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

Antonis Papadogiannakis
FORTH-ICS, Greece
papadog@ics.forth.gr

Michalis Polychronakis
Columbia University, USA
mikepo@cs.columbia.edu

Evangelos P. Markatos
FORTH-ICS, Greece
markatos@ics.forth.gr

ABSTRACT

Many network monitoring applications must analyze traffic beyond the network layer to allow for connection-oriented analysis, and achieve resilience to evasion attempts based on TCP segmentation. However, existing network traffic capture frameworks provide applications with just raw packets, and leave complex operations like flow tracking and TCP stream reassembly to application developers. This gap 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, and can process more than five times higher traffic loads when eight cores are used for parallel stream processing in a pattern matching application.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network Monitoring*

Keywords

Traffic Monitoring; Stream Reassembly; Packet Capturing; Packet Filtering; Overload Control; Performance

1. 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 [36, 40] and pinpoint compromised computers [18, 43]. Similarly, traffic classification tools inspect traffic to identify different communication patterns and spot potentially undesirable traffic [1, 24]. To make meaningful decisions, these monitoring applications usually analyze network traffic at the transport layer and above. For instance, NIDSs reconstruct the transport-layer data streams to detect attack vectors spanning multiple packets, and perform traffic normalization to avoid evasion attacks [14, 19, 38].

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, 28, 29]. Upon receiving the captured packets at user space, monitoring applications usually perform TCP stream reassembly using an existing library such as Libnids [2] or a custom stream reconstruction engine [36, 40]. This results in additional memory copy operations for extracting the payloads of TCP segments and merging them into larger stream "chunks" in contiguous memory. Moreover, it 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, in this paper we present the *Stream capture library (Scap)*, a unified passive network monitoring framework built around the abstraction of the *Stream*. Designed from the beginning for stream-oriented network monitoring, Scap (i) provides the high-level functionality needed by 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.

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 [50], used mostly to avoid evasion attacks against intrusion detection and other monitoring systems, we view streams 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 an-

IMC'13, October 23–25, 2013, Barcelona, Spain.
Copyright 2013 ACM 978-1-4503-1953-9/13/10 ...$15.00.
http://dx.doi.org/10.1145/2504730.2504750.

other, Scap not only avoids redundant packet copies, but also avoids bringing some packets in main memory 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 [9, 26, 27, 33]. 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 [34, 45]. Scap allows applications to (i) define different priorities for different streams and (ii) configure threshold mechanisms that give priority to new and small streams.

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 [22] for transparent parallelization of stream processing.

We have evaluated Scap in a 10GbE environment using real traffic and showed that it outperforms existing alternatives like Libnids [2] and Snort's stream reassembly [40] in a variety of scenarios. For instance, our results demonstrate that Scap can capture and deliver 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 increased CPU utilization for stream reassembly at user level. A single-threaded Scap pattern matching application can handle 33% higher traffic rates than Snort and Libnids, and can process three times more traffic at 6 Gbit/s. When eight cores are used for parallel stream processing, Scap can process 5.5 times higher rates with no packet loss.

The main contributions of this paper are:

1. We introduce the notion of *stream capture*, and present the design, implementation, and evaluation of *Scap* , a stream-oriented network traffic processing framework. Scap provides a flexible and expressive API that elevates streams to first-class objects, and uses aggressive optimizations that allows it to deliver transport-layer streams for two times higher traffic rates than previous approaches.

2. We introduce *subzero packet copy*, a technique that takes advantage of filtering capabilities of commodity NICs to not only avoid copying uninteresting packets (such as the long tails of large flows) across different memory areas, but to avoid bringing them in main memory altogether.

3. 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.

2. DESIGN AND FEATURES

The design of Scap is driven by two key objectives: programming expressiveness and runtime performance. In this section, we introduce the main aspects of Scap across these two dimensions.

2.1 Subzero-Copy Packet Transfer

Several network monitoring applications [9, 26, 27, 33] are interested in analyzing 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 [27]. 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*, or set a different cutoff for each stream direction, allowing for greater flexibility.

Besides a stream cutoff size, monitoring applications may be interested in efficiently discarding other types of less interesting traffic. Many applications often use a BPF filter [28] 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* [26, 33]. Also, depending on the stream reassembly mode used by an 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 [21] 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 [13]. When available, Scap uses FDIR filters to implement all above mentioned cases of early packet discarding. Else, the uninteresting packets are dropped within the OS kernel.

2.2 Prioritized Packet Loss

Scap introduces *Prioritized Packet Loss* (PPL) to enable the system to invest its resources effectively during overload. This is necessary because sudden traffic bursts or overload conditions may force the packet capturing subsystem to fill up its buffers and randomly drop 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 in NIDSs has shown that being able to handle different flows [16, 25, 34], or different parts of each flow [26, 33], in different ways can enable the system to invest its resources more effectively and significantly improve detection accuracy. PPL is a priority assignment technique that enables user applications to define the priority of each stream so that in case of overload, packets from low-priority streams are the first ones to go. User applications can also define a threshold for the maximum stream size under overload (*overload_cutoff*). Then, packets situated 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, the 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 memory used is between $watermark_i$ and $watermark_{i-1}$, PPL makes use of the *overload_cutoff*, if it has been defined by the user. Then, if the packet is located in its stream beyond the *overload_cutoff* byte, it is dropped. In this way, high priority streams, as well as newly created and short streams if an *overload_cutoff* is defined, will be accommodated with higher probability.

2.3 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, which facilitates stream processing operations, and (ii) to perform protocol normalization [19, 51]. Scap currently supports two different modes of TCP stream reassembly: `SCAP_TCP_STRICT` and `SCAP_TCP_FAST`. In the strict mode, streams are reassembled according to existing guidelines [14,51], 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. However, to accommodate for lost segments, 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 the reassembly of a particular chunk.

Scap uses target-based stream reassembly to implement different TCP reassembly policies according to different operating systems. Scap applications can set a different reassembly policy per each stream. This is motivated by previous work, which has shown that stream reassembly performed in a NIDS may not be accurate [38]. For instance, the reconstructed data stream may differ from the actual data stream observed by the destination. This is due to the different TCP reassembly policies implemented by different operating systems, e.g., when handling overlapping segments. Thus, an attacker can exploit such differences to evade detection. Shankar and Paxson [42] developed an active mapping solution to determine what reassembly policy a NIDS should follow for each stream. Similarly to Scap, Snort uses target-based stream reassembly [32] to define the reassembly policy per host or subnet.

Scap also supports UDP: a UDP stream consists of the concatenation of the payloads of the arriving packets of the respective UDP flow. For other protocols without sequenced delivery, Scap return each packet for processing without reassembly.

2.4 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. Using affinity calls, the mapping of threads to CPU cores is practically one-to-one. Scap also dedicates a kernel thread on each core for handling packet reception and stream reassembly. The kernel and worker threads running on the same core process the same streams. As each stream is assigned to only one kernel and worker thread, all processing of a particular stream is done on the same core, reducing, in this way, context switches, cache misses [15, 37], 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) [22], and dynamic load balancing approaches, such as flow director filters (FDIR) [21]. This provides resiliency to short-term load imbalance that could adversely affect application performance. First, Scap detects a load imbalance when one of the cores is assigned a portion of the total streams larger than a threshold. Then, subsequent streams assigned by RSS to this core are re-directed with an FDIR to the core that handles the lowest number of streams at the time.

2.5 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, the stream reassembly operation is performed only once within the kernel, instead of multiple times for each user-level application. If applications have different configurations, e.g., for stream size cutoff or BPF filters, the capture system takes a best effort approach to satisfy all requirements. For instance, it sets the largest among the cutoff sizes for all streams, and keeps streams that match at least one of the filters, marking the applications that should receive each stream and their cutoff.

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 [35]. In contrast, Scap provides user-level applications with reassembled streams instead of randomly interleaved packets, allowing for improved memory locality and reduced cache misses.

3. SCAP API

The main functions of the Scap API are listed in Table 1.

3.1 Initialization

An Scap program begins with the creation of an Scap socket using `scap_create()`, which specifies the interface to be monitored. Upon successful creation, the returned `scap_t` descriptor is used for all subsequent configuration operations. These include setting a BPF filter [28] 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 for processing. 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 a pattern that might span consecutive chunks. The `flush_timeout` parameter can be used to deliver for processing a chunk smaller than the chunk size when this timeout passes, in case the user needs to ensure timely processing.

3.2 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. Each callback

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

Table 1: The main functions of the Scap API.

function takes as a single argument a `stream_t` descriptor `sd`, which corresponds to the stream that triggered the event. 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 handshake or invalid sequence numbers. There is also a pointer to the `stream_t` in the opposite direction, and stream's properties like cutoff, priority, and chunk size.

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 this 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. Using the stream descriptor, the application is able to set the stream's priority, cutoff, and other parameters like stream's 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 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 [41]. In Section 6.5.3 we show that packet delivery does not affect the performance of Scap-based applications. The only difference between Scap's packet delivery and packet-based capturing systems is that packets from the same stream are processed together, due to the chunk-based delivery. As an added benefit, such flow-based packet reordering has been found to significantly improve cache locality [35].

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 processed with very slow rates and delay the application, e.g., due to algorithmic complexity attacks [34, 45]. 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 this adversarial traffic will not affect the application's correct operation.

3.3 Use Cases

We now show two simple applications written with Scap.

3.3.1 Flow-Based Statistics Export

The following listing shows the code of an Scap application for gathering and exporting per-flow statistics. Scap already gathers these statistics and stores them in the `stream_t` structure of each stream, so there is no need to receive any stream data. Thus, the stream cutoff can be set to zero, to efficiently discard all data. All the required statistics for each stream can be retrieved upon stream termination by registering a callback function.

```
scap_t *sc = scap_create("eth0", SCAP_DEFAULT,
    SCAP_TCP_FAST, 0);
scap_set_cutoff(sc, 0);
scap_dispatch_termination(sc, stream_close);
scap_start_capture(sc);

void stream_close(stream_t *sd) {
  export(sd->hdr.src_ip, sd->hdr.dst_ip,
    sd->hdr.src_port, sd->hdr.dst_port,
    sd->stats.bytes, sd->stats.pkts,
    sd->stats.start, sd->stats.end);
}
```

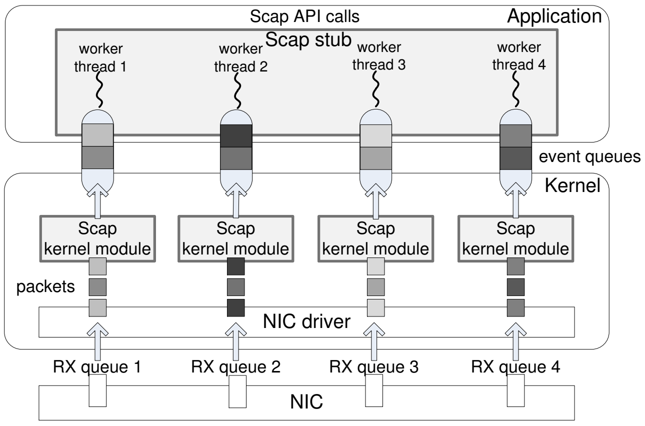


Figure 1: Overview of Scap's architecture.

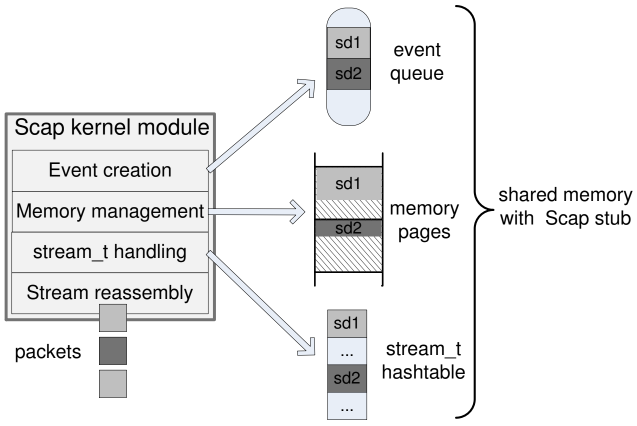


Figure 2: The operation of the Scap kernel module.

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

3.3.2 Pattern Matching

The following listing shows the few lines of code that are required using Scap for an application that searches for a set of known patterns in the captured reassembled TCP streams.

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

We begin by creating an Scap socket without setting a cutoff, so that all traffic is captured and processed (lines 1–2). Then, we configure Scap with eight worker threads, each pinned to a single CPU core (assuming a machine with eight cores), to speed up pattern matching with parallel stream processing. Finally, we register `stream_process()` as the callback function for processing stream chunks (line 4) and start the capturing process (line 5). The `search()` function looks for the set of known patterns within `sd->len` bytes starting from the `sd->data` pointer, and calls the `MatchFound()` function in case of a match.

4. ARCHITECTURE

This section describes the architecture of the Scap monitoring framework for stream-oriented traffic capturing and processing.

4.1 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).

4.2 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 into multiple hardware receive queues. To balance the network traffic load across the available queues and cores, Scap uses both RSS [22], which uses a hash function based on the packets' 5-tuple, and dynamic load balancing, using flow director filters [21], 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 [52].

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 will receive a different subset of network streams, as shown in Figure 1. Consequently, the stream reassembly process is distributed across all the available cores. To match the level of parallelism provided by the Scap kernel module, the Scap's user-level stub creates as many worker threads as the available cores, hiding from the programmer the complexity 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, in this way, context switches, cache misses [15, 37], 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.

5. IMPLEMENTATION

We now give more details on the implementation of the Scap monitoring framework.

5.1 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 the driver schedules them for processing within the software interrupt handler—the Scap's protocol handler in our case.

5.2 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 them 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 without unnecessarily spending further CPU and memory resources for 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 the specified period by creating stream termination events.

5.3 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. 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. Thus, the number of streams that can be tracked concurrently is not limited by Scap.

5.4 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.

5.5 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 this 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 (or TCP acknowledgements), 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 Scap kernel module for stream termination.

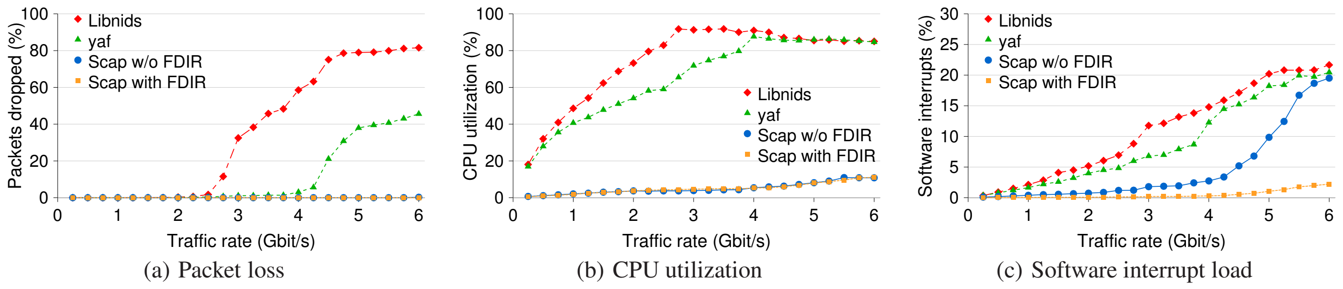
Streams may also be terminated based on an inactivity timeout. For this reason Scap associates a timeout with each filter, and keeps a list with all filters sorted by their timeout values. Thus, an FDIR filter is removed (i) when a TCP RST or FIN packet arrives for a given stream, or (ii) when the timeout 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 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 from NIC's filters. 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 had 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 each 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 [21], which supports RSS and flow director filters. Similarly to this card, most modern 10GbE NICs such as Solarflare [46], SMC [44], Chelsio [10], and Myricom [30], also support RSS and filtering capabilities, so Scap can be effectively used with these NICs as well.

5.6 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 and improve overall performance. To this end, all Scap sockets share a single memory buffer for stream data and



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

Figure 3: Performance comparison of flow-based statistics export for YAF, Libnids, and Scap, for varying traffic rates.

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.

5.7 Packet Delivery

An application may be interested in receiving both reassembled streams, as well as their individual packets, e.g., to detect TCP-level attacks [41]. Scap supports the delivery of the original packets as captured from the network, if an application indicates that it needs them. Then, Scap internally uses another memory-mapped buffer that contains records for each packet of a stream. Each record contains a packet header with the timestamp and capture length, and a pointer to the original packet payload in the stream.

5.8 API Stub

The Scap API stub uses setsockopt() to pass parameters to the kernel module for handling API calls. 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.

6. EXPERIMENTAL EVALUATION

We experimentally evaluate the performance of Scap, comparing it to other stream reassembly libraries, for common monitoring tasks, such as flow statistics export and pattern matching, while replaying a trace of real network traffic at different rates.

6.1 Experimental Environment

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).

The trace.

To evaluate stream reassembly implementations with real traffic, 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.

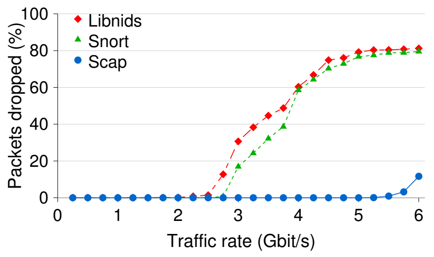
The parameters.

We compare the following systems: (i) Scap, (ii) Libnids v1.24 [2], (iii) YAF v2.1.1 [20], a Libpcap-based flow export tool, and (iv) the Stream5 preprocessor of Snort v2.8.3.2 [40]. YAF, Libnids and Snort rely on Libpcap [29], 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 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. The majority of TCP streams terminate explicitly with TCP FIN or RST packet, but we also use an inactivity timeout to expire UDP and TCP flows that do not close normally. As we replay the trace at much higher rates than its actual capture rate, an inactivity timeout of 10 seconds is a reasonable choice.

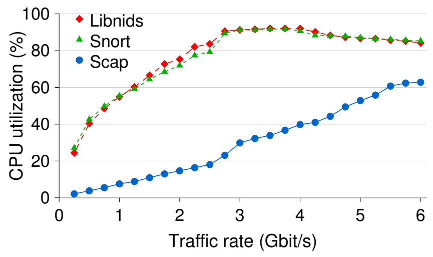
6.2 Flow-Based Statistics Export: Drop Anything Not Needed

In our first experiment we evaluate the performance of Scap for exporting flow statistics, comparing with YAF and with a Libnids-based program that receives reassembled flows. By setting the stream cutoff value to zero, Scap discards all stream data after updating stream statistics. When Scap is configured to use the FDIR filters, the NIC discards all packets of a flow after TCP connection establishment, except from the TCP FIN/RST packets, which are used by Scap for flow termination. Although Scap can use all eight available cores, for a fair comparison, we configure it to use a single worker thread, as YAF and Libnids are single-threaded. However, for all tools, interrupt handling for packet processing in the kernel takes advantage of all cores, utilizing NIC's multiple queues.

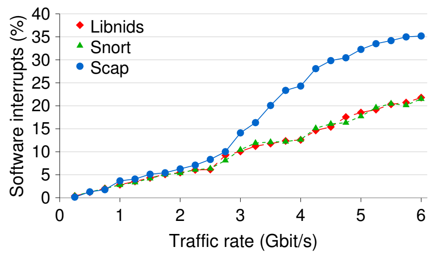
Figures 3(a), 3(b), and 3(c) present the percentage of dropped packets, the average CPU utilization of the monitoring application on a single core, and the software interrupt load while varying the traffic rate from 250 Mbit/s to 6 Gbit/s. We see that Libnids starts losing packets when the traffic rate exceeds 2 Gbit/s. The reason can be seen in Figures 3(b) and 3(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 in user space and then drop them, as the packets themselves are not needed.



(a) Packet loss

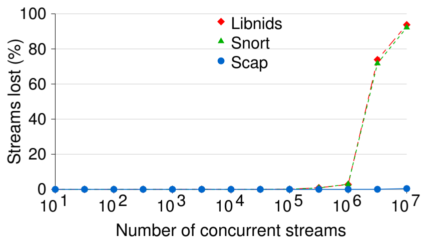


(b) CPU utilization

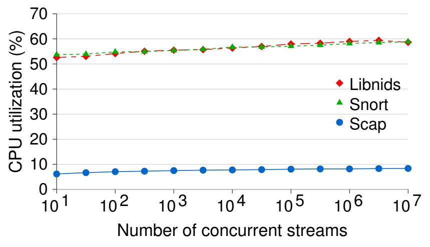


(c) Software interrupt load

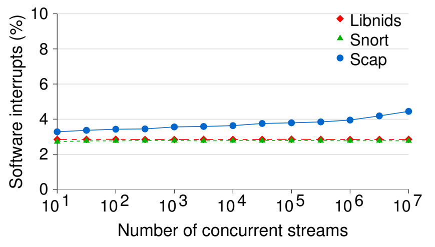
Figure 4: Performance comparison of stream delivery for Snort, Libnids, and Scap, for varying traffic rates.



(a) Lost streams



(b) CPU utilization



(c) Software interrupt load

Figure 5: Performance comparison of Snort, Libnids, and Scap, for a varying number of concurrent streams.

Scap processes all packets even at 6 Gbit/s load. As shown in Figure 3(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 3(c)) to be relatively high. Fortunately, however, the software interrupt load of Scap is even lower compared to YAF and Libnids, even when FDIR filters are not used, 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 it does not perform stream reassembly.

When Scap uses FDIR filters to discard the majority of the packets at NIC layer it achieves even better performance. Figure 3(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.

6.3 Delivering Streams to User Level: The Cost of an Extra Memory Copy

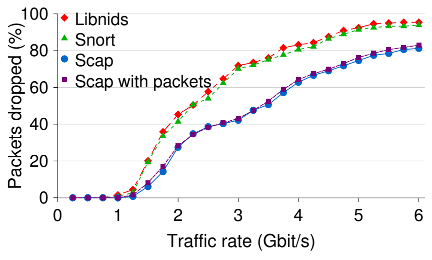
In this experiment, we explore the performance of Scap, Snort, and Libnids when delivering reassembled streams without any further processing. The Scap application receives all data from all streams with no cutoff, and runs as a single thread. Snort is configured with only the Stream5 preprocessor enabled, without any rules. Figure 4(a) shows the percentage of dropped packets as a function of the traffic rate. Scap delivers all streams 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 the 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 have 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 4(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: all the stream reassembly is performed in the kernel module, which increases the software interrupt load, as can be seen in Figure 4(c).

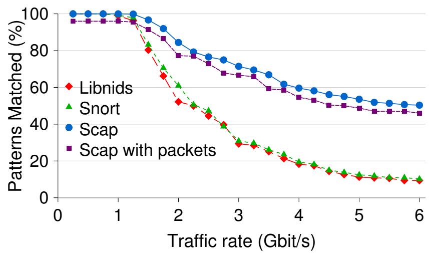
6.4 Concurrent Streams

An attacker could try to saturate the flow table of a stream reassembly library by creating a large number of established TCP flows, so that a subsequent malicious flow cannot be stored. In this experiment, we evaluate the ability of Scap, Libnids, and Snort to handle such cases while increasing the number of concurrent TCP streams up to 10 million. Each stream consists of 100 packets with the maximum TCP payload, and streams are multiplexed so that the desirable number of concurrent streams is achieved. For each case, we create a respective packet trace and then replay it at a constant rate of 1 Gbit/s, as we want to evaluate the effect of concurrent streams without increasing the traffic rate. As in the previous experiment, the application uses a single thread and receives all streams at user level, without performing any further processing.

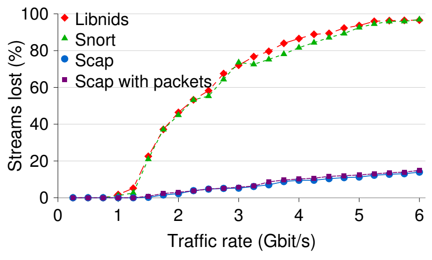
Figure 5 shows that Scap scales well with the number of concurrent streams: as we see in Figure 5(a), no stream is lost even for 10 million concurrent TCP streams. Also, Figures 5(b) and 5(c) show that the CPU utilization and software interrupt load of Scap



(a) Packet loss



(b) Patterns Successfully Matched



(c) Lost streams

Figure 6: Performance comparison of pattern matching for Snort, Libnids, and Scap, for varying traffic rates.

slightly increase with the number of concurrent streams, as the traffic rate remains constant. On the other hand, Snort and Libnids cannot handle more than one million concurrent streams, even though they can handle 1 Gbit/s traffic with less than 60% CPU utilization. This is due to internal limits that these libraries have for the number of flows they can store in their data structures. In contrast, Scap does not have to set such limits because it uses a dynamic memory management approach: when more memory is needed for storing `stream_t` records, Scap allocates dynamically the necessary memory pools to capture all streams. In case an attacker tries to overwhelm the Scap flow table, Scap will use all the available memory for `stream_t` records. When there is no more free memory, Scap's policy is to always store newer streams by removing from the flow table the older ones, i.e., streams with the highest inactivity time based on the access list.

6.5 Pattern Matching

In the following experiments, we measure the performance of Scap with an application that receives all streams and searches for a set of patterns. We do not apply any cutoff so that all traffic is delivered to the application, and a single worker thread is used. Pattern matching is performed using the Aho-Corasik algorithm [5]. We extracted 2,120 strings based on the `content` field of the "web attack" rules from the official VRT Snort rule set [4], and use them as our patterns. These strings resulted in 223,514 matches in our trace. We compare Scap with Snort and Libnids using the same string matching algorithm and set of patterns in all three cases. To ensure a fair comparison, Snort is configured only with the Stream5 preprocessor enabled, using a separate Snort rule for each of the 2,120 patterns, applied to all traffic, so that all tools end up using the same automaton. The Scap and Libnids programs load the 2,120 patterns from a file, build the respective DFA, and start receiving streams. We use the same chunk size of 16KB for all tools.

Figure 6(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 Scap 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 improved cache locality when grouping multiple packets into their respective transport-layer streams, and the reduced memory copies during stream reassembly.

Moreover, Scap 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 matches. As shown in Figure 6(b), under the high load of 6 Gbit/s, Snort and Libnids match less than 10% of the patterns, while Scap matches five times as many: 50.34%. Although the percentage of missed matches for Snort and Libnids is proportional to the percentage of dropped packets, the accuracy of the Scap application is affected less from

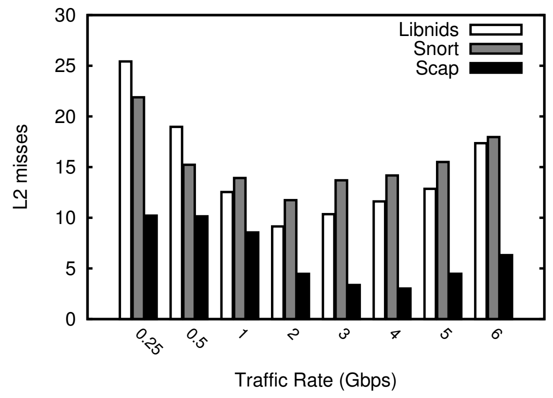


Figure 7: L2 cache misses of pattern matching using Snort, Libnids, and Scap, for varying traffic rates.

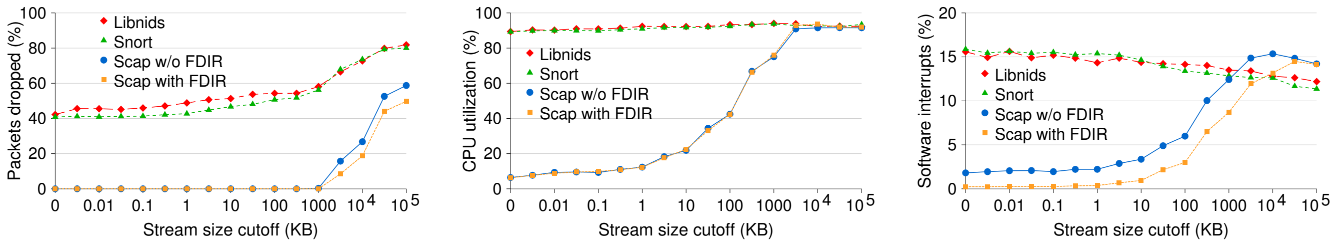
high packet loss rates. This is because Scap under overload tends to retain more packets towards the beginning of each stream. As we use patterns from web attack signatures, they are usually found within the first few bytes of HTTP requests or responses. Also, Scap tries to deliver contiguous chunks, which improves the detection abilities compared to delivery of chunks with random holes.

6.5.1 *Favoring Recent and Short Streams*

We turn our attention now to see how dropped packets affect the different stream reassembly approaches followed by Scap, Libnids, and Snort. While Libnids and Snort drop packets randomly under overload, 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 during the TCP handshake. In contrast, when a packet capture library drops these packets due to overload, the user-level stream reassembly library will not be able to reassemble the respective streams. Indeed, Figure 6(c) shows that the percentage of lost streams in Snort and Libnids is proportional to the packet loss rate. In contrast, Scap 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 lost.

6.5.2 *Locality*

Let's now turn our attention to see how different choices made by different tools impact locality of reference and, in the end, determine application performance. For the previous experiment, we also measure the number of L2 cache misses as a function of the traffic rate (Figure 7), using the processor's performance counters [3]. We see that when the input traffic is about 0.25 Gbit/s,



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

Figure 8: Performance comparison of Snort, Libnids, and Scap, for varying stream size cutoff values at 4 Gbit/s rate.

Snort experiences about 25 misses per packet, Libnids about 21, while Scap experiences *half* of them: just 10.2 misses per packet. We have to underline that at this low traffic rate none of the three tools misses any packets, and we know that none of the tools is stressed, so they all operate in their comfort zone. The reason that Lidnids and Snort have twice as many cache misses as Scap can be traced to the better locality of reference of the Scap approach. By reassembling packets into streams from the moment they arrive, packets are not copied around: consecutive segments are stored together, and are consumed together. On the contrary, Libnids and Snort perform packet reassembly too late: the segments have been stored in (practically) random locations all over the main memory.

6.5.3 Packet Delivery

To evaluate the packet delivery performance of Scap, we ran the same application when Scap was configured with packet support, and pattern matching was performed on the delivered packet payloads. The results are shown in Figure 6 as well. We see that the performance of Scap remains the same when the pattern matching application operates on each packet, i.e., the percentages of dropped packets and lost streams do not change. We just observe a slight decrease in the number of successful matches, which is due to missed matches for patterns spanning the payloads of multiple successive packets.

6.6 Cutoff Points: Discarding Less Interesting Packets Before It Is Too Late

Several network monitoring applications need to receive only the initial part of each flow [26, 33]. Other systems, such as Time Machine [27], elevate the ability to store only the beginning of each flow into one of their fundamental properties. In this experiment, we set out to explore the effectiveness of Libnids, Snort, and Scap when implementing cutoff points. For Snort, we modified Stream5 to discard packets from streams that exceed a given cutoff value. Similarly, when the size of a stream reaches the cutoff value, Libnids stops the collection of data for this stream. In Scap, we just call the `scap_set_cutoff()` function in program's preamble using the desirable cutoff. We also compare Scap with and without using FDIR filters. The applications search for the same set of patterns as in the previous experiment.

Figures 8(a), 8(b), and 8(c) show the packet loss, CPU utilization, and software interrupt load as a function of the cutoff for a fixed traffic rate of 4 Gbit/s. Interestingly, even for a zero cutoff size, i.e., when all data of each flow is discarded, both Snort and Libnids experience as much as 40% packet loss, as shown in the left part of Figure 8(a). This is because Snort and Libnids first bring *all* packets to user space, and then discard the bytes they do not need. Indeed, Figures 8(b) and 8(c) show that the total CPU utilization of Libnids and Snort is always close to 100% at this traffic rate irrespectively of the cutoff point.

In contrast, for cutoff points smaller than 1MB, Scap has no packet loss and very small CPU utilization. For instance, when Scap uses a 10KB cutoff, the CPU load is reduced from 97% to just 21.9%, as 97.6% of the total traffic is efficiently discarded. At the same time, 83.6% of the matches are still found, and no stream is lost. This outcome demonstrates how the stream cutoff, when implemented efficiently, can improve performance by cutting the long tails of large flows, and allows applications to keep monitoring the first bytes of each stream at high speeds. When the cutoff point increases beyond 1MB, CPU utilization reaches saturation and even Scap starts dropping packets. Enhancing Scap with hardware filters reduces the software interrupt load, and thus reduces the packet loss for cutoff values larger than 1MB.

6.7 Stream Priorities: Less Interesting Packets Are The First Ones To Go

To experimentally evaluate the effectiveness of Prioritized Packet Loss (PPL), we ran the same pattern matching application using a single worker thread while setting two priority classes. As an example, we set a higher priority to all streams with source or destination port 80, which correspond to 8.4% of the total packets in our trace. The rest of the streams have the same (low) priority. Figure 9 shows the percentage of dropped packets for high-priority and low-priority streams as a function of the traffic rate. When the traffic rate exceeds 1 Gbit/s, the single-threaded pattern matching application cannot process all incoming traffic, resulting in a fraction of dropped packets that increases with higher traffic rates. However, we see that no high-priority packet is dropped for traffic rates up to 5.5 Gbit/s, while a significant number of low-priority packets are dropped at these rates—up to 85.7% at 5.5 Gbit/s. At the traffic rate of 6 Gbit/s, we see a small packet loss of 2.3% for high-priority packets out of the total 81.5% of dropped packets.

6.8 Using Multiple CPU Cores

In all previous experiments the Scap application ran on a single thread, to allow for a fair comparison with Snort and Libnids, which are single-threaded. However, Scap is naturally parallel and can easily use a larger number of cores. In this experiment, we explore how Scap scales with the number of cores. We use the same pattern matching application as previously, without any cutoff, and configure it to use from one up to eight worker threads. Our system has eight cores, and each worker thread is pinned to one core.

Figure 10(a) shows the packet loss rate as a function of the number of worker threads, for three different traffic rates. When using a single thread, Scap processes about 1 Gbit/s of traffic without packet loss. When using seven threads, Scap processes all traffic at 4 Gbit/s with no packet loss. Figure 10(b) shows the maximum

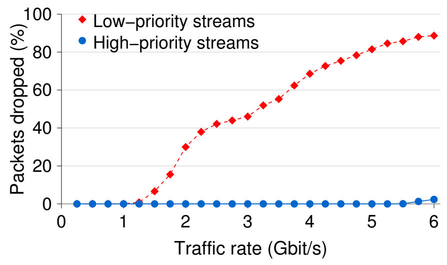
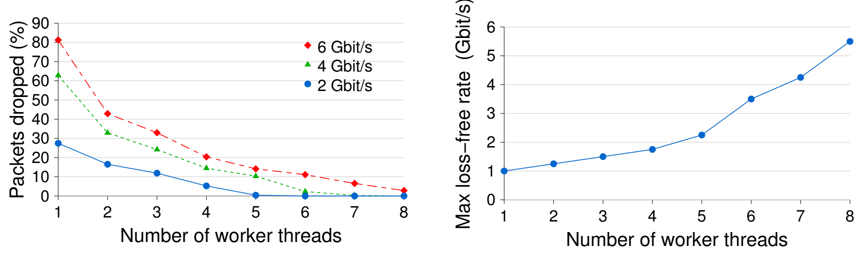


Figure 9: Packet loss for high- and low-priority streams, for varying traffic rates.



(a) Packet loss (b) Speedup

Figure 10: Performance of an Scap pattern matching application for a varying number of worker threads.

loss-free rate achieved by the application as a function of the number of threads. We see that performance improves linearly with the number of threads, starting at about 1 Gbit/s for one worker thread and going all the way to 5.5 Gbit/s for eight threads.

The reason that we do not see a speedup of eight when using eight worker threads is the following: even though we restrict the user application to run on a limited number of cores, equal to the number of worker threads, the operating system kernel runs always on all the available cores of the processor. Therefore, when Scap creates less than eight worker threads, it is only the user-level application that runs on these cores. The underlying operating system and Scap kernel module runs on all cores.

7. ANALYSIS

In this section, we analyze the performance of Prioritized Packet Loss (PPL) under heavy load, aiming to explore at what point PPL should start dropping low-priority packets so that high priority ones do not have to be dropped. For simplicity lets assume that we have two priorities: low and $high$. We define N to be $(memory\_size - base\_threshold)/2$. If the used memory exceeds N, then PPL will start dropping low priority packets. Given that N is finite, we would like to explore what is the probability that N will fill up and we will have to drop high-priority packets. To calculate this probability we need to make a few more assumptions. Assume that high-priority packet arrivals follow a Poisson distribution with a rate of λ, and that queued packets are consumed by the user level application. We assume that the service times for packets follow an exponential distribution with parameter μ. Then, the whole system can be modeled as an $M/M/1/N$ queue. The probability that all the memory will fill up is:

$$P_{full} = \frac{1-\rho}{1-\rho^{N+1}}\rho^N \quad (1)$$

where $\rho = \lambda/\mu$. Due to the PASTA property of the Poisson processes, this is exactly the probability of packet loss: $P_{loss} = P_{full}$.

Figure 11 plots the packet loss probability for high-priority packets as a function of N. We see that a memory size of a few tens of packet slots are enough to reduce the probability that a high-priority packet is lost to 10^{-8}. We note, however, that the speed with which the probability is reduced depends on ρ: the fraction of the high-priority packets over all traffic which can be served by the full capacity of the system. We see that when ρ is 0.1, that is, when only 10% of the packets are high-priority ones, then less than 10 slots are more than enough to guarantee that there will be practically no packet loss. When ρ is 0.5 (i.e., 50% of the traffic is high-priority), then a little more than 20 packet slots are enough, while when ρ is 0.9, then about 150 packet slots are enough.

The analysis can be extended to more priority levels as well. Assume, for example, that we have three priority levels: low, $medium$, and $high$, that $N = (memory\_size - base\_threshold)/3$, that medium-priority packet arrivals follow a Poisson distribution with a rate of λ_1, and that high-priority packet arrivals follow a Poisson distribution with a rate of λ_2. As previously, assume that the service times for packets follow an exponential distribution with parameter μ. Then, the system can be described as a Markov chain with $2N$ nodes:

$$\begin{tikzcd}
\textcircled{1} \arrow[r, bend left, "\lambda_1+\lambda_2"] & \textcircled{2} \arrow[l, bend left, "\mu"] \arrow[r, bend left, "\lambda_1+\lambda_2"] & \cdots \arrow[l, bend left, "\mu"] \arrow[r, bend left, "\lambda_1+\lambda_2"] & \textcircled{N} \arrow[l, bend left, "\mu"] \arrow[r, bend left, "\lambda_2"] & \cdots \arrow[l, bend left, "\mu"] \arrow[r, bend left, "\lambda_2"] & \textcircled{2N} \arrow[l, bend left, "\mu"]
\end{tikzcd}$$

The packet loss probability for high-priority packets is:

$$P_{loss} = \rho_1^N \rho_2^N p_0 \quad (2)$$

where $\rho_1 = (\lambda_1 + \lambda_2)/\mu$, $\rho_2 = \lambda_2/\mu$, and

$$p_0 = 1/(\frac{1-\rho_1^{N+1}}{1-\rho_1} + \rho_1^{N/3}\frac{1-\rho_2^{N+1}}{1-\rho_2})$$

The packet loss probability for medium-priority packets remains:

$$P_{loss} = \frac{1-\rho_1}{1-{\rho_1}^{N+1}}{\rho_1}^N \quad (3)$$

Figure 12 plots the packet loss probability for high-priority and medium-priority packets as a function of N. We assume that $\rho_1 = \rho_2 = 0.3$. We see that a few tens of packet slots are enough to reduce the packet loss probability for both high-priority and medium-priority packets to practically zero. Thus, we believe that PPL provides an effective mechanism for preventing uncontrolled loss of important packets in network monitoring systems.

8. RELATED WORK

In this section we review prior work related to Scap.

8.1 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 [8, 11, 31, 39]. 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.

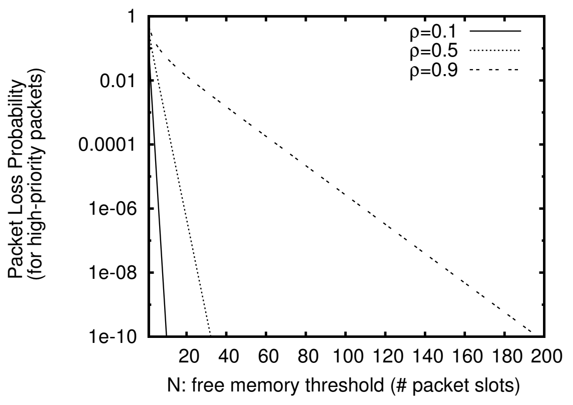


Figure 11: Packet loss probability for high-priority packets as a function of N.

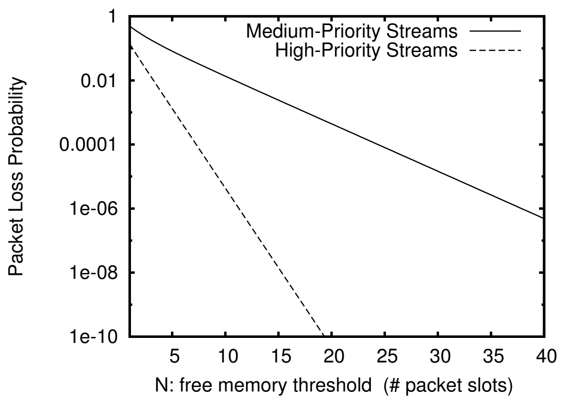


Figure 12: Packet loss probability for high-priority and medium-priority packets as a function of N.

Papadogiannakis et al. [35] 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.

8.2 Taking Advantage of Multi-core Systems

Fusco and Deri [17] utilize the receive-side scaling feature of modern NICs in conjunction with multi-core systems to improve packet capture performance. Sommer et al. [47] 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. [37] improve TCP connection locality in multicore servers 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.

8.3 Packet Filtering

Dynamic packet filtering reduces the cost of adding and removing filters at runtime [12, 49, 53]. Deri et al. [13] 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 [6, 8, 23]. 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.

8.4 TCP Stream Reassembly

Libnids [2] 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 [32] preprocessor, part of Snort NIDS [40], 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 [50], mostly for the avoidance of evasion attacks against intrusion detection systems [14, 19, 51]. 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.

8.5 Per-flow Cutoff

The Time Machine network traffic recording system [27] 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 [26, 33], by focusing detection on the beginning of each connection. Canini et al. [9] 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.

8.6 Overload Control

Load shedding is proposed as a defense against overload attacks in Bro [36], whereby the NIDS operator is responsible for defining a discarding strategy. Barlet-Ros et al. [7] 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. [16] 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.

8.7 Summary

To place our work in context, Figure 13 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 [29] use the *packet* as basic abstraction and are implemented in user level. More sophisticated systems such as netmap [39], FLAME [6], and PF_RING [11] also use the packet as basic abstraction, but are implemented in kernel and deliver better performance. MAPI [48] and FFPF [8] use higher

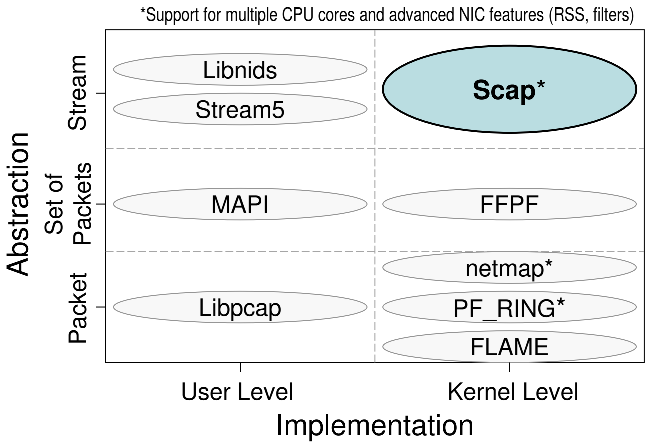


Figure 13: Categorization of network monitoring tools and systems that support commodity NICs.

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.

9. 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 notions 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 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 is able to deliver all streams for rates up to 5.5 Gbit/s using a single core, two times higher than the other existing approaches. An Scap-based application for pattern matching handles 33% higher traffic rates and processes three times more traffic at 6 Gbit/s than Snort and Libnids. Moreover, we observe that user-level implementations of per-flow cutoff just reduce the packet loss rate, while Scap's kernel-level implementation and subzero copy eliminate completely packet loss for stream cutoff values of up to 1MB when performing pattern matching operations at 4 Gbit/s. This outcome demonstrates that cutting the long tails of large flows can be extremely beneficial when traffic is discarded at early stages, i.e., within the kernel or even better at the NIC, in order to spend the minimum possible number of CPU cycles for uninteresting packets. When eight cores are used for parallel stream processing, Scap is able to process 5.5 times higher traffic rates with no packet loss.

As networks are getting increasingly faster and network monitoring applications are getting more sophisticated, we believe that approaches like Scap, which enable aggressive optimizations at kernel-level or even at the NIC level, will become increasingly more important in the future.

Acknowledgements

We would like to thank our shepherd Katerina Argyraki, the anonymous reviewers, and Periklis Akritidis for their valuable feedback. 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.

10. REFERENCES

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

[2] Libnids. `http://libnids.sourceforge.net/`.

[3] Performance application programming interface. `http://icl.cs.utk.edu/papi/`.

[4] Sourcefire vulnerability research team (vrt). `http://www.snort.org/vrt/`.

[5] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18, 1975.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Chelsio Communications. T4 unified wire adapters. `http://www.chelsio.com/adapters/t4_unified_wire_adapters`.

[11] L. Deri. Improving Passive Packet Capture: Beyond Device Polling. In *International System Administration and Network Engineering Conference (SANE)*, 2004.

[12] L. Deri. High-Speed Dynamic Packet Filtering. *Journal of Network and Systems Management*, 15(3), 2007.

[13] 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.

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

[15] 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.

[16] 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.

[17] F. Fusco and L. Deri. High Speed Network Traffic Analysis with Commodity Multi-core Systems. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2010.

[18] 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.

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

[20] C. M. Inacio and B. Trammell. YAF: Yet Another Flowmeter. In *USENIX Large Installation System Administration Conference (LISA)*, 2010.

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

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

[23] 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.

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

[25] 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.

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

[27] 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.

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

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

[30] Myricom. 10G-PCIE-8B-S Single-Port 10-Gigabit Ethernet Network Adapters. `https://www.myricom.com/images/stories/10G-PCIE-8B-S.pdf`.

[31] Myricom. Myricom Sniffer10G. `http://www.myricom.com/scs/SNF/doc/`, 2010.

[32] J. Novak and S. Sturges. Target-Based TCP Stream Reassembly. `http://assets.sourcefire.com/snort/developmentpapers/stream5-model-Aug032007.pdf`, 2007.

[33] 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.

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

[35] 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*, 35(1), 2012.

[36] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24), 1999.

[37] 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.

[38] T. H. Ptacek and T. N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., 1998.

[39] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference (ATC)*, 2012.

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

[41] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP Congestion Control with a Misbehaving Receiver. *ACM SIGCOMM Computer Communication Review*, 29(5), 1999.

[42] U. Shankar and V. Paxson. Active Mapping: Resisting NIDS Evasion without Altering Traffic. In *IEEE Symposium on Security and Privacy*, 2003.

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

[44] SMC Networks. SMC10GPCIe-XFP Tiger Card 10G PCIe 10GbE XFP Server Adapter. `http://www.smc.com/files/AF/ds_SMC10GPCIe_XFP.pdf`.

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

[46] Solarfrare. 10GbE LOM/Controller Family. `http://www.solarflare.com/Content/UserFiles/Documents/Solarflare_SFC9000_10GbE_Controller_Brief.pdf`.

[47] 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*, 21(10), 2009.

[48] 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.

[49] J. Van Der Merwe, R. Caceres, Y. Chu, and C. Sreenan. mmdump: A Tool for Monitoring Internet Multimedia Traffic. *ACM SIGCOMM CCR*, 30(5), 2000.

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

[51] M. Vutukuru, H. Balakrishnan, and V. Paxson. Efficient and Robust TCP Stream Normalization. In *IEEE Symposium on Security and Privacy*, 2008.

[52] S. Woo and K. Park. Scalable TCP Session Monitoring with Symmetric Receive-side Scaling. Technical report, Technical report, KAIST, 2012.

[53] Z. Wu, M. Xie, and H. Wang. Swift: A Fast Dynamic Packet Filter. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.