scap can discard the appropriate packets at an early stage within the kernel, while in many cases packets can be discarded even earlier at the NIC.

To achieve this, Scap capitalizes on modern network interfaces that provide filtering facilities directly in hardware. For example, Intel's 82599 10G interface [25] supports up to 8K perfect match and 32K signature (hash-based) Flow Director filters (FDIR). These filters can be added and removed dynamically, within no more than 10 microseconds, and can match a packet's source and destination IP addresses, source and destination port numbers, protocol, and a flexible 2-byte tuple anywhere within the first 64 bytes of the packet. Packets that match an FDIR filter are directed to the hardware queue specified by the filter. If this hardware queue is not used by the system, the packets will be just dropped at the NIC layer, and they will never be copied to the system's main memory [26]. When available, Scap uses FDIR filters to implement all above types of early packet discarding, otherwise packets are discarded within the OS kernel.

B. Prioritized Packet Loss

Scap introduces *Prioritized Packet Loss* (PPL) to enable the system to invest its resources effectively during overloads. This is necessary because sudden traffic bursts or overload conditions may force the packet capturing subsystem to fill up its buffers and start dropping packets in a haphazard manner. Even worse, attackers may intentionally overload the monitoring system while an attack is in progress so as to evade detection. Previous research has shown that being able to handle different flows [21], [27], [28], or different parts of each flow [18], [19], in different ways can enable the system to invest its resources more effectively and significantly improve detection accuracy. PPL enables user applications to define the priority of each stream so that under overload conditions packets from low-priority streams are the first to go. User applications can also define a threshold for the maximum stream size under overload (*overload_cutoff*). Then, packets exceeding beyond this threshold are the ones to be dropped.

As long as the percentage of used memory is below a user-defined threshold, called $base\_threshold$, PPL drops no packets. When, however, used memory exceeds the $base\_threshold$, PPL kicks in: it first divides the memory above $base\_threshold$ into n (equal to the number of used priorities) regions using $n + 1$ equally spaced watermarks (i.e., $watermark_0$, $watermark_1$, ..., $watermark_n$), where $watermark_0 = base\_threshold$ and $watermark_n = memory\_size$. When a packet belonging to a stream with the i_{th} priority level arrives, PPL checks the percentage of memory used by Scap at that time. If it is above $watermark_i$, the packet is dropped. Otherwise, if the percentage of used memory is between $watermark_i$ and $watermark_{i-1}$, PPL makes use of the *overload_cutoff* (if it has been defined). Then, if the packet is located in its stream beyond the *overload_cutoff* byte, it is dropped. This way, high-priority, newly created, and short streams will be accommodated with higher probability.

C. Flexible Stream Reassembly

To support monitoring at the transport layer, Scap provides different modes of TCP stream reassembly. The two main objectives of stream reassembly in Scap are: (i) to provide transport-layer reassembled chunks in continuous memory regions, and (ii) to perform protocol normalization [8], [29]. Scap currently supports two TCP stream reassembly modes: `SCAP_TCP_STRICT` and `SCAP_TCP_FAST`. In the strict mode, streams are reassembled according to existing guidelines [7], [29], offering protection against evasion attempts based on IP/TCP fragmentation. In the fast mode, streams are reassembled in a *best-effort* way, offering resilience against packet loss caused in case of overloads. In this mode, Scap follows the semantics of the strict mode as closely as possible, e.g., by handling TCP retransmissions, out-of-order packets, and overlapping segments. To accommodate for lost segments, however, stream data is written without waiting for the correct next sequence number to arrive. In that case, Scap sets a flag to report that errors occurred during reassembly.

Moreover, Scap supports different reassembly policies, e.g., according to different operating systems. Scap applications can set a different reassembly policy per stream. This is motivated by previous works, which have shown that the reconstructed data stream in a NIDS may differ from the actual data stream observed at the destination, due to disparities in different TCP implementations, e.g., when handling overlapping segments. Attackers may exploit such differences to evade detection [9].

Scap also supports UDP: a UDP stream is the concatenation of the packet payloads of a given UDP flow. For other protocols without sequenced delivery, Scap returns each packet for processing without any reassembly.

D. Parallel Processing and Locality

Scap has inherent support for multi-core systems, hiding from the programmer the complexity of creating and managing multiple processes or threads. This is achieved by transparently creating a number of worker threads for user-level stream processing (typically) equal to the number of the available cores. Scap also dedicates a kernel thread on each core for handling packet reception and stream reassembly. Each stream is assigned to one pair of kernel and worker threads running on the same core, reducing this way context switches, cache misses [30], [31], and inter-thread synchronization operations. The kernel and worker threads on each core communicate through shared memory and events: a new event for a stream is created by the kernel thread and is handled by the worker thread using a user-defined callback function for stream processing.

To balance the network traffic load across multiple NIC queues and cores, Scap uses both static hash-based approaches, such as Receive Side Scaling (RSS) [23], and dynamic load balancing approaches, such as flow director filters (FDIR) [25]. This provides resiliency against short-term load imbalances that could adversely affect application performance.

E. Performance Optimizations

In case that multiple applications running on the same host monitor the same traffic, Scap provides all of them with a shared copy of each stream. Thus, stream reassembly is performed only once within the kernel, instead of multiple times

Table I
DATA FIELDS OF THE STREAM_T STREAM DESCRIPTOR.

Data field	Description
stream_hdr **hdr;**	*Stream header*
uint32_t **src_ip, dst_ip;**	Source/Destination IP address
uint16_t **src_port, dst_port;**	Source/Destination port
uint8_t **protocol;**	Protocol
uint8_t **direction;**	Stream direction
stream_stats **stats;**	*Stream statistics*
struct timeval **start, end;**	Beggining/end time
uint64_t **bytes,**	Total bytes,
bytes_dropped,	dropped bytes,
bytes_discarded,	discarded bytes,
bytes_captured;	captured bytes
uint32_t **pkts,**	Total packets,
pkts_dropped,	dropped packets,
pkts_discarded,	discarded packets,
pkts_captured;	captured packets
	Other fields
uint8_t **status;**	Stream status
uint8_t **error;**	Error flags
char *__data;__	Pointer to last chunk's data
int **data_len;**	Data length of the last chunk
stream_t *__opposite;__	Stream in the opposite direction
int **cutoff;**	Stream's cutoff
int **priority;**	Stream's priority
int **chunk_size;**	Stream's chunk size
int **chunks;**	Stream's total chunks
int **processing_time;**	Stream's processing time
void *__user_state;__	User state

for each user-level application. If applications have different configurations, e.g., for stream size cutoff or BPF filters, Scap takes a best effort approach to satisfy all requirements.

Performing stream reassembly in the kernel also offers significant advantages in terms of cache locality. Existing user-level TCP stream reassembly implementations receive packets of different flows highly interleaved, which results in poor cache locality [32]. In contrast, Scap provides user-level applications with reassembled streams instead of randomly interleaved packets, allowing for improved memory locality and reduced cache misses.

III. SCAP API

Scap is based around the abstraction of the *stream*: a reconstructed session between two endpoints defined by a 5-tuple (protocol, source and destination IP address, source and destination port). Applications receive a unique stream descriptor `stream_t` for each new stream, which can be used to access all information, data, and statistics about a stream, and is provided as a parameter to all stream manipulation functions. Table I presents the main `stream_t` fields and Table II lists the main functions provided by the Scap API.

A. Initialization

An Scap program begins with the creation of an Scap socket using `scap_create()`, which specifies the interface to be monitored. Programmers can also specify various properties, such as the memory size of the buffer for storing stream data, the stream reassembly mode, and whether the application needs to receive the individual packets of each stream. Upon successful creation, the returned `scap_t` descriptor is used for all subsequent configuration operations. These include setting a BPF filter [12] to receive a subset of the traffic, cutoff values for different stream classes or stream directions, the number of worker threads for balancing stream processing among the available cores, the chunk size, the overlap size between subsequent chunks, and an optional timeout for delivering the next chunk. The `overlap` argument is used when some of the last bytes of the previous chunk are also needed in the beginning of the next chunk, e.g., for matching patterns that might span consecutive chunks. The `flush_timeout` parameter can be used to deliver for processing a chunk smaller than the chunk size after this timeout passes, in case a user needs to ensure timely processing.

B. Stream Processing

Scap allows programmers to write and register callback functions for three different types of events: stream creation, the availability of new stream data, and stream termination. When a stream is created or terminated, or when enough data has been captured for a stream's chunk processing, a new event is triggered and the respective callback is executed. Each callback function takes as a single argument a `stream_t` descriptor `sd`, which corresponds to the stream that triggered the event. As shown in Table I, this descriptor provides access to detailed information about the stream, such as the stream's IP addresses, port numbers, protocol, and direction, as well as useful statistics such as byte and packet counters for all, dropped, discarded, and captured packets, and the timestamps of the first and last packet of the stream. Among the rest of the fields, the `sd->status` field indicates whether the stream is active or closed (by TCP FIN/RST or by inactivity timeout), or if its stream cutoff has been exceeded, and the `sd->error` field indicates stream reassembly errors, such as incomplete TCP handshakes or invalid sequence numbers. Other properties include a stream's cutoff, priority, and chunk size, while there is also a pointer to the `stream_t` of the opposite direction. The `user_state` field can be used by the application to store any per-stream state, as needed.

The stream processing callback can access the last chunk's data and its size through the `sd->data` and `sd->data_len` fields. In case no more data is needed, `scap_discard_stream()` can notify the Scap core to stop collecting data for that stream. Chunks can be efficiently merged with following ones using `scap_ keep_chunk()`. In the next invocation, the callback will receive a larger chunk consisting of both the previous and the new one. Through the stream descriptor, applications can set a stream's priority, cutoff, and other parameters such as the chunk size, overlap size, flush timeout, and reassembly mode.

In case they are needed by an application, individual packets can be delivered using `scap_next_stream_packet()`. Packet delivery is based on the chunk's data and metadata kept by Scap's packet capture subsystem for each packet. Based on this metadata, even reordered, duplicate, or packets with overlapping sequence numbers can be delivered in the same

Table II
THE MAIN FUNCTIONS OF THE SCAP API.

Scap Function Prototype	Description
`scap_t *scap_create(const char *device, int memory_size, int reassembly_mode, int need_pkts)`	Creates an Scap socket
`int scap_set_filter(scap_t *sc, char *bpf_filter)`	Applies a BPF filter to an Scap socket
`int scap_set_cutoff(scap_t *sc, int cutoff)`	Changes the default stream cutoff value
`int scap_add_cutoff_direction(scap_t *sc, int cutoff, int direction)`	Sets a different cutoff value for each direction
`int scap_add_cutoff_class(scap_t *sc, int cutoff, char* bpf_filter)`	Sets a different cutoff value for a subset of the traffic
`int scap_set_worker_threads(scap_t *sc, int thread_num)`	Sets the number of threads for stream processing
`int scap_set_parameter(scap_t *sc, int parameter, int value)`	Changes defaults: inactivity_timeout, chunk_size, overlap_size, flush_timeout, base_threshold, overload_cutoff
`int scap_dispatch_creation(scap_t *sc, void (*handler)(stream_t *sd))`	Registers a callback routine for handling stream creation events
`int scap_dispatch_data(scap_t *sc, void (*handler)(stream_t *sd))`	Registers a callback routine for processing newly arriving stream data
`int scap_dispatch_termination(scap_t *sc, void (*handler)(stream_t *sd))`	Registers a callback routine for handling stream termination events
`int scap_start_capture(scap_t *sc)`	Begins stream processing
`void scap_discard_stream(scap_t *sc, stream_t *sd)`	Discards the rest of a stream's traffic
`int scap_set_stream_cutoff(scap_t *sc, stream_t sd, int cutoff)`	Sets the cutoff value of a stream
`int scap_set_stream_priority(scap_t *sc, stream_t *sd, int priority)`	Sets the priority of a stream
`int scap_set_stream_parameter(scap_t *sc, stream_t *sd, int parameter, int value)`	Sets a stream's parameter: inactivity_timeout, chunk_size, overlap_size, flush_timeout, reassembly_mode
`int scap_keep_stream_chunk(scap_t *sc, stream_t *sd)`	Keeps the last chunk of a stream in memory
`char *scap_next_stream_packet(stream_t *sd, struct scap_pkthdr *h)`	Returns the next packet of a stream
`int scap_get_stats(scap_t *sc, scap_stats_t *stats)`	Reads overall statistics for all streams seen so far
`void scap_close(scap_t *sc)`	Closes an Scap socket

order as captured. This allows Scap to support packet-based processing along with stream-based processing, e.g., to allow the detection of TCP attacks such as ACK splitting [33]. The only difference between Scap's packet delivery and packet-based capture systems is that packets from the same stream are processed together. As an added benefit, such flow-based packet reordering has been found to significantly improve cache locality [32].

The stream's processing time and the total number of processed chunks are available through the `sd->processing_time` and `sd->chunks` fields. This enables the identification of streams that are being processed with very low rates and delay the system, e.g., due to algorithmic complexity attacks [21], [22]. Upon the detection of such a stream, the application can handle it appropriately, e.g., by discarding it or reducing its priority, to ensure that adversarial traffic will not affect the correct operation of the application.

IV. ARCHITECTURE

A. Kernel-level and User-level Support

Scap consists of two main components: a loadable kernel module and a user-level API stub, as shown in Figure 1. Applications communicate through the Scap API stub with the kernel module to configure the capture process and receive monitoring data. Configuration parameters are passed to the kernel through the Scap socket interface. Accesses to `stream_t` records, events, and actual stream data are handled through shared memory. For user-level stream processing, the stub receives events from the kernel module and calls the respective callback function for each event.

The overall operation of the Scap kernel module is depicted in Figure 2. Its core is a software interrupt handler that receives packets from the network device. For each packet, it locates the respective `stream_t` record through a hash table and updates all relevant fields (`stream_t` handling). If a packet belongs to a new stream, a new `stream_t` record is created and added into the hash table. Then, it extracts the actual data from each TCP segment by removing the protocol headers, and stores it in the appropriate memory page, depending on the stream in which it belongs (memory management). Whenever a new stream is created or terminated, or a sufficient amount of data has been gathered, the kernel module generates a respective event and enqueues it to an event queue (event creation).

B. Parallel Packet and Stream Processing

To scale performance, Scap uses all available cores in the system. To efficiently utilize multi-core architectures, modern network interfaces can distribute incoming packets to multiple hardware receive queues. To balance the network traffic load across the available queues and cores, Scap uses both RSS [23], which uses a hash function based on packets' 5-tuple, and dynamic load balancing, using flow director filters [25], to deal with short-term load imbalance. To map the two different streams of each bi-directional TCP connection to the same core, we modify the RSS seeds as proposed by Woo and Park [34].

Each core runs a separate instance of the NIC driver and Scap kernel module to handle interrupts and packets from the respective hardware queue. Thus, each Scap instance running on each core receives a different subset of network streams, as shown in Figure 1, effectively distributing the stream reassembly process across all available cores. To match the level of parallelism provided by the Scap kernel module, Scap's user-level stub creates as many worker threads as the available cores, hiding from the programmer the complexity

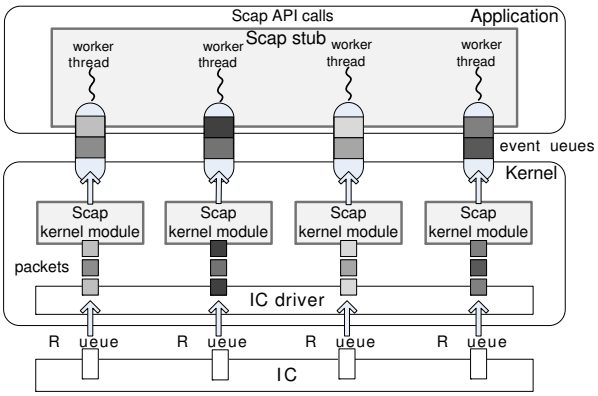


Figure 1. Overview of the Scap architecture.

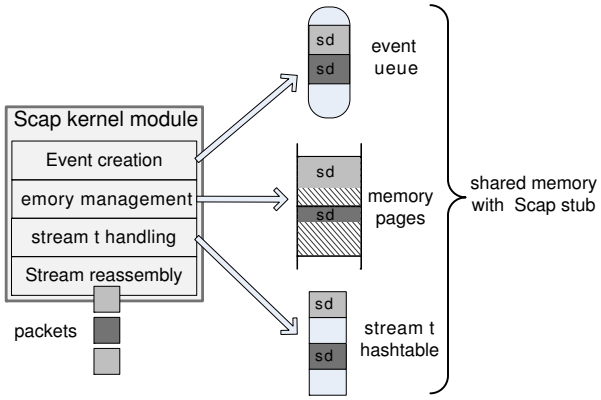


Figure 2. Overall operation of the Scap kernel module.

of creating and managing multiple processes or threads. Each worker thread processes the streams delivered to its core by its kernel-level counterpart. This collocation of user-level and kernel-level threads that work on the same data maximizes locality of reference and cache affinity, reducing this way context switches, cache misses [30], [31], and inter-thread synchronization. Each worker thread polls a separate event queue for events created by the kernel Scap thread running on the same core, and calls the respective callback function registered by the application to process each event.

V. Implementation

A. Scap Kernel Module

The Scap kernel module implements a new network protocol for receiving packets from network devices, and a new socket class, `PF_SCAP`, for communication between the Scap stub and the kernel module. Packets are transferred to memory through DMA and are scheduled for processing within the software interrupt handler—Scap's protocol handler in our case.

B. Fast TCP Reassembly

For each packet, the Scap kernel module finds and updates its respective `stream_t` record, or creates a new one. For fast lookup, we use a hash table by randomly choosing a hash function during initialization. Based on the transport-layer protocol headers, Scap extracts the packet's data and writes it directly to the current memory offset indicated in the `stream_t` record. Packets belonging to streams that exceed their cutoff value, as well as duplicate or overlapping TCP segments, are discarded immediately by the Scap kernel module without unnecessarily spending further CPU and memory resources on them. Streams can expire explicitly (e.g., via TCP FIN/RST), or implicitly, due to an inactivity timeout. For the latter, Scap maintains an *access list* with the active streams sorted by their last access time. Upon packet reception, the respective `stream_t` record is simply placed at the beginning of the access list, to keep it sorted. Periodically, starting from the end of the list, the kernel module compares the last access time of each stream with the current time, and expires all streams for which no packet was received within a specified period by creating stream termination events.

C. Memory Management

Reassembled streams are stored in a large memory buffer allocated by the kernel module and mapped in user level by the Scap stub. The size of this buffer is given as an argument (`buffer_len`) to `scap_create()`. The kernel module allocates the respective memory pages during initialization, and it is responsible for managing the usage of this memory among the several active streams. For each stream, a contiguous memory block is allocated (by our own memory allocator) according to the stream's chunk size. When this block fills up, the chunk is delivered for processing (by creating a respective event) and a new block is allocated for the next chunk. The Scap stub has access to this block through memory mapping, so an offset is enough for locating each stored chunk. To avoid dynamic allocation overhead, a large number of `stream_t` records are pre-allocated during initialization, and are memory-mapped by the Scap stub. More records are allocated dynamically as needed, in case of a large number of active streams. Thus, the number of streams that can be tracked concurrently is not limited by Scap.

D. Event Creation

A new event is triggered on stream creation, stream termination, and whenever stream data is available for processing. A data event can be triggered for one of the following reasons: (i) a memory chunk fills up, (ii) a flush timeout is passed, (iii) a cutoff value is exceeded, or (iv) a stream is terminated. When a stream's cutoff threshold is reached, Scap creates a final data processing event for its last chunk. However, its `stream_t` record remains in the hash table and in the access list, so that monitoring continues throughout its whole lifetime. This is required for gathering flow statistics and generating the appropriate termination event.

To avoid contention when the Scap kernel module runs in parallel across several cores, each core inserts events in a separate queue. When a new event is added into a queue, the `sk_data_ready()` function is called to wake up the corresponding worker thread, which calls `poll()` whenever its event queue is empty. Along with each event, the Scap stub receives and forwards to the user-level application a pointer to the respective `stream_t` record. To avoid race conditions

between the Scap kernel module and the application, Scap maintains a second instance of each `stream_t` record. The first copy is updated within the kernel, while the second is read by the user-level application. The kernel module updates the necessary fields of the second `stream_t` instance right before a new event for this stream is enqueued.

E. Hardware Filters

Packets taking part in the TCP three-way handshake are always captured. When the cutoff threshold is triggered for a stream, Scap adds dynamically the necessary FDIR filters to drop at the NIC layer all subsequent packets belonging to that stream. Note that although packets are dropped before they reach main memory, Scap needs to know when a stream ends. For this reason, we add filters to drop only packets that contain actual data segments, and still allow Scap to receive TCP RST or FIN packets that may terminate a stream.

This is achieved using the flexible 2-byte tuple option of FDIR filters. We have modified the NIC driver to allow for matching the offset, reserved, and TCP flags 2-byte tuple in the TCP header. Using this option, we add two filters for each stream: the first matches and drops TCP packets for which only the ACK flag is set, and the second matches and drops TCP packets for which only the ACK and PSH flags are set. The rest of the filter fields are based on each stream's 5-tuple. Thus, only TCP packets with RST or FIN flag will be forwarded to the Scap kernel module for stream termination.

Streams may also be terminated due to an inactivity timeout. For this reason Scap associates a timeout with each filter, and keeps a list of all filters sorted by their timeout values. An FDIR filter is removed (i) when a TCP RST or FIN packet arrives for a given stream, or (ii) when the timeout associated with a filter expires. Note that in the second case the stream may still be active, so if a packet of this stream arrives upon the removal of its filter, Scap will immediately re-install the filter. This is because the cutoff of this stream has been exceeded and the stream is still active. To handle long running streams, re-installed filters get a timeout twice as large as before. In this way, long-running flows will only be evicted a logarithmic number of times. If there is no space left on the NIC to accommodate a new filter, a filter with a small timeout is evicted, as it does not correspond to a long-lived stream.

Scap needs to provide accurate flow statistics upon the termination of streams that exceeded their cutoff, even if most of their packets were discarded at the NIC. Unfortunately, existing NICs provide only aggregate statistics for packets across all filters—not per filter. However, Scap is able to estimate accurate per-flow statistics, such as flow size and flow duration, based on the TCP sequence numbers of the RST/FIN packets. Also, by removing the NIC filters when their timeout expires, Scap receives packets from these streams periodically and updates their statistics.

Our implementation is based on the Intel 82599 NIC [25], which supports RSS and flow director filters. Similarly to this card, most modern 10GbE NICs such as Solarflare [35], SMC [36], Chelsio [37], and Myricom [38], also support RSS and filtering capabilities, so Scap can be effectively used with these NICs as well.

F. Handling Multiple Applications

Multiple applications can use Scap concurrently on the same machine. Given that monitoring applications require only read access to the stream data, there is room for stream sharing to avoid multiple copies. To this end, all Scap sockets share a single memory buffer for stream data and `stream_t` records. As applications have different requirements, Scap tries to combine and generalize all requirements at kernel level, and apply application-specific configurations at user level.

G. Packet Delivery

Applications may need to receive both reassembled streams and their individual packets, e.g., to detect TCP attacks [33]. Scap supports the delivery of the original packets if an application indicates that it needs them. In that case, Scap internally uses another memory-mapped buffer that contains a record for each packet of a stream. Each record contains a packet header with timestamp and capture length, and a pointer to the original packet payload in the stream.

H. API Stub

When `scap_start_capture()` is called, each worker thread runs an event-dispatch loop that polls its corresponding event queue, reads the next available event, and executes the registered callback function for the event (if any). The event queues contain `stream_t` objects, which have an `event` field and a pointer to the next `stream_t` object in the event queue. If this pointer is NULL, then there is no event in the queue, and the stub calls `poll()` to wait for future events.

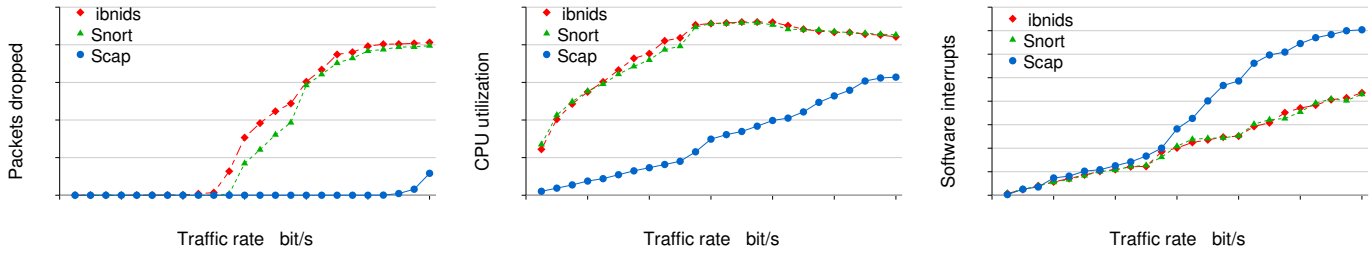
VI. Experimental Evaluation

A. Experimental Environment

a) The hardware: We use a testbed comprising two PCs interconnected through a 10 GbE switch. The first, equipped with two dual-core Intel Xeon 2.66 GHz CPUs with 4MB L2 cache, 4GB RAM, and an Intel 82599EB 10GbE NIC, is used for traffic generation. The second, used as a monitoring sensor, is equipped with two quad-core Intel Xeon 2.00 GHz CPUs with 6MB L2 cache, 4GB RAM, and an Intel 82599EB 10GbE NIC used for stream capture. Both PCs run 64-bit Ubuntu Linux (kernel version 2.6.32).

b) The trace: We replay a one-hour long full-payload trace captured at the access link that connects to the Internet a University campus with thousands of hosts. The trace contains 58,714,906 packets and 1,493,032 flows, totaling more than 46GB, 95.4% of which is TCP traffic. To achieve high replay rates (up to 6 Gbit/s) we split the trace in smaller parts of 1GB that fit into main memory, and replay each part 10 times while the next part is being loaded in memory.

c) The parameters: We compare the following systems: (i) Scap, (ii) Libnids v1.24 [14], and (iii) the Stream5 preprocessor of Snort v2.8.3.2 [1]. Libnids and Snort rely on Libpcap [13], which uses the `PF_PACKET` socket for packet capture on Linux. Similarly to Scap's kernel module, the `PF_PACKET` kernel module runs as a software interrupt handler that stores incoming packets to a memory-mapped buffer, shared with



(a) Packet loss (b) CPU utilization (c) Software interrupt load

Figure 3. Performance comparison of stream delivery for Snort, Libnids, and Scap, for varying traffic rates. We see than Scap can deliver streams to user space without any packet loss for two times higher traffic rates that Snort and Libnids, and with significantly lower CPU utilization. The Scap software interrupt load at high rates is higher than the other systems, because Scap process all packets with no loss at these rates, and it performs flow tracking and stream reassembly in the kernel, at the software interrupt handler.

Libpcap's user-level stub. In our experiments, the size of this buffer is set to 512MB, and the buffer size for reassembled streams is set to 1GB for Scap, Libnids, and Snort. We use a chunk size of 16KB, the `SCAP_TCP_FAST` reassembly mode, and an inactivity timeout of 10 seconds.

B. Results

We compare the performance of Scap, Snort, and Libnids when delivering reassembled streams to user level without any further processing. The Scap application receives all data from all streams with no cutoff, and runs as a single thread. Snort is configured with only the Stream5 preprocessor enabled.

Figure 3(a) shows the percentage of dropped packets as a function of the traffic rate. Scap delivers all steams to the user-level application without any packet loss for rates up to 5.5 Gbit/s. On the other hand, Libnids starts dropping packets at 2.5 Gbit/s (drop rate: 1.4%) and Snort at 2.75 Gbit/s (drop rate: 0.7%). Thus, Scap is able to deliver reassembled streams to monitoring applications for more than two times higher traffic rates. When the input traffic reaches 6 Gbit/s, Libnids drops 81.2% and Snort 79.5% of the total packets received.

The reason for this performance difference lies in the extra memory copy operations needed for stream reassembly at user level. When a packet arrives for Libnids and Snort, the kernel writes it in the next available location in a common ring buffer. When performing stream reassembly, Libnids and Snort may need to *copy* each packet's payload from the ring buffer to a memory buffer allocated specifically for this packet's stream. Scap avoids this extra copy operation because the kernel module copies the packet's data *not* to a common buffer, but directly to a memory buffer allocated specifically for this packet's stream.

Figure 3(b) shows that the CPU utilization of the Scap user-level application is considerably lower than the utilization of Libnids and Snort, which at 3 Gbit/s exceeds 90%, saturating the processor. In contrast, the CPU utilization for the Scap application is less then 60% even for speeds up to 6 Gbit/s, as the user application does very little work: stream reassembly is performed by the kernel module, which increases the software interrupt load, as we see in Figure 3(c).

When the traffic rate is less than 2.5 Gbit/s, the interrupt load of the Scap kernel module is almost the same as for the other two systems. However, as soon as the traffic rate exceeds 3 Gbit/s, a rate for which both Libnids and Snort lose packets, the software interrupt load of Scap becomes higher. This is because Scap manages to process all packets at these rates, while the other two systems drop most of them. Indeed, dropped packets are not copied in the memory-mapped buffer and are not being processed any further by the software interrupt handler, so the software interrupt load for systems that drop packets becomes lower. A more detailed evaluation of Scap can be found in our previous work [24].

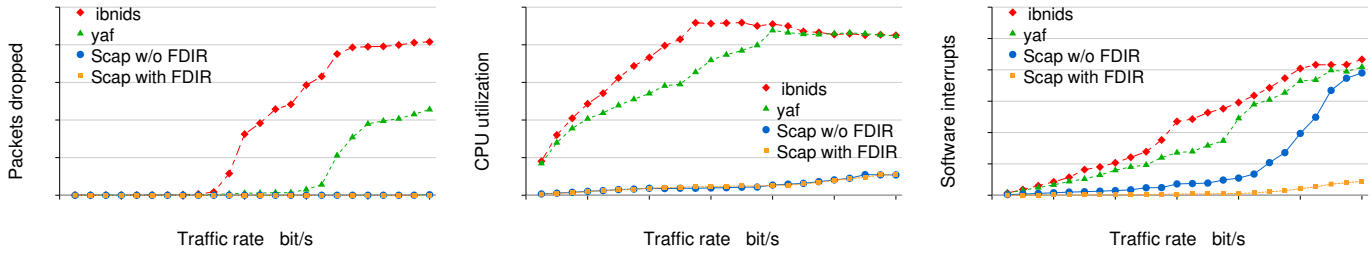
VII. Scap Applications

A. Flow Statistics Export

Our first Scap application collects and exports flow-based statistics. Scap already gathers these statistics in its kernel module, and stores them in the `stream_t` structure of each stream. Thus, there is no need to receive any actual stream data at user level at all. To achieve this, the stream cutoff is set to zero, to efficiently discard all data in the kernel or NIC. All the required statistics for each stream are retrieved upon stream termination by registering a callback function. The following listing shows the code of this Scap application.

```
1  scap_t *sc = scap_create("eth0", SCAP_DEFAULT,
2      SCAP_TCP_FAST, 0);
3  scap_set_cutoff(sc, 0);
4  scap_dispatch_termination(sc, stream_close);
5  scap_start_capture(sc);
6
7  void stream_close(stream_t *sd) {
8    export(sd->hdr.src_ip, sd->hdr.dst_ip,
9      sd->hdr.src_port, sd->hdr.dst_port,
10     sd->stats.bytes, sd->stats.pkts,
11     sd->stats.start, sd->stats.end);
12 }
```

In line 1 we create a new Scap socket for capturing streams from the `eth0` interface. Then, we set the stream cutoff to zero for discarding all stream data (line 3), set the `stream_close()` as a callback function to be called upon stream termination (line 4), and finally start the capturing process (line 5). The `stream_ close()` function exports stream statistics through the `sd` descriptor that is passed as its argument (lines 7–12).



(a) Packet loss (b) CPU utilization (c) Software interrupt load

Figure 4. Performance evaluation of the Scap flow statistics export application in comparison to YAF and Libnids while varying the traffic rate.

We now evaluate the performance of this Scap flow export application by comparing it with YAF v2.1.1 [39], a Libpcap-based flow export tool, and with a Libnids-based program that receives reassembled flows and exports the same flow statistics. Although Scap can use all eight available cores, for a fair comparison with the other two tools which are single-threaded, we configure it to use a single worker thread. However, for all tools, hardware and software interrupt handling for packet processing in kernel takes advantage of all available cores, utilizing the NIC's multiple queues and RSS.

Figures 4(a), 4(b), and 4(c) present the percentage of dropped packets, the average CPU utilization of the monitoring application on a single core, and the software interrupt load while varying the traffic rate from 250 Mbit/s to 6 Gbit/s. We see that Libnids starts losing packets when the traffic rate exceeds 2 Gbit/s. The reason can be seen in Figures 4(b) and 4(c), where the total CPU utilization of Libnids exceeds 90% at 2.5 Gbit/s. YAF performs slightly better than Libnids, but when the traffic reaches 4 Gbit/s, it also drives CPU utilization to 100% and starts losing packets as well. This is because both YAF and Libnids receive all packets captured by Libpcap in user space and then drop them, as the packets themselves are not needed.

Scap processes all packets even for a 6 Gbit/s load. As shown in Figure 4(b), the CPU utilization of the Scap application is always less than 10%, as it practically does not do any work at all. All the work has already been done by Scap's kernel module. One would expect the overhead of this module (shown in Figure 4(c)) to be relatively high. Surprisingly, however, the software interrupt load of Scap is even lower compared to YAF and Libnids. This is because Scap does not copy the incoming packets around: as soon as a packet arrives, the kernel module accesses only the needed information from its headers, updates the respective *stream_t*, and just drops it. In contrast, Libnids and YAF receive all packets to user space, resulting in much higher overhead. YAF performs better than Libnids because it receives only the first 96 bytes of each packet and does not perform stream reassembly.

When Scap uses FDIR filters to discard the majority of the packets at the NIC level it achieves even better performance. Figure 4(c) shows that the software interrupt load is significantly lower with FDIR filters: as little as 2% for 6 Gbit/s. Indeed, Scap with FDIR brings into main memory as little as 3% of the total packets—just the packets involved in TCP session creation and termination. The rest of the packets are just not needed, and they are never brought to main memory.

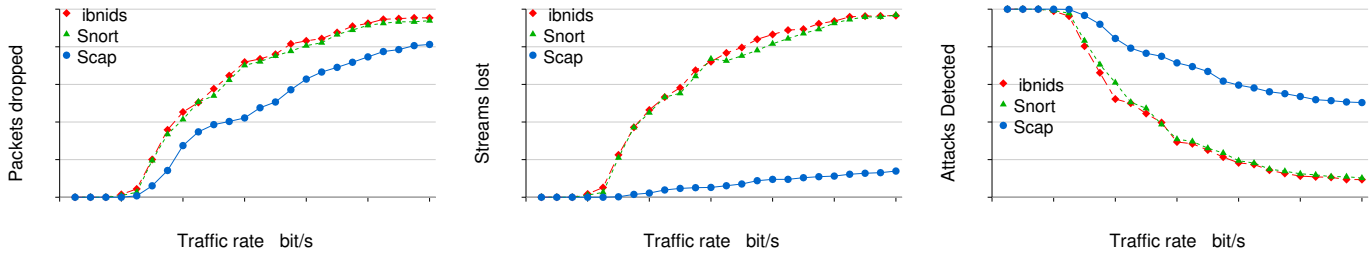
B. NIDS Signature Matching

The next Scap application we implemented is a simple signature-based network-level intrusion detection system. This application matches simple NIDS signatures in the captured reassembled transport-layer streams. Signatures contain patterns that are matched against the captured streams using the Aho-Corasick pattern matching algorithm [40]. To match patterns spanning multiple packets of the same connection, and avoid evasion attempts based on TCP segmentation discrepancies, stream reassembly and protocol normalization are needed, respectively. Fortunately, Scap provides both of them. The following listing shows the few lines of the code we used to implement this application with Scap.

```
1  load_signatures(dfa, signatures);
2  scap_t *sc = scap_create("eth0", 512M,
3      SCAP_TCP_FAST, 0);
4  scap_set_worker_threads(sc, numCPU);
5  scap_dispatch_data(sc, stream_process);
6  scap_start_capture(sc);
7
8  void stream_process(stream_t *sd) {
9   search(dfa, sd->data, sd->data_len, MatchFound);
10 }
```

We begin by loading the signatures and compiling them into a DFA for the Aho-Corasick pattern matching algorithm. Then, we create an Scap socket without setting a cutoff, so that all traffic is captured and processed (lines 2–3). Then, we configure Scap to use $numCPU$ worker threads, each one pinned to a single CPU core (where $numCPU$ is the number of available CPU cores), to speed up pattern matching with parallel stream processing. Finally, we register `stream_process()` as the callback function for processing stream chunks (line 5) and start the capturing process (line 6). As we do not define a chunk size, the default value is used (16KB). The `stream_process()` function calls `search()`, which looks for the set of known signatures we added into the DFA within the `sd->data_len` bytes of the stream chunk, starting from the `sd->data` pointer. In case of a match, it calls the `MatchFound()` function.

We evaluate the performance of this Scap NIDS application using only one worker thread, to compare its performance with Snort and Libnids, which are single-threaded. We do not apply any cutoff so that all traffic is delivered to the application. We loaded 2,120 signatures from the "web attack" rules of the official VRT Snort rule set [41]. These signatures



(a) Packet loss (b) Lost streams (c) Detected Attacks

Figure 5. Performance evaluation of the Scap NIDS signature matching application in comparison to Snort and Libnids while varying the traffic rate.

resulted in 223,514 matches in our trace. We compare this Scap application with Snort and Libnids using the same string matching algorithm and the same set of signatures. To ensure a fair comparison, Snort is configured only with the Stream5 preprocessor enabled, which performs transport-layer stream reassembly, using the same set of rules, so that all tools end up using the same automaton.

Figure 5(a) shows the percentage of dropped packets for each application as a function of the traffic rate. We see that Snort and Libnids process traffic rates of up to 750 Mbit/s without dropping any packets, while the Scap application processes up to 1 Gbit/s traffic with no packet loss with one worker thread. The main reasons for the improved performance of Scap are the reduced memory copies during stream reassembly and the improved cache locality when grouping multiple packets into their respective transport-layer streams. Indeed, by reassembling packets into streams from the moment they arrive, packets are not copied around: consecutive segments arrive together, are stored together, and are consumed together. In contrast, Libnids and Snort perform stream reassembly too late: the segments have been stored in (practically) random locations all over main memory.

Moreover, for high traffic rates, the Scap application drops significantly fewer packets than Snort and Libnids, e.g., at 6 Gbit/s it processes three times more traffic. This behavior has a positive effect on the number of detected attacks. As shown in Figure 5(c), under the high load of 6 Gbit/s, Snort and Libnids detect less than 10% of the attacks, as more than 90% of the packets are dropped, while the Scap application detects five times as many: 50.34% of the attacks, when 80% of the packets were dropped. Although the percentage of missed attacks for Snort and Libnids is proportional to the percentage of dropped packets, the accuracy of the Scap application is affected less by high packet loss rates. This is because Scap under overload tends to retain more packets towards the beginning of each stream. As we use web attack signatures, matches are usually found within the first few bytes of HTTP requests or responses. Also, Scap tries to deliver contiguous chunks, which improves detection accuracy compared to the delivery of chunks with random holes.

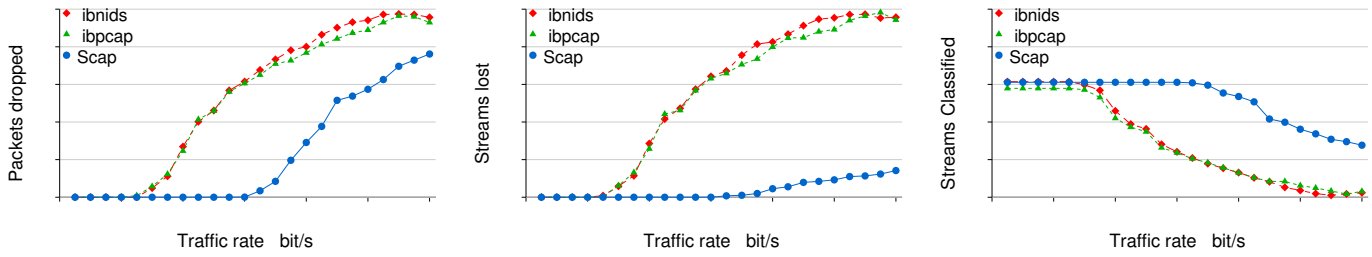
Figure 5(b) shows that the percentage of lost streams for Snort and Libnids is proportional to the packet loss rate. In contrast, the Scap application loses significantly less streams than the corresponding packet loss ratio. Even for 81.2% packet loss at 6 Gbit/s, only 14% of the total streams are completely lost. This is because Libnids and Snort drop packets randomly under overload, while Scap is able to (i) assign more memory to new or small streams, (ii) cut the long tails of large streams, and (iii) deliver more streams intact when the available memory is limited. Moreover, the Scap kernel module always receives and processes all important protocol packets, e.g., during the TCP handshake. These packets may result in the creation of new streams, but they do not carry data to be stored. In contrast, when a packet capture library drops these packets, user-level stream reassembly libraries cannot reassemble the respective streams, and completely misses them.

C. Layer-7 Traffic Classification

Next, we use Scap to implement a traffic classification application, which aims to identify application-layer protocols within stream payloads. Protocol normalization and stream reassembly are essential for accurate protocol identification, as they ensure that application-specific patterns can be matched in stream payloads that might be spanning different TCP segments. Also, each transport-layer stream should be assigned one application protocol. When one stream is classified to one protocol, the rest of the packets belonging to that stream do not have to be inspected for protocol detection. They are just accounted for, to compute an accurate distribution of the monitored traffic to the identified application-layer protocols. Scap is an attractive framework to build such L7 traffic classification applications: it supports all of the above features with reduced application code complexity and improved performance.

We based our implementation on the L7-filter library [5], which provides a set of patterns that can be used to detect application-layer protocols. The packets of each stream are matched against the application-specific patterns until a match is found. Then, the stream is marked with this application protocol. The rest of the packets of already classified streams are not further inspected. Instead, they are just accounted for, by updating per-stream and per-application statistics. Flows for which an application pattern cannot be identified are marked as "unknown."

Scap provides several useful features for implementing this application and improving its performance. When a match is found and a stream is classified, the rest of the stream data is just discarded (at kernel or NIC level) using `scap_discard_stream()`. The Scap kernel module is responsible for gathering statistics for identified flows, without



(a) Packet loss (b) Lost streams (c) Classified Streams

Figure 6. Performance evaluation of an Scap L7 traffic classification application using L7-filter patterns in comparison to L7-filter on top of Libpcap and Libnids while varying the traffic rate.

copying any stream data to user level. Furthermore, as flows are typically classified after inspecting the first few KBs of a stream, we can significantly improve performance using a stream cutoff. In our Scap application, we set a cutoff of 10 MB per stream, and compared its accuracy with L7-filter implemented over other libraries. We observed that accuracy is not reduced, as no more protocols are detected after 10 MB per stream, but in fact is increased due to stream reassembly and protocol normalization, and mainly because Scap avoids (or reduces) packet loss in high rates.

The following listing is the main part of the Scap code for implementing traffic classification using L7-filter.

```
1  load_l7_patterns(dfa, l7_patterns);
2  scap_t *sc = scap_create("eth0", 512M,
3      SCAP_TCP_FAST, 0);
4  scap_set_cutoff(sc, 10485760);
5  scap_set_worker_threads(sc, numCPU);
6  scap_dispatch_data(sc, stream_process);
7  scap_dispatch_termination(sc, stream_close);
8  scap_start_capture(sc);
9
10 void stream_process(stream_t *sd) {
11   int protocol;
12   if (sd->user_state!=NULL) {
13     protocol=search(dfa, sd->data, sd->data_len);
14     if (protocol!=NO_MATCH) {
15       (int)sd->user_state=protocol;
16       scap_discard_stream(sc, sd);
17     }
18   }
19 }
20
21 void stream_close(stream_t *sd) {
22   export(sd->hdr.src_ip, sd->hdr.dst_ip,
23     sd->hdr.src_port, sd->hdr.dst_port,
24     sd->stats.bytes, sd->stats.pkts,
25     sd->stats.start, sd->stats.end,
26     sd->user_state);
27 }
```

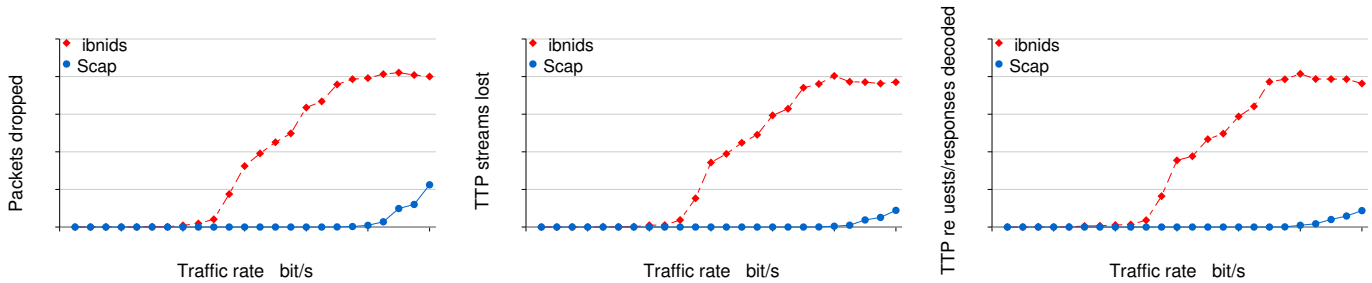
We begin by loading the L7-filter patterns into a DFA (line 1). Then, we create an Scap socket as usual (lines 2–3) and set a cutoff of 10 MB per stream (line 4). We configure Scap to use as many worker threads as the available CPU cores to enable parallel stream processing using the `scap_set_worker_threads()` function (line 5). To process incoming stream data chunks (of 16 KB by default) we set the `stream_process()` callback (line 6), and then set the `stream_close()` callback to export stream statistics along with the application protocol on stream termination events (line 7). Then, stream capture begins (line 8).

The `stream_process()` function (lines 10–19) attempts to match a stream data chunk against the set of L7-filter patterns (line 13) only for streams that have not been already classified (line 12). In case of a match (lines 14–17), the application protocol that is detected for this stream is stored in the `sd->user_state` field, which is provided by Scap for keeping application-specific state (line 15). This allows the application to know whether a stream has been already classified, and the protocol that was identified. Also, in case of a match, the `scap_discard_stream()` function is called to efficiently discard the data of this stream in kernel or NIC, as it is not needed for inspection—just for accounting, which will be handled by the Scap kernel module. Finally, the `stream_close()` function (lines 21–27) exports the statistics of each terminated stream along with the `sd->user_state`, which is set to the identified application protocol for this stream, or NULL for streams unidentified streams.

We evaluate the performance of this Scap application in comparison to two similar applications built on top of Libpcap and Libnids, using the same set of L7-filter patterns. The Libpcap-based application matches these patterns in the captured packets without performing stream reassembly, as it only supports flow tracking. We set one worker thread for Scap in this experiment to make a fair comparison.

Figures 6(a)–6(c) show the percentage of dropped packets, lost streams, and classified streams, respectively. We see that L7-filter on top of Libpcap and Libnids starts loosing packets at 1.5 Gbit/sec, while Scap starts dropping packets at 3.25 Gbit/sec, which is a 2.2 times improvement. Although Libnids and Libpcap lose a similar percentage of streams to their respective packet loss rates, Scap retains much more streams: 86% of the streams are still captured by Scap at 6 Gbit/sec, despite 76% of dropped packets. At the same rate, Libpcap tracks only 5.7% of the streams (due to a 93% packet loss) and Libnids captures 4.3% of the total streams (with a 95.7% packet loss). This is due to the overload control mechanisms that Scap internally implements.

Figure 6(c) shows the percentage of streams that were classified by L7-filter into one of the supported applications as a function of the traffic rate. We see that at low rates (with no packet loss), L7-filter on top of Libnids and Scap classifies 61.5% of the total streams in our trace. The rest of the streams



(a) Packet loss (b) Lost streams (c) HTTP requests/responses decoded

Figure 7. Performance evaluation of an Scap HTTP parsing application in comparison to a Libnids HTTP parsing application while varying the traffic rate.

were reported as "unknown." L7-filter on top of Libpcap at low rates classifies 58% of the streams, which is 3.5% less classified streams than L7-filter over Libnids and Scap. This is because the Libpcap application does not perform stream reassembly, so it misses patterns that span multiple segments or require protocol normalization.

At higher rates, the percentage of classified streams rapidly drops for the Libnids and Libpcap implementations. The unclassified streams are either streams that were not classified by L7-filter, or streams that were lost due to overload conditions. As the traffic rate increases, the percentage of lost streams is significantly higher, as we see in Figure 6(b). In contrast, the Scap application retains and classifies much more streams than the other implementations. For example, 27.7% of the streams are classified at 6 Gbit/sec, with 76% of the packets getting dropped. At the same rate, the Libpcap and Libnids applications classified only 3% and 2.5% of the streams, respectively. This is due to the intelligent overload control provided by Scap: the first few bytes of each stream are stored in memory with higher priority, so initial stream parts are delivered even for very high rates. The 10 MB stream cutoff has also a positive impact on the percentage of classified streams at high traffic rates.

D. HTTP Parsing

The last application we implemented with Scap is an HTTP parsing tool that analyzes all HTTP headers in HTTP requests and responses. To be accurate, it requires to operate on the reassembled HTTP payload. The following code listing is the main part of this Scap application:

```
1  scap_t *sc = scap_create("eth0", 512M,
2      SCAP_TCP_FAST, 0);
3  scap_set_worker_threads(sc, numCPU);
4  scap_set_filter(sc, "port 80");
5  scap_dispatch_data(sc, stream_process);
6  scap_start_capture(sc);
7
8  void stream_process(stream_t *sd) {
9    int res;
10   res=HTTP_parse(sd->data, sd->data_len,
11       (HTTP_header*)sd->user_state);
12   if (res==HTTP_HEADER_INCOMPLETE)
13     scap_keep_stream_chunk(sc, sd);
14 }
```

We start by creating an Scap socket (lines 1–2) and starting multiple worker threads (line 3). We do not set any stream cutoff, as many HTTP requests and responses can be sent through the same stream, due to persistent HTTP connections. Then, we set a BPF filter to process only streams with source or destination port 80 (line 4), set the `stream_process()` callback for stream data processing (line 5), and start the stream capture (line 6). The `stream_process()` function calls `HTTP_parse()` to process the `sd->data_len` captured bytes of an HTTP stream, starting from the `sd->data` pointer. The decoded HTTP header values are stored in `sd->user_state` (lines 10–11). If the HTTP header remains incomplete in this stream chunk, `stream_process()` asks from the Scap kernel module to keep this chunk into memory and merge it with the subsequent chunk, until the whole HTTP header is received, using the `keep_stream_chunk()` function (line 13). This way, the whole HTTP header will end up in contiguous memory and can then be easily processed. We cannot discard the rest of the stream as it may contain more HTTP requests and responses that need to be decoded.

We evaluate the performance of this Scap application when using one worker thread, comparing with a respective program written on top of Libnids (also single-threaded). In Figure 7(a) we see that Libnids starts dropping packets at 2 Gbit/sec (0.9%), while Scap drops 0.9% of the packets at 5 Gbit/sec. At 6 Gbit/sec, the Libnids application loses 80% of the packets, 77% of the HTTP streams, and 76% of the HTTP requests and responses. In contrast, our Scap application performs much better at 6 Gbit/sec: it drops 22.5% of the packets, 8.9% of the HTTP streams, and 8.7% of the HTTP requests and responses.

VIII. Related Work

A. Improving Packet Capture

Several techniques have been proposed to reduce the kernel overhead and the number of memory copies for delivering packets to the application [11], [42]–[44]. Scap can also use such techniques to improve its performance. The main difference, however, is that all these approaches operate at the network layer. Thus, monitoring applications that require transport-layer streams should implement stream reassembly, or use a separate user-level library, resulting in reduced performance and increased application complexity. In contrast, Scap operates at the transport layer and directly assembles incoming

packets to streams in the kernel, offering the opportunity for a wide variety of performance optimizations and many features.

Papadogiannakis et al. [32] show that memory access locality in passive network monitoring applications can be improved when reordering the packet stream based on source and destination port numbers. Scap also improves memory access locality and cache usage in a similar manner when grouping packets into streams.

B. Taking Advantage of Multi-core Systems

Fusco and Deri [45] utilize the receive-side scaling feature of modern NICs in conjunction with multi-core systems to improve packet capture performance, by mapping each hardware receive queue to one core. Sommer et al. [46] take advantage of multi-core processors to parallelize event-based intrusion prevention systems using multiple event queues that collect semantically related events for in-order execution. Storing related events in a single queue localizes memory access to shared state by the same thread. Pesterev et al. [30] improve TCP connection locality using the flow director filters to optimally balance the TCP packets among the available cores. We view these works as orthogonal to Scap: such advances in multi-core systems can be easily used by Scap.

C. Packet Filtering

Dynamic packet filtering reduces the cost of adding and removing filters at runtime [47]–[49]. Deri et al. [26] propose to use the flow director filters for common filtering needs. Other approaches allow applications to move simple tasks to the kernel packet filter to improve performance [42], [50], [51]. Scap suggests a relatively different approach: applications empowered with a *Stream* abstraction can communicate their stream-oriented filtering and processing needs to the underlying kernel module at runtime through the Scap API, to achieve lower complexity and better performance. For instance, Scap is able to filter packets within the kernel or at the NIC layer based on a flow size cutoff limit, allowing to set dynamically different cutoff values per-stream, while the existing packet filtering systems are not able to support a similar functionality.

D. TCP Stream Reassembly

Libnids [14] is a user-level library on top of Libpcap for TCP stream reassembly based on the emulation of a Linux network stack. Similarly, the Stream5 [52] preprocessor, part of Snort NIDS [1], performs TCP stream reassembly at user level, emulating the network stacks of various operating systems. Scap shares similar goals with Libnids and Stream5. However, previous works treat TCP stream reassembly as a necessity [15], mostly for the avoidance of evasion attacks against intrusion detection systems [7], [8], [10], [29]. On the contrary, Scap views transport-layer streams as the fundamental abstraction that is exported to network monitoring applications, and as the right vehicle to implement aggressive optimizations.

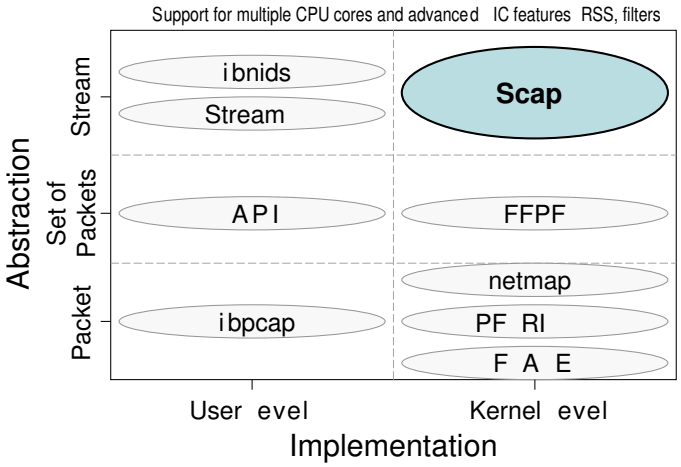


Figure 8. Categorization of network monitoring tools and systems that support commodity NICs.

E. Per-flow Cutoff

The Time Machine network traffic recording system [17] exploits the heavy-tailed nature of Internet traffic to reduce the number of packets stored on disk for retrospective analysis, by applying a per-flow cutoff. Limiting the size of flows can also improve the performance of intrusion detection systems under load [18], [19], by focusing detection on the beginning of each connection. Lin et al. [20] present a system for storing and replaying network traffic, using an (N, M, P) scheme to reduce the traffic stored: they suggest to capture N bytes per flow and then M bytes per packet for the next P packets of the flow. Canini et al. [16] propose a similar scheme for traffic classification, by sampling more packets from the beginning of each flow. Scap shares a similar approach with these works, but implements it within a general framework for fast and efficient network traffic monitoring, using the *Stream* abstraction to enable the implementation of performance improvements at the most appropriate level. For instance, Scap implements the per-flow cutoff inside the kernel or at the NIC layer, while previous approaches have to implement it in user space. As a result, they first receive *all* packets from kernel in user space, and then discard those that are not needed.

F. Overload Control

Load shedding is proposed as a defense against overload attacks in Bro [2], whereby the NIDS operator is responsible for defining a discarding strategy. Barlet-Ros et al. [53] also propose a load shedding technique using an on-line prediction model for query resource requirements, so that the monitoring system sheds load under conditions of excessive traffic using uniform packet and flow sampling. Dreger et al. [27] deal with packet drops due to overloads in a NIDS using load levels, which are precompiled sets of filters that correspond to different subsets of traffic enabled by the NIDS depending on the workload.

G. Summary

To place our work in context, Figure 8 categorizes Scap and related works along two dimensions: the main abstraction

provided to applications, i.e., packet, set of packets, or stream, and the level at which this abstraction is implemented, i.e., user or kernel level. Traditional systems such as Libpcap [13] use the *packet* as basic abstraction and are implemented in user level. More sophisticated systems such as netmap [44], FLAME [51], and PF_RING [11] also use the packet as basic abstraction, but are implemented in kernel and deliver better performance. MAPI [54] and FFPF [42] use higher level abstractions such as the *set of packets*. Libnids and Stream5 provide the transport-layer *Stream* as their basic abstraction, but operate at user level and thus achieve poor performance and miss several opportunities of efficiently implementing this abstraction. We see Scap as the only system that provides a high-level abstraction, and at the same time implements it at the appropriate level, enabling a wide range of performance optimizations and features.

IX. Conclusion

In this paper, we have identified a gap in network traffic monitoring: applications usually need to express their monitoring requirements at a high level, using context from the transport layer or even higher, while most monitoring tools still operate at the network layer. To bridge this gap, we have presented the design, implementation, and evaluation of Scap, a stream-oriented network monitoring framework that offers an expressive API and significant performance improvements for applications that process traffic at the transport layer and beyond. Scap gives the stream abstraction a first-class status, and provides an OS subsystem for capturing transport-layer streams while minimizing data copy operations by optimally placing network data into stream-specific memory regions.

The results of our experimental evaluation demonstrate that Scap can deliver all streams will no packet loss for rates up to 5.5 Gbit/s using a single core, achieving two times higher performance than other systems. Moreover, we showed how four popular network monitoring applications can be easily implemented with Scap by leveraging its *stream* abstraction, and how their performance is significantly improved as a result of Scap's stream processing features and optimizations. As networks get increasingly faster and network monitoring applications more sophisticated, we believe that approaches like Scap, based on aggressive optimizations at the kernel level or even on the NIC, will become important for improving the overall performance of network monitoring applications.

Acknowledgements

This research was performed with the financial support of the Prevention of and Fight against Crime Programme of the European Commission – Directorate-General Home Affairs (project GCC). This work was also supported in part by the FP7 project SysSec and the FP7-PEOPLE-2009-IOF project MALCODE, funded by the European Commission under Grant Agreements No. 254116 and No. 257007. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein. Antonis Papadogiannakis and Evangelos Markatos are also with the University of Crete.

References

[1] M. Roesch, "Snort: Lightweight Intrusion Detection for Networks," in *USENIX Large Installation System Administration Conference (LISA)*, 1999.

[2] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Computer Networks*, vol. 31, no. 23-24, 1999.

[3] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee, "BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation," in *USENIX Security Symposium*, 2007.

[4] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated worm fingerprinting," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[5] "Application Layer Packet Classifier for Linux (L7-filter)," http://l7-filter.sourceforge.net/.

[6] T. Karagiannis, A. Broido, M. Faloutsos, and K. claffy, "Transport Layer Identification of P2P Traffic," in *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2004.

[7] S. Dharmapurikar and V. Paxson, "Robust TCP Stream Reassembly in the Presence of Adversaries," in *USENIX Security Symposium*, 2005.

[8] M. Handley, V. Paxson, and C. Kreibich, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics," in *USENIX Security Symposium*, 2001.

[9] T. H. Ptacek and T. N. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," Secure Networks, Inc., Tech. Rep., 1998.

[10] T.-H. Cheng, Y.-D. Lin, Y.-C. Lai, and P.-C. Lin, "Evasion Techniques: Sneaking through Your Intrusion Detection/Prevention Systems," *IEEE Communications Surveys Tutorials*, vol. 14, no. 4, 2012.

[11] L. Deri, N. S. P. A, V. D. B. Km, and L. L. Figuretta, "Improving Passive Packet Capture: Beyond Device Polling," in *International System Administration and Network Engineering Conference (SANE)*, 2004.

[12] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Winter USENIX Conference*, 1993.

[13] S. McCanne, C. Leres, and V. Jacobson, "Libpcap," http://www.tcpdump.org/, Lawrence Berkeley Laboratory.

[14] "Libnids," http://libnids.sourceforge.net/.

[15] G. Varghese, J. A. Fingerhut, and F. Bonomi, "Detecting Evasion Attacks at High Speeds without Reassembly," in *ACM SIGCOMM Conference on Data Communication*, 2006.

[16] M. Canini, D. Fay, D. J. Miller, A. W. Moore, and R. Bolla, "Per Flow Packet Sampling for High-Speed Network Monitoring," in *International Conference on Communication Systems and Networks (COMSNETS)*, 2009.

[17] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider, "Enriching Network Security Analysis with Time Travel," in *ACM SIGCOMM Conference on Data Communication*, 2008.

[18] T. Limmer and F. Dressler, "Improving the Performance of Intrusion Detection using Dialog-based Payload Aggregation," in *IEEE Global Internet Symposium (GI)*, 2011.

[19] A. Papadogiannakis, M. Polychronakis, and E. P. Markatos, "Improving the Accuracy of Network Intrusion Detection Systems Under Load Using Selective Packet Discarding," in *ACM European Workshop on System Security (EUROSEC)*, 2010.

[20] Y.-D. Lin, P.-C. Lin, T.-H. Cheng, I.-W. Chen, and Y.-C. Lai, "Low-Storage Capture and Loss Recovery Selective Replay of Real Flows," *IEEE Communications Magazine*, vol. 50, no. 4, 2012.

[21] A. Papadogiannakis, M. Polychronakis, and E. P. Markatos, "Tolerating Overload Attacks Against Packet Capturing Systems," in *USENIX Annual Technical Conference (ATC)*, 2012.

[22] R. Smith, C. Estan, and S. Jha, "Backtracking Algorithmic Complexity Attacks Against a NIDS," in *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[23] Intel Server Adapters, "Receive Side Scaling on Intel Network Adapters," http://www.intel.com/support/network/adapter/pro100/sb/cs-027574.htm.

[24] A. Papadogiannakis, M. Polychronakis, and E. P. Markatos, "Scap: Stream-Oriented Network Traffic Capture and Analysis for High-Speed Networks," in *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2013.

[25] Intel, "82599 10 GbE Controller Datasheet," http://download.intel.com/design/network/datashts/82599_datasheet.pdf.

[26] L. Deri, J. Gasparakis, P. Waskiewicz, and F. Fusco, "Wire-Speed Hardware-Assisted Traffic Filtering with Mainstream Network Adapters," *Advances in Network-Embedded Management and Applications*, 2011.

[27] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer, "Operational Experiences with High-Volume Network Intrusion Detection," in *ACM Conference on Computer and Communications Security (CCS)*, 2004.
[28] W. Lee, J. B. D. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and Y. Zhang, "Performance Adaptation in Real-Time Intrusion Detection Systems," in *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.
[29] M. Vutukuru, H. Balakrishnan, and V. Paxson, "Efficient and Robust TCP Stream Normalization," in *IEEE Symposium on Security and Privacy*, 2008.
[30] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris, "Improving Network Connection Locality on Multicore Systems," in *ACM European Conference on Computer Systems (EuroSys)*, 2012.
[31] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: Exploiting Parallelism to Scale Software Routers," in *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009.
[32] A. Papadogiannakis, G. Vasiliadis, D. Antoniades, M. Polychronakis, and E. P. Markatos, "Improving the Performance of Passive Network Monitoring Applications with Memory Locality Enhancements," *Computer Communications*, vol. 35, no. 1, 2012.
[33] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "TCP Congestion Control with a Misbehaving Receiver," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 5, 1999.
[34] S. Woo and K. Park, "Scalable TCP Session Monitoring with Symmetric Receive-side Scaling," Technical report, KAIST, Tech. Rep., 2012.
[35] Solarfrare, "10GbE LOM/Controller Family," http://www.solarflare.com/Content/UserFiles/Documents/Solarflare_SFC9000_10GbE_Controller_Brief.pdf.
[36] SMC Networks, "SMC10GPCIe-XFP Tiger Card 10G PCIe 10GbE XFP Server Adapter," http://www.smc.com/files/AF/ds_SMC10GPCIe_XFP.pdf.
[37] Chelsio Communications, "T4 unified wire adapters," http://www.chelsio.com/adapters/t4_unified_wire_adapters.
[38] Myricom, "10G-PCIE-8B-S Single-Port 10-Gigabit Ethernet Network Adapters," https://www.myricom.com/images/stories/10G-PCIE-8B-S.pdf.
[39] C. M. Inacio and B. Trammell, "YAF: Yet Another Flowmeter," in *USENIX Large Installation System Administration Conference (LISA)*, 2010.
[40] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Communications of the ACM*, vol. 18, 1975.
[41] "Sourcefire vulnerability research team (vrt)," http://www.snort.org/vrt/.
[42] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis, "FFPF: Fairly Fast Packet Filters," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2004.
[43] Myricom, "Myricom Sniffer10G," http://www.myricom.com/scs/SNF/doc/, 2010.
[44] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," in *USENIX Annual Technical Conference (ATC)*, 2012.
[45] F. Fusco and L. Deri, "High Speed Network Traffic Analysis with Commodity Multi-core Systems," in *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2010.
[46] R. Sommer, V. Paxson, and N. Weaver, "An Architecture for Exploiting Multi-Core Processors to Parallelize Network Intrusion Prevention," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 10, 2009.
[47] L. Deri, "High-Speed Dynamic Packet Filtering," *Journal of Network and Systems Management*, vol. 15, no. 3, 2007.
[48] J. Van Der Merwe, R. Caceres, Y. Chu, and C. Sreenan, "mmdump: A Tool for Monitoring Internet Multimedia Traffic," *ACM SIGCOMM CCR*, vol. 30, no. 5, 2000.
[49] Z. Wu, M. Xie, and H. Wang, "Swift: A Fast Dynamic Packet Filter," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.
[50] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. D. Keromytis, "xPF: Packet Filtering for Low-Cost Network Monitoring," in *IEEE Workshop on High-Performance Switching and Routing (HPSR)*, 2002.
[51] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, M. B. Greenwald, and J. M. Smith, "Efficient Packet Monitoring for Network Management," in *IFIP/IEEE Network Operations and Management Symposium (NOMS)*, 2002.
[52] J. Novak and S. Sturges, "Target-Based TCP Stream Reassembly," https://www.snort.org/documents/36, 2007.
[53] P. Barlet-Ros, G. Iannaccone, J. Sanjuàs-Cuxart, D. Amores-López, and J. Solé-Pareta, "Load Shedding in Network Monitoring Applications," in *USENIX Annual Technical Conference (ATC)*, 2007.
[54] P. Trimintzios, M. Polychronakis, A. Papadogiannakis, M. Foukarakis, E. P. Markatos, and A. Øslebø, "DiMAPI: An Application Programming Interface for Distributed Network Monitoring," in *IFIP/IEEE Network Operations and Management Symposium (NOMS)*, 2006.

Antonis Papadogiannakis is a Research Assistant in the Distributed Computing Systems Lab at FORTH-ICS. He received the B.Sc. ('05), M.Sc. ('07), and Ph.D. ('14) degrees in Computer Science from the University of Crete, Greece. His main research interests are in the areas of network monitoring, packet capture, network and systems security, privacy, and network measurements.

Michalis Polychronakis is an Associate Research Scientist in the Computer Science Department at Columbia University. He received the B.Sc. ('03), M.Sc. ('05), and Ph.D. ('09) degrees in Computer Science from the University of Crete, Greece, while working as a research assistant in the Distributed Computing Systems Lab at FORTH-ICS. Between 2010-2013, he was a Marie Curie fellow (IOF) at Columbia University and FORTH-ICS. His main research interests are in the areas of network and system security, network monitoring and measurement, and online privacy.

Evangelos P. Markatos is a Full professor of Computer Science at the University of Crete and the director of the Distributed Computing Systems Lab at FORTH-ICS. He received the Diploma in Computer Engineering from the University of Partas in 1988, and the M.Sc. ('90) and Ph.D. ('93) degrees in Computer Science from the University of Rochester, NY. His research interests are in the areas of computing systems, security, and privacy.