FloSIS: A Highly Scalable Network Flow Capture System for Fast Retrieval and Storage Efficiency

Jihyung Lee, Sungryoul Lee*, Junghee Lee*, Yung Yi, KyoungSoo Park

Department of Electrical Engineering, KAIST
The Attached Institute of ETRI*

Abstract

Network packet capture performs essential functions in network management such as attack analysis, network troubleshooting, and performance debugging. As the network edge bandwidth exceeds 10 Gbps, the demand for scalable packet capture and retrieval is rapidly increasing. However, existing software-based packet capture systems neither provide high performance nor support flow-level indexing for fast query response. This would either prevent important packets from being stored or make it too slow to retrieve relevant flows.

In this paper, we present FloSIS, a highly scalable, software-based flow storing and indexing system. FloSIS is characterized as the following three aspects. First, it exercises full parallelism in multiple CPU cores and disks at all stages of packet processing. Second, it constructs two-stage flow-level indexes, which helps minimize expensive disk access for user queries. It also stores the packets in the same flow at a contiguous disk location, which maximizes disk read throughput. Third, we optimize storage usage by flow-level content deduplication at real time. Our evaluation shows that FloSIS on a dual octa-core CPU machine with 24 HDDs achieves 30 Gbps of zero-drop performance with real traffic, consuming only 0.25% of the space for indexing.

1 Introduction

Network traffic capture plays a critical role in attack forensics and troubleshooting of abnormal network behaviors. Network administrators often resort to packet capture tools such as tcpdump [6], wireshark [8], netsniff-ng [3] to dump all incoming packets and to analyze the behaviors from multiple dimensions. These tools can help pinpoint a problem in network configuration and performance and even reconstruct the entire trace of an intrusion attempt by malicious hackers.

As the edge network bandwidth upgrades to beyond 10 Gbps, existing packet capture systems have exposed a few fundamental limitations. First, they exhibit poor performance in packet capture and dumping, sometimes unable to catch up with packet transmission rates in a high-speed network. This is mainly because these tools do not properly exploit the parallelism with multiple CPU cores and disks, and the existing kernel is not optimized for high packet rates. Second, most existing tools do not support indexing of stored packets. Lack of indexing would incur a long delay to retrieve the content of interest since the search performance would be limited by sequential disk access throughput, often translating to hours of delay in a multi-TB disk. Third, the huge traffic volume at a high-speed link would significantly limit the monitoring period. For example, it takes less than 17 hours to fill up 24 disks of 3TB at the rate of 10 Gbps. For extended monitoring period, one must enhance the storage efficiency by compressing the stored content.

Recent development of high-performance packet I/O libraries [1, 22, 26, 28] has in part alleviated some of the problems. For instance, n2disk10g [17] and tcpdump-netmap [7] exploit a scalable packet I/O library to achieve a packet capture performance of multi-10 Gbps. Moreover, n2disk10g allows parallel packet dumping to disk and packet-level indexing for fast access. However, the primary problem with these solutions is that they still deal with *packets* instead of network *flows*. Working with packets presents a few fundamental performance issues. First, query processing would be inefficient. Most content-level queries would inspect the packets in the same TCP flow rather than those that belong to completely unrelated flows. Time-ordered packet dumping would scatter the packets in the same flow across a disk. Even with indexing, gathering all relevant packets for a query could either increase disk seeks or waste disk bandwidth from reading unrelated packets nearby the targets. Second, per-packet indexing would be more expensive than per-flow indexing. Per-packet indexing would not only use more metadata disk space but also significantly increase the search time.

In this paper, we present FloSIS (Flow-aware Storage and Indexing System), a highly scalable software-based network traffic capture system that supports efficient flow-level indexing for fast query response. FloSIS is characterized by three design choices. First, it achieves high performance packet capture and disk writing by exercising full parallelism in computing resources such as network cards, CPU cores, memory, and hard disks. It adopts the PacketShader I/O Engine (PSIO) [22] for scalable packet capture, and performs parallel disk write for high-throughput flow dumping. Towards high zero-drop performance, it strives to minimize the fluctuation of packet processing latency. Second, FloSIS generates two-stage flow-level indexes in real time to reduce the query response time. Our indexing utilizes Bloom filters [13] and sorted arrays to quickly reduce the search space of a query. Also, it is designed to consume small amount of memory while it allows flexible queries with wildcards, ranges of connection tuples, and flow arrival times. Third, FloSIS supports flow-level content deduplication in real time for storage savings. Even with deduplication, the system preserves the packet arrival time and packet-level headers to provide exact timing and size information. For an HTTP connection, FloSIS parses the HTTP response header and body to maximize the hit rate of deduplication for HTTP objects.

We find that our design choice brings enormous performance benefits. On a server machine with dual octa-core CPUs, four 10 Gbps network interfaces, and 24 SATA disks, FloSIS achieves up to 30 Gbps for packet capture and disk writing without packet drop. Its indexes take up only 0.25% of the stored content while avoiding slow linear disk search and redundant disk access. On a machine with 24 hard disks of 3 TB, this translates into 180 GB for 72 TB total disk space, which could be managed entirely in memory or stored into solid state disks for fast random access. Finally, FloSIS deduplicates 34.5% of the storage space for 67 GB of a real traffic trace only with 256 MB of extra memory consumption for a deduplication table. In terms of performance, it achieves about 15 Gbps zero-drop throughput with real-time flow deduplication.

We believe that our key techniques in producing high scalability and high zero-drop performance are applicable to other high-performance flow processing systems like L7 protocol analyzers, intrusion detection systems, and firewalls. We show how one should allocate various computing resources for high performance scalability. Also, our efforts for maintaining minimal processing variance should be valuable in high-speed network environments where packet transmission rates vary over time.

2 Design Goals and Overall Architecture

In this section, we highlight our system design goals and describe overall system architecture of FloSIS.

2.1 Design Goals

(1) High scalability: A high-speed network traffic capture system should acquire tens of millions of packets per second from multiple 10G network interfaces, and write them to many hard disks without creating a hotspot. To achieve high performance, the system requires a highly scalable architecture that enhances the overall performance by utilizing multiple CPU cores and hard disks in parallel. Moreover, it should be easily configurable.

(2) High zero-drop performance: The system should ensure high zero-drop performance. A peak zero-drop performance refers to the maximum throughput that the system can achieve without any packet drop. It is an important performance metric of a network traffic capture system since we need to minimize packet drop for accurate network monitoring and attack forensics. It typically differs from the peak throughput that allows packet drop. Even if the input traffic rate is much lower than the peak throughput, a temporary spike of delay caused by blocking I/O calls or scheduling can incur packet drop regardless of the amount of available computation resources.

(3) Flow-level packet processing: Indexing plays a critical role in fast query response. Instead of packet-level indexing [17, 19], we make an index per each flow. Flow-level indexing significantly reduces the index space overhead and improves disk access efficiency. However, it requires managing the packets by their flows so that the packets in the same flow are written to the same disk location. While this incurs extra CPU and memory overheads, it also provides an opportunity to deduplicate the flow content, making more efficient use of the storage.

(4) Flow-aware load balancing: For efficient flow processing, the packets in the same flow need to be transferred to the same CPU core from the NICs while the per-core workload should be balanced on a multicore system. Otherwise, packets should be moved across the CPU cores for flow management, which might cause severe lock contention and degrade CPU cache efficiency.

2.2 Overall Architecture

The basic operation of FloSIS is simple. It captures the network packets mirrored by one or a few switch ports, manages them by their flows [1], and writes them to disk. It also generates per-flow metadata and its associated index entries, and responds to user queries that find a set of matching flows on disk. In case a flow content is redundant, the system deduplicates the flow by having its on-disk metadata point to a version that exists in a disk.

For scalable operation, FloSIS adopts a parallelized pipelined architecture that effectively exploits the parallelism in modern computing resources, and minimizes

[1] We mainly focus on TCP flows in this paper, and non-TCP packets are written to disk as they arrive without flow-level indexing.

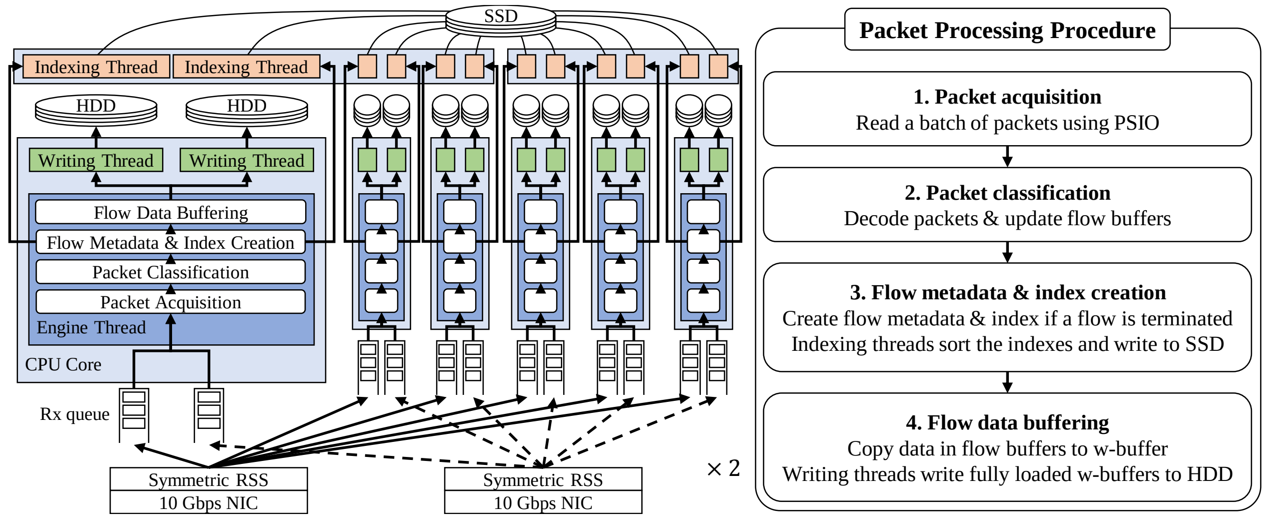
SSD
Indexing Thread
Indexing Thread
HDD
HDD
Writing Thread
Writing Thread
Flow Data Buffering
Flow Metadata & Index Creation
Packet Classification
Packet Acquisition
Engine Thread
CPU Core
Rx queue
Symmetric RSS
10 Gbps NIC
Symmetric RSS
10 Gbps NIC
× 2
Packet Processing Procedure
1. Packet acquisition
Read a batch of packets using PSIO
2. Packet classification
Decode packets & update flow buffers
3. Flow metadata & index creation
Create flow metadata & index if a flow is terminated
Indexing threads sort the indexes and write to SSD
4. Flow data buffering
Copy data in flow buffers to w-buffer
Writing threads write fully loaded w-buffers to HDD

Figure 1: The half of overall architecture and thread arrangement of FloSIS on a machine with 16 CPU cores, four 10G NIC ports, 24 HDDs, and 2 SSDs

the contention among them. It is multi-threaded with three distinct thread types: engine, writing, and indexing threads. The engine thread is responsible for packet acquisition from NICs, flow management, and index generation. It also coordinates flow and index writing to disk with writing and indexing threads. The writing thread shares the buffers for writing with an engine thread, and periodically writes them onto a disk. All packets in the same flow are written consecutively to disk unless they are deduplicated. The indexing thread shares the flow metadata and indexes with an engine thread, and sorts the indexes when it gathers enough entries. Also, it writes the metadata and indexes to disk for durability and helps with resolving the queries by searching through the indexes. We use hard disk drives (HDDs) for storing the flow data and solid state drives (SSDs) for metadata and index information. Our system benefits from cost-effective packet data dumping and prompt query response from fast retrieval of flow metadata and indexes.

Figure 1 shows the mapping of FloSIS threads into a server machine with 16 CPU cores, four 10G NIC ports, and 24 HDDs and 2 SSDs, which we use as a reference platform in this paper. The guiding principle in resource mapping is to parallelize each thread type as much as possible to maximize the throughput while minimizing packet drop from resource contention. Every thread is affinitized to a CPU core to avoid undesirable interference from inter-core thread migration. We co-locate multiple writing threads with an engine thread on the same CPU core since writing threads rarely consume CPU cycles and mostly perform blocking disk operations. Since a writing thread shares the buffers that its engine thread fills in, it would benefit from shared CPU cache if it is co-located with an engine thread. In contrast, an indexing thread runs on a different CPU core since it periodically executes CPU-intensive operations, which might interfere with high-speed packet acquisition with engine threads. Since indexing thread operations are not time-critical, we can place multiple indexing threads on the same CPU core. Each writing thread has its own disk to benefit from sequential disk I/O without concurrent access by other threads. However, SSDs are shared by multiple indexing threads since they do not suffer from performance degradation by concurrent random access.

3 High-speed Flow Dumping

In this section, we describe scalable packet capture and flow dumping of FloSIS. FloSIS uses a fast user-level packet capture library that exploits multiple CPU cores, and performs direct I/O to bypass redundant memory buffering at disk writing while minimizing CPU and memory consumption for disk I/O.

3.1 High-speed Packet Acquisition

The first task of FloSIS is to read packets from NICs. It is of significant importance to allocate enough resource on the packet acquisition, because its performance serves as an upper-bound of the achievable system throughput. Thus, we allocate so many CPU cores as to read the packets from NICs at line rate. For scalable packet I/O, FloSIS employs the PSIO library [22], which allows parallel packet acquisition by flow-aware distribution of incoming packets across available CPU cores. PSIO is known to achieve a line rate regardless of packet size with batch processing and efficient memory management [22]. Also, we use symmetric receive-side scaling (S-RSS) [31] that maps both upstream and downstream packets in the same TCP connection to the same CPU core. S-RSS hashes the 4-tuple of a TCP packet to place the packets in the same connection in the same RX queue inside a NIC. Each RX

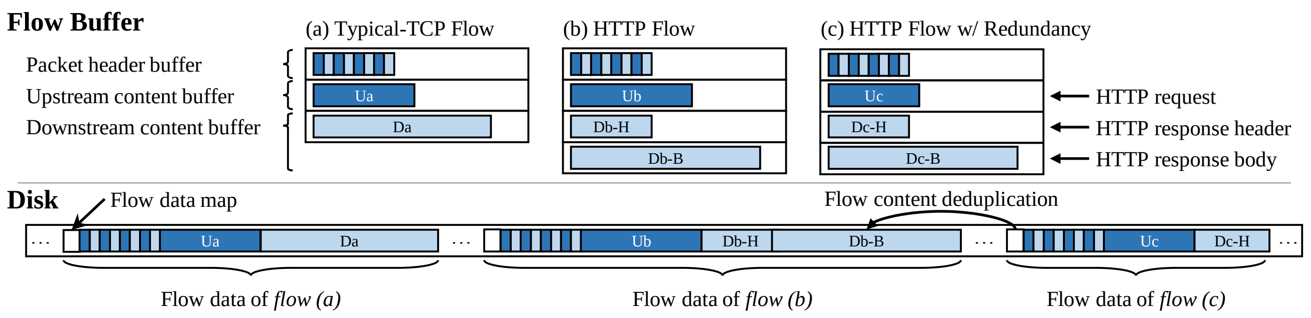

Figure 2: Flow data in the flow buffers and the disk (U: Upstream, D: Downstream, H: Head, B: Body). (a) a typical TCP flow, (b) an HTTP flow with a request in an upstream buffer, and a response in a downstream buffer, and (c) another HTTP flow with the same response body as (b)'s.

queue in a NIC is dedicated to one of the engine threads, and all packets in the queue are processed exclusively by the thread. This architecture eliminates the need of inter-thread synchronization and achieves high scalability without sharing any state across engine threads.

One may have concern that S-RSS may not distribute the packet load evenly across multiple CPU cores. However, a recent study reveals that S-RSS effectively distributes real traffic at a 10 Gbps link almost evenly across multiple CPU cores, by curbing the maximum load difference at 0.2% for a 16-core machine [31].

3.2 Flow-aware Packet Buffering

When packets are read from NIC RX queues, the engine thread manages them by their *flows*. The engine thread classifies the received packets into individual flows, and each flow maintains its current TCP state as well as its packet content. FloSIS keeps all packets in the same TCP connection at a single conceptual buffer called *flow buffer*. A flow buffer consists of three data buffers: packet header, upstream and downstream buffers as shown in Figure 2. The packet header buffer collects all packet headers (*e.g.*, Ethernet and TCP/IP headers) and their timestamps in the order of their arrival. While FloSIS focuses on flow-level indexing and querying, it also supports packet-level operations to provide information such as packet retransmission, out-of-order packet delivery, inter-packet delay, etc, where packet header buffers are needed to reconstruct such information. The upstream and downstream buffers maintain reassembled content of TCP segments so that the user can check the entire message at one disk seek. In case of an HTTP flow, the downstream buffer is further divided into response header and body buffers. This header-body division adds efficiency in deduplication, since the response header tends to differ for the same object (see Section 5 for more details).

One challenge in maintaining the flow buffer is memory management of three data buffers. Since a flow size is variable, one might be tempted to dynamically (re)allocate the buffers. However, we find that dynamic memory allocation often induces unpredictable delays, which increases random packet drops at engine threads. Instead, FloSIS uses user-level memory management with pre-allocated memory pools of fixed-sized chunks. For the flow buffer, one chunk is allocated initially for each data buffer when a flow starts. If more memory is needed, a new chunk is allocated but the new chunk is linked to the previous one with doubly-linked pointers. The flow management module has to follow the links to write the data into right memory chunks. While managing non-contiguous memory chunks is more difficult, the benefit of fixed-cost operations ensures a predictable performance at high-speed packet processing.

When a TCP flow terminates, the engine thread generates a flow metadata consisting of the following information: (i) start and end timestamps, (ii) source and destination [2] IP addresses and port numbers, and (iii) the location and length of the flow data on disk. Then the data in the flow buffer is moved to a large contiguous buffer called *w-buffer*, and the flow buffer is recycled for other flows. The flow data is also moved to w-buffer if its size grows larger than a single w-buffer. The w-buffer serves two purposes. First, it enables to have all flow data at a contiguous location before disk writing, which ensures to read all flow data with one disk seek. Second, it buffers the data from multiple flows to maximize the sequential write throughput of an HDD. It is the unit of disk writing, where a larger size would lead to a higher disk throughput while reducing the CPU usage. When the w-buffer is filled up, the engine thread gives the ownership to its writing thread, which writes to disk and recycles it.

For each flow, a flow data map is written prior to its flow data in w-buffer. The flow data map contains an array of disk locations and lengths of flow data buffers. In most cases, three (or four) flow data buffers follow the flow data map, but for a deduplicated buffer, the flow data map points to an earlier version. If a flow consists of multiple w-buffers, the flow data map is responsible for keeping track of all data buffers in the same flow, allowing flows of arbitrary size.

[2] Interpreted from the client side

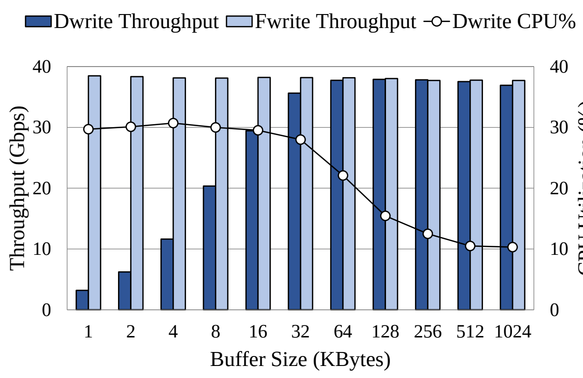

Figure 3: Disk writing throughput and CPU utilization

We comment that FloSIS avoids using buffered I/O, because the file system cache would be of little use for flow dumping and random disk access for user queries. Worse yet, the file system cache behavior becomes unpredictable when the cache buffer is full, which often leads to a catastrophic result such as kernel thrashing. In contrast, direct I/O in FloSIS performs I/O operations by direct DMA-mapping of user buffer to disk, and minimizes the effect of file system cache.

To figure out what size of w-buffer is appropriate as a good design choice in our reference platform, we run a benchmark test with varying buffer sizes. Figure 3 shows disk writing throughputs and CPU utilization of direct I/O using the `O_DIRECT` flag (Dwrite) and buffered I/O using `fwrite()` (Fwrite), where we compare the performance of buffered I/O just to understand the maximum disk performance. In these microbenchmarks, we run 24 threads on our platform, each of which occupies its own HDD and calls `write()` (`fwrite()` in Fwrite) continuously, and measure the aggregate throughputs and CPU utilization. Not surprisingly, Fwrite shows good performance regardless of the buffer size due to its data batching in the file system buffer. Since memory copying to a file system write buffer is much faster than actual disk I/O, the kernel always has enough data to feed to disk. This allows Fwrite to achieve the maximum disk throughput, whereas Dwrite achieves a similar performance only at 128 KB or larger buffer sizes due to the absence of kernel-level buffering. As expected, the CPU usage of Dwrite decreases as the buffer size increases. In FloSIS, we choose 512 KB as the size of w-buffer, which achieves almost peak performance only with 10% CPU usage.

4 Flow-level Indexing & Query Processing

In this section, we explain the two-stage flow-level indexing of FloSIS and real-time index generation.

4.1 Two-stage Flow-level Indexing

FloSIS writes flow data to a file (called a dump file) on each disk. When the current file size reaches a given threshold (*e.g.*, 1GB), FloSIS closes it and moves on to the next file for flow data dumping. The reason for file I/O instead of raw disk I/O is to allow other tools to access the dumped content. For sequential disk I/O, we preallocate the disk blocks contiguously in the files (*e.g.*, using Linux's `fallocate()`).

FloSIS handles flow data stored on each disk by two-stage flow-level indexes as shown in Figure 4. It uses the first-stage indexes to determine the files that contain the queried flows (file indexing). If a dump file has relevant flows, it filters the exact flows using the second-stage indexes (flow indexing). The details are as follows.

File indexing: The first-stage indexing is *file indexes*. Each dump file has in-memory file indexes that consist of two timestamps and four Bloom filters. These indexes are used to determine whether queried flows do **not** exist in the dump file, and the file can be passed. The two timestamps record the arrival times of the first and the last packet stored in the file. Each Bloom filter holds the information about one of the 4-tuple of a flow. For example, the source IP Bloom filter records all source IPs of the flows stored in the dump file. Using the filters, we can quickly figure out whether the dump file does **not** have any flow with a queried IP (or a port number).

Even if a Bloom filter confirms a hit with a queried IP address or a port number, there is still a small probability that it is a false positive. To achieve a tolerable false positive rate, we need to adjust the size of a Bloom filter and the number of hash functions, considering the number of elements. Assuming that an average flow size is 32 KB [31], we expect 32K flows in a dump file of 1 GB. A Bloom filter with 7 hash functions and 128 KB (or 2^{20} bits) of size, would reduce the false positive rate to 0.0011%, which should be small enough. This means that the memory requirement for file indexes is about 1.5 GB per 3 TB HDD or 36 GB for 24 HDDs.

Flow indexing: The second-stage is *flow indexes*. Flow indexes of a dump file consist of all flow metadata of those stored in the file and four sorted arrays. These are used to retrieve the flow metadata entries that match a user query. Using these entries, FloSIS reads the matched flow data from the dump file.

Each sorted array contains one of the flow tuple values in increasing order. An element in the array consists of a tuple value and a pointer to the flow metadata with that value. For example, a sorted array for destination IP has the entries with (a flow's destination IP, a pointer to the flow metadata) for all flows in the dump file, sorted by the destination IPs. When a query determines a target dump file in the first stage, sorted arrays are used to confirm if there are such flows that match the query. Using a binary search, one can locate the flow metadata fast with a specific tuple value in a query.

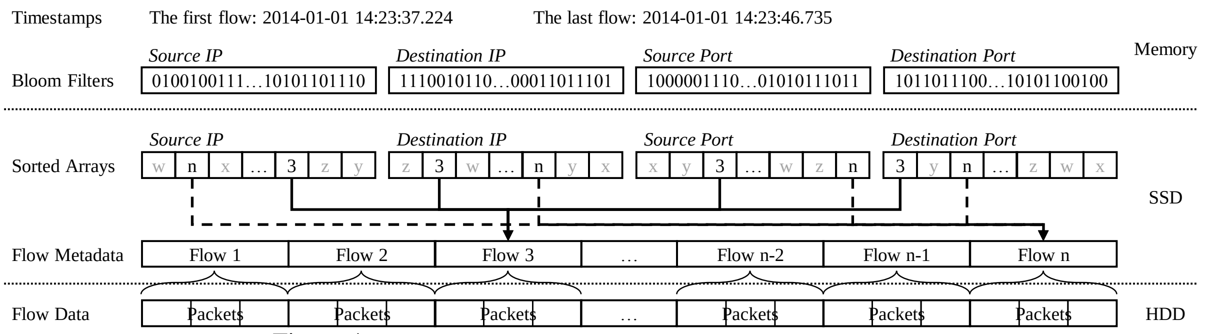

Figure 4: A hierarchical indexing architecture for each file in FloSIS

In terms of the space overhead, assuming a dump file contains 32K flows, four sorted arrays would take up 1 MB, and the array of 32-byte flow metadata would consume another 1 MB. The total space overhead for flow indexes for 24 HDDs of 3 TB would be 140 GB. Since this might be too large to fit in memory, we store these indexes on SSDs, and keep a fraction of them in memory if they are necessary for query processing.

4.2 Real-time Index Generation

To create the two-stage flow-level indexes in real time, FloSIS needs to produce the flow metadata, and updates the Bloom filters and the sorted arrays, for each completed flow with a budget of a few CPU cycles. However, sorting is a memory-intensive operation that consumes many CPU cycles. To avoid the interference caused by sorting, FloSIS separates the index creation in two steps.

In the first step, an engine thread creates the flow metadata with the 4-tuple values, start time, duration, and disk location of the flow when the flow terminates. Then, the engine thread computes Bloom filter hash values of source and destination IP addresses and port numbers, and updates the Bloom filters. Also, it adds an element at the end of each sorted array and leaves the array unsorted. The new element has one of the 4-tuple values and a pointer to the newly-created flow metadata. Sorting of the arrays is postponed until the engine thread fills up the dump file with flow data. Since sorting is skipped, the engine thread creates an index just in a few CPU cycles.

When a dump file is filled up, the second step begins. The engine thread sends an event to an indexing thread to sort the arrays for the 4-tuple values. The indexing thread sorts the arrays, and writes the flow metadata and the sorted arrays into the SSD. While the indexing thread performs CPU-intensive sorting, other threads working on the same core could suffer from a lack of computation cycles, which could affect the overall performance. To prevent this contention, FloSIS runs the indexing threads on dedicated CPU cores. We note that array sorting and index writing happen once every few seconds since it would take at least 5 seconds to fill a dump file of 1 GB, assuming the disk write throughput is 200 MBps. Dedicating a CPU core for each indexing thread would waste CPU cycles, so we co-locate several indexing threads on the same CPU core as shown in Figure 1.

4.3 Flow Query Processing

A FloSIS query consists of all or a part of six fields; a period of time (a time range), source and destination IP addresses and port numbers, and a packet-level condition in the BPF syntax. Each field can be a singleton value or a range of values using an IP prefix, a hyphen (-), or a wildcard (‘*’). FloSIS searches the flows that satisfy every condition in the query, *i.e.*, the query fields are interpreted as conjunctive. Every query field is optional and it is assumed to be any (’*’) in case it is missing in a query.

An example query looks like following.

```
./flowget -st 2015-1-1:0:0:0 -et
2015-1-2:0:0:0 -sip 10.1.2.3 -dip
128.142.33.1/24 -sp 20000-30000 -dp
80 -bpf tcp[tcpflags]&(tcp-syn) != 0
```

`flowget` is a query client program that accepts a user query and sends it to the query front-end server of FloSIS. The query searches the flows that overlap with the time range of one day from January 1st in 2015, and the source IP is 10.1.2.3 with ports between 20000 and 30000 with any destination IPs in 128.142.33.1/24 with port 80. It wants to see only the SYN packets of the matching flows.

When the query front-end server receives the user query, it distributes the query to all indexing threads and collects the results. Each indexing thread processes the query independently using the two-staged indexes of the dump files on their disks. The first step is ‘dump-file filtering’ using the file indexes. In this step, the indexing thread identifies which dump files might have matching flows. The indexing thread first finds a set of dump files whose timestamps overlap with the queried time range, and then checks the Bloom filters to see if the dump file might contain the flows with the requested fields. If a query field is a range, the indexing thread assumes that the corresponding Bloom filter returns a hit since checking all possible values would be too costly.

The second step is ’flow filtering’ using flow indexes. With the target dump files to look up, the indexing thread performs a binary search on each sorted array to find the entries with the requested field. If the field is a range, it first checks whether the array has any overlap with the range, and a lookup for the closest entry to one end of the range would find all matching entries. Assuming the number of entries in an array is 32K, each binary search would require at most 15 entry lookups. That is, 60 entry lookups would be enough to find all matching flows in a dump file in the worst case, which takes a few microseconds on our platform. For each dump file, the indexing thread takes an intersection of the results from the four sorted arrays. After that, the indexing thread retrieves the matching flow metadata entries, and reads the flow data using the disk location in each flow metadata entry.

The last step in query processing is 'packet filtering'. The indexing thread runs BPF for detailed query condition on each packet header in the matching flows. A BPF provides a user-friendly filtering syntax, and is widely used for packet capture and filtering. While this step requires sequential processing, it is applied to a much reduced set of packets compared to brute-force filtering. If a BFP field is missing, then the last step is omitted and all data is passed to the query front-end, which relays it to the query client on a per-flow basis.

5 Flow-level Content Deduplication

It is well-known that the Internet traffic exhibits a significant portion of redundancy in the transferred contents [11, 31]. For example, a recent study reveals that one can save up to 59% of the cellular traffic bandwidth with simple network deduplication with 4KB fixed-sized chunking [31]. Thus, if we apply flow content deduplication to FloSIS, we may expect a non-trivial saving on storage space, which would help to extend the monitoring period at a high-speed network.

In FloSIS, we choose to adopt "whole-flow deduplication," which uses the entire flow data as a unit of redundancy (*i.e.*, a chunk). If the size of a flow exceeds that of the w-buffer, we make each w-buffer for the flow as a chunk. Whole-flow deduplication is a reasonable design choice, given that 80 to 90% of the total redundancy comes from serving the same or aliased HTTP objects [31], and that it consumes a smaller fraction of CPU cycles compared to other alternatives with fine-grained chunking [11]. Note that CPU cycles are important resources in FloSIS that regularly runs CPU-intensive tasks such as packet acquisition and index sorting.

For flow-level deduplication, an engine thread calculates the SHA-1 hash of the flow-assembled content in each direction (or only the response body in case of an HTTP transaction). CPU cycles for hash computation are amortized over the packets in a flow as the hash is partially updated on each packet arrival. This minimizes the fluctuation of CPU usage, which produces more predictable latency in hash computation. Out-of-order packets are buffered until missing packets arrive, and the hash is updated over the partially reassembled data. When a flow finishes (or when the flow size exceeds the w-buffer size), the SHA-1 hash is finalized. The hash is used as the content name to look up in a global deduplication table for cache hit. The deduplication table is a hash table that stores previous content hashes and their disk locations. It is shared by all engine threads to maximize the deduplication rate. When a lookup is a cache hit, the flow data map would be made point to the location of the earlier version when the flow is written to disk. If it is a cache miss, the SHA-1 hash and its disk location is inserted to the table.

We implement the deduplication table so as to minimize CPU usage and maximize the deduplication performance. We use a per-entry access lock to minimize the lock contention due to frequent accesses by multiple engine threads. We also pre-allocate a fixed number of table entries at initialization and use FIFO as the replacement policy (known to perform as well as LRU in network deduplication [31]). Each entry in the table is designed to be as small as possible to maximize the number of in-memory entries: 44 bytes per entry, 20 bytes SHA-1 hash, 8 bytes for disk location, and 16 bytes for doubly-linked pointers. The size of the deduplication table (or simply cache size) is a design parameter that trades off memory usage and deduplication performance. However, it is expected that the cache size does not have to be very big, because it is known that even a small cache significantly helps and the performance improvement diminishes as the cache size increases (i.e., diminishing return) [31, 11]. A recent study reveals that one can achieve 30 to 35% of deduplication rates with only 512 GB of content cache for a 10 Gbps cellular traffic link [31]. This translates to 16 million entries (or 1 GB) in a deduplication table assuming 32KB of average flow size. We understand that the actual numbers would vary in different environments, but we believe that a few GB of table entries should suffice in most cases.

6 Implementation

We implement FloSIS with 7,271 lines of C code that include engine, writing, and indexing threads as well as a query front-end server and a client. We mainly use Linux kernel version 2.6.32-47 for development and testing, but the code does not depend on the kernel version except for the PSIO library that requires Linux kernel 2.6.3x due to its driver code.

The number of engine, writing and indexing threads and their CPU core mapping are configurable depending on the hardware configuration. Each thread is affinitized to a CPU core and we replicate the thread mapping per each non-uniform memory access (NUMA) domain. FloSIS ensures that the network cards deliver the packets only to the CPU cores in the same NUMA domain since crossing NUMA domains is expensive [22].

Since high zero-drop performance is one of our design goals, we pay special attention to the implementation that avoids unpredictable processing delay, whose key techniques are summarized in what follows: First, FloSIS limits maximum processing latency per each batch of packets from a NIC. Normally, an engine thread reads a batch of packets, processes them all, and moves on to read the next batch. However, when the number of packets in a batch is too large, it handles only a fraction of them at a time while leaving the rest in the buffer to minimize packet drop in a NIC. This technique is applied to other implementations such as SHA-1 hash calculation and flow management. Second, FloSIS pins each thread to a CPU core to avoid random scheduling delay and to improve CPU cache utilization. Also, it preallocates performance-critical memory chunks such as data, flow, and write buffers, and core data structures like flow metadata, indexes, flow and deduplication table entries at initialization to avoid the run-time overhead of dynamic memory allocation. FloSIS efficiently recycles the memory space, and dynamically allocates memory only when there is no more available pre-allocated memory space. The amount of pre-allocated memory is configurable, which in our prototype is set based on the recent traffic measurement [31]. Third, FloSIS minimizes unpredictable disk I/O delay as much as possible. We use direct I/O with enough buffer size to saturate the disk I/O capacity, and pre-allocate disk blocks in each file so that they are accessed sequentially. This significantly reduces the variance in disk I/O latency.

For evaluating FloSIS's performance, we extend a network workload generator initially developed for [23]. We implement the workload generator to (i) produce synthetic TCP packets with a random payload at a specified rate up to 40 Gbps regardless of packet size, and (ii) replay a real traffic packet trace (in the pcap file format) at a configurable replay speed. We ensure that the packets in the same flow are forwarded to the same destination NIC port in the order of their original record.

7 Evaluation

The goals of our evaluation are three folds. First, we evaluate if FloSIS provides a good performance [3] of packet- and flow-level capture and disk dumping with synthetic and real traffic. Second, we show how much reduction in query response time FloSIS's two-stage flow-level indexing brings over the existing state-of-the-art packet-level indexing. Third, we evaluate the effectiveness of flow content deduplication with a real traffic trace and measure the overhead.

[3]Unless specified otherwise, the throughput numbers include Ethernet frame overhead (24 bytes) to reflect the actual line rate.

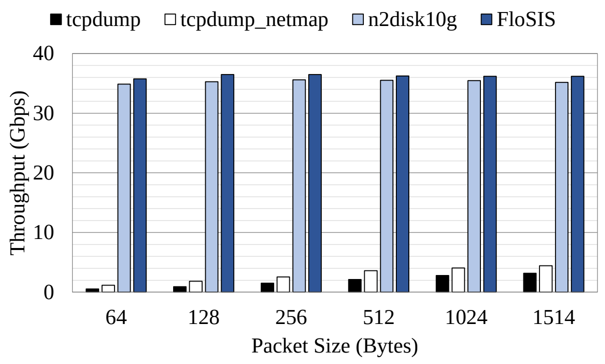

Figure 5: Synthetic TCP packet dumping throughputs of FloSIS, n2disk10g, and tcpdump at 40 Gbps input rate with various packet sizes

7.1 Experiment Setup

We run FloSIS and other packet capture systems on our reference server machine with dual Intel E5-2690 CPUs (2.90 GHz, 16 cores in total), two dual-port Intel 10 Gbps NICs with 82599 chipsets, 128 GB of RAM, 24 3TB SATA HDDs of 7200 RPM, and 2 SSDs of 512 GB Samsung 840 Pro. For workload generation, we run our packet generator on a similar machine with the same number of CPUs and NICs as the server. For synthetic workload, the packet generator sends the packets of the same size at a specified rate. For real traffic tests, the packet generator replays 67.7 GBs of a packet trace obtained from a large cellular ISP in South Korea [31]. This trace represents a few minutes of real traffic at one of 10 Gbps cellular backhaul links, and has 89 million TCP packets with 760 bytes of average packet size. The client and server machines are directly connected by four 10G cables since the traffic is either synthetically generated or replayed. However, in practice, the live traffic should be port-mirrored to the server via a switch.

7.2 Packet Capture & Dumping

We measure the performance of traffic capture and disk dumping of FloSIS for synthetic and real traffic workloads and compare them with those of existing solutions.

7.2.1 Synthetic Packet Workload

We first evaluate whether FloSIS achieves a good performance with packet capture and disk dumping regardless of incoming packet size. We have the packet generator transmit the packets of the same size at 40 Gbps, and measure the disk writing performance of captured packets. To compare the performance of packet-level capture and disk dumping with other tools, we disable other features of FloSIS such as flow management, index generation and writing, and deduplication.

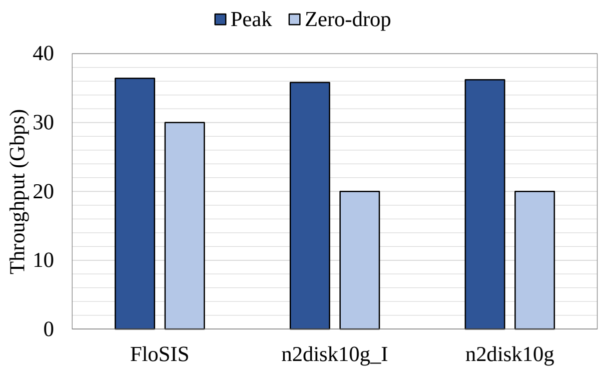

Figure 6: Peak and zero-drop throughput of FloSIS and n2disk10g with real traffic trace replay

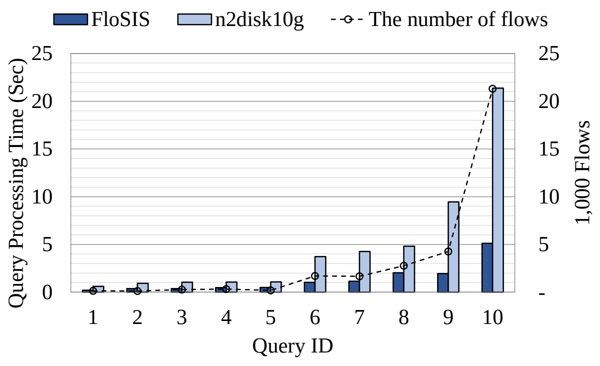

Figure 7: Query processing times over various number of matching flows

We compare the performances with tcpdump [6], tcpdump-netmap [7], and n2disk10g [17]. tcpdump is a popular packet capture system based on the libpcap library, but the libpcap library allows only one process/thread for each network interface. Since our machine has four 10G interfaces, we run 4 tcpdump processes and have each process write the captured packets to its dedicated HDD. On the other hand, tcpdump-netmap uses the netmap packet I/O framework [28] to accelerate the libpcap library performance. n2disk10g [17] uses PF_RING DNA [26] as scalable packet acquisition library, and can be configured to write to multiple disks in parallel. We select the parameters suggested by the manual of n2disk10g [2] for the best behavior. For fair comparison, we disable the indexing module for this test.

Figure 5 shows the throughputs [4] of the systems. FloSIS achieves 35.8 to 36.5 Gbps regardless of packet size, which is close to the peak aggregate disk writing performance as shown in Figure 3. This implies that FloSIS's packet buffering works well to achieve a sequential disk write performance. We observe that n2disk10g performs as well (34.9 to 35.6 Gbps), which is not surprising since its packet capture and parallel disk I/O is similar to that of FloSIS. Unfortunately, tcpdump and tcpdump-netmap perform poorly, achieving only 0.4 to 3.2 Gbps and 0.9 to 4.4 Gbps, respectively. While the netmap support improves the performance of tcpdump, the inefficiency in the libpcap library itself seems to limit the improvement.

7.2.2 Real Traffic Workload

We now measure the performance of handling the real traffic by replaying the LTE packet trace at a high speed. For this test, we enable flow management and indexing in FloSIS but disable flow deduplication to focus on the performance of flow processing. For comparison, we show the performances of n2disk10g with and without packet-level indexing. We also measure the peak zero-drop performances of both systems to estimate the practical performance for accurate monitoring.

As depicted in Figure 6, the peak performances of FloSIS and n2disk10g with/without indexing are similarly high as 36.4 Gbps, implying that disk writing bandwidth is the primary bottleneck, as in the synthetic workload case. This, in turn, implies that the extra overhead for flow management and two-stage flow index generation is small enough not to degrade the overall performance. However, FloSIS achieves better than n2disk10g in terms of zero-drop performance, 30 Gbps by FloSIS and 20 Gbps by n2disk10g regardless of indexing. We believe that our design choices such as minimizing the contention by separating the resources between interfering tasks, and aggressive amortization of resource consumption help produce more predictable processing delays, despite extra tasks like flow management and indexing. We do not know the root cause for lower zero-drop performance with n2disk10g since we do not have access to the source code, but it seems to pay less attention to the latency burstiness in packet capturing and disk dumping.

7.3 Query Processing

We evaluate the effectiveness of flow-level indexing of FloSIS. For the experiments, we replay the LTE traffic trace, and issue 10 queries to retrieve all flows between two IP addresses randomly chosen by us. The number of matching flows ranges from 119 to 21,300, and they are scattered around the disks. We compare the query response times with those of n2disk10g. The indexing of n2disk10g similar to that of FloSIS in that it uses tuple-based Bloom filters to skip the dump files, but it performs linear search on per-packet indexes called packet digests. Per-packet indexing not only incurs more space overhead but also requires more disk seeks to read the packets scattered around the disks.

Figure 7 compares the response times of 10 queries. We find that FloSIS outperforms n2disk10g by a factor of 2.2 to 4.9. As the number of flows increases, the per-

[4]We do not include the Ethernet frame overhead in throughput calculation in this test to focus on the performance of disk writing rather than packet capturing.

Queries	Q1	Q2	Q3	Q4
Bloom filter	100%	0.05%	-	-
Sorted array	-	10.83%	100%	99.43%
Flow read	-	89.12%	-	0.57%
Latency	3.3ms	330.2ms	80.5s	82.2s
Response	0B	2.7KB	0B	2.7KB

Table 1: Query processing times of singleton and range queries with/without disk access to read indexes and flow data

formance gap widens since n2disk10g suffers from more disk activity to gather all data. Unfortunately, we could not fill more data to examine the performance difference further since n2disk10g that we have is an evaluation version that runs for only a few minutes.

To investigate the FloSIS behavior with more flow data, we disable deduplication in FloSIS and fill the LTE trace repeatedly until we have 10 TBs of stored data. During the data filling phase, we feed in a few random flows that we retrieve in our queries. We use four types of queries to evaluate the effectiveness of our indexing structure. We use two singleton (Q1, Q2) and two range queries (Q3, Q4) for the entire time period. One of the two queries in each query type (Q1, Q3) looks for a flow that does not exist on disk, while the other queries (Q2, Q4) would return a small flow (2.7 KB) as a response.

Table 1 shows the response times for all queries. Q1 takes only 3.3 milliseconds (ms) since it would require checking only the Bloom filters in memory. Q2 takes more time (330.2 ms) since it has to read in the sorted arrays and flow metadata for the matching dump file from an SSD after cache hits with the Bloom filters. It also needs to read in the flow data from an HDD. Q3 and Q4 take much longer, 80.5 and 82.2 seconds, respectively, since FloSIS skips the Bloom filters for range queries, and reads in sorted arrays and flow metadata for all dump files. Since there are about 10,000 dump files for 10 TBs of data, the total size of all sorted arrays and flow metadata would be 20 GB. Actually, we could optimize the behavior further by reading the sorted arrays first, and read the flow metadata only if the query is a hit with the sorted arrays. Without indexing, it would require reading all data from HDDs which could take at least a few hours to resolve the queries.

7.4 Deduplication

We evaluate flow content deduplication with the real traffic trace. Deduplication is the most CPU-intensive task in FloSIS due to the overhead of real-time content hashing. We see almost full CPU utilization when we enable deduplication at a high traffic rate. While we expect to improve the performance in a cost-effective manner with SHA-1 computation offloading to off-chip GPUs [24], we focus on the CPU-only performance here. We also measure the level of storage compression by deduplication as we use more deduplication table entries.

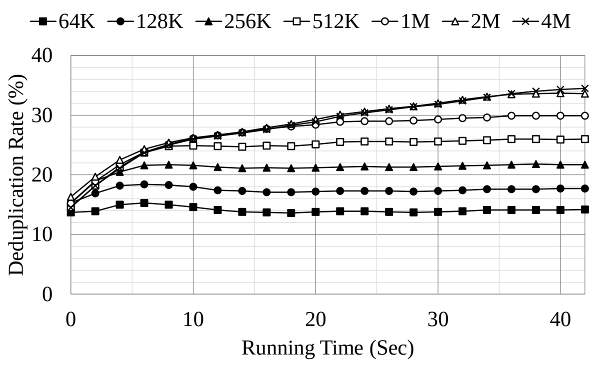

Figure 8: Deduplication rates over various numbers of deduplication table entries

FloSIS achieves about 15 Gbps of the peak zero-drop performance with deduplication. The performance is almost halved from the peak zero-drop performance without deduplication (in Figure 6), but its performance reaches 75% of the peak zero-drop performance of n2disk10g. Even if we amortize the latency of SHA-1 hashing over time, even a short spike of SHA-1 computation delay induces random packet drop, which drags the zero-drop performance. Nevertheless, we believe that one can monitor a full 10 Gbps link with deduplication without worrying about packet drop.

Finally, we estimate how many deduplication table entries are needed for a good deduplication rate. This question is difficult to answer given that we have only 67 GB of real traffic workload. For this workload, we draw a graph that shows the deduplication rates over various deduplication table size. Figure 8 presents the results over the entire period of traffic replay at 13.5 Gbps. The trend we find here is that (i) even a small cache works very effectively: A table with only 64K entries (or just 4 MB of content cache on disk) provides about 14% deduplication rate, (ii) it requires almost eight times more entries to double the deduplication rate, (iii) once the entries are filled up, the deduplication rate stabilizes over time. We obtain 34.5% of deduplication rate with 4 million entries, which is enough to suppress the actual redundancy in the original data. We do note that our trace is too small to draw any definitive conclusions, but we observe that a much larger workload shares a few similarities as our workload [31]. Given that 512 GB of content cache with 4KB chunks (or 16 million entries in the deduplication table) gives 30 to 35% of deduplication rate at a busy 10 Gbps link, we believe that a few GB of entries would be enough to achieve a good performance in practice.

8 Related Work

We briefly discuss the related work here. Due to a large body of works, comprehensive coverage is difficult, so we attempt to provide a summary of representative works.

High-speed packet acquisition and writing: Scalable and fast packet capture heavily affects the performance of network security monitoring systems such as firewalls and intrusion detection/prevention systems (IDS/IPS). High-end security systems typically adopt a parallel packet acquisition architecture that exploit the parallelism in modern CPUs and network cards. For example, software-based IDSes such as SnortSP [4], Suricata [5] and Para-Snort [14] deploy parallelized pipelining architectures suitable for a multi-core environment. Gnort [20], MIDeA [30] and Kargus [23] adopt a similar parallelized architecture for fast packet capture, but use general-purpose GPUs to offload the heavy pattern matching operations for extra performance improvement. n2disk10g [17] is a recent high-speed packet capture and storing system that adopts PF_RING DNA for packet acquisition, and uses direct disk I/O for scalable disk throughput. FloSIS shares the parallel packet capture architecture with these systems, but it goes beyond scalable packet capture and shows how one should map heterogeneous tasks of different computation budget (e.g., engine, writing, indexing threads) to computing resources for high zero-drop performance.

Intelligent indexing for fast retrieval: It is of critical importance to have an intelligent indexing structure for fast query response. pcapIndex [19] uses compressed bitmap indexes to index pcap files at off-line, and supports user queries with the BPF syntax. Hyperion [18] presents a stream file system optimized for sequential immutable streaming writes to store high-volume data streams, and proposes a two-level index structure based on Bloom filters. Gigascope [16] supports SQL-like queries on a packet stream, however, without archiving long-duration data. n2disk10g [17] supports per-packet indexing based on the Bloom filter and a packet metadata called packet digest. However, the per-packet indexing structures are used only for rough filtering of the entire data and it still requires linear search for query processing, often causing a long delay. FloSIS adopts flow-level indexing and query processing, which eliminates linear disk search and minimizes disk access in query processing. The required storage space for indexing is much smaller than that of packet-level indexing.

Network Traffic Compression: There have been many works on network traffic deduplication [9, 10, 11, 12, 29, 31]. These works show that typical Internet traffic has significant content-level redundancy. In terms of duplicate suppression, a smaller chunking unit and content-based chunking algorithm like Rabin's fingerprinting [27] generally leads to a higher deduplication rate at the cost of a larger computation overhead. To the best of our knowledge, FloSIS is the first traffic capture system that employs deduplication, and our choice of whole-flow deduplication is reasonable given the trade-off of computation overhead and the level of storage savings.

Cooke *et al.* present a multi-format storage that stores detailed information for recent data, but maintains only the summary of old data as time goes by [15]. Horizon Extender [21] proposes Web traffic classification and storing based on white-listing, considering the Web service popularity. It mainly focuses on compressing stored HTTP data for storage savings while minimizing the loss of data required to find the evidence of data leakage. Time Travel [25] stores only the first part of the large flows considering the heavy-tailed nature of network traffic. While these works reduce the storage requirement at the cost of losing a fraction of the data, FloSIS focuses on lossless storing of full flow data at a high-speed network for accurate traffic monitoring. However, we believe a hybrid approach is possible for further storage savings.

9 Conclusion

As the network edge bandwidth exceeds 10 Gbps, the demand for scalable packet capture and retrieval, used for attack analysis, network troubleshooting and performance debugging, is rapidly increasing. However, existing software-based packet capture systems neither provide high performance nor support flow-level indexing for fast query response. In this paper, we have proposed a highly scalable, software-based flow storing and indexing system, FloSIS, characterized by three features: exercising full parallelism, flow-aware processing for minimizing expensive disk access for user queries, and deduplication for efficient storage usage.

We have demonstrated that FloSIS achieves up to 30 Gbps of zero-drop performance without deduplication, and 15 Gbps with deduplication with real traffic storing and indexing, at the indexing storage cost of only 0.25% of the stored data. The two-stage flow-level indexing of FloSIS completes searching and reading 2.7 KB of flow data from 10 TB in 330.2 ms. Finally, our flow content deduplication reduces 34.5% of storage space for 67 GB of the real traffic trace with 256 MB of additional memory consumption.

Acknlowdgments

We would like to thank Shinjo Park and Sanghyeok Chun for their help on the initial design of the system. We also thank the anonymous reviewers and our shepherd, Keith Adams for valuable comments on the paper. This work was supported in part by National Security Research Institute grants #N04130037, #N04140063, Institute for Information & communications Technology Promotion (IITP) [B0126-15-1078], and the ICT R&D program of MSIP/IITP, Republic of Korea [B0101-15-1368, Development of an NFV-inspired networked switch and an operating system for multi-middlebox services].

References

[1] Intel DPDK: Data Plane Development Kit. http://dpdk.org/.

[2] n2disk User's Guide. http://www.ntop.org/.

[3] netsniff-ng. http://netsniff-ng.org/.

[4] SnortSP (Security Platform). http://www.snort.org/snort-downloads/snortsp/.

[5] Suricata Intrusion Detection System. http://www.openinfosecfoundation.org/index.php/download-suricata.

[6] Tcpdump & Libpcap. http://www.tcpdump.org/.

[7] Tcpdump: netmap support for lipcap. http://seclists.org/tcpdump/2014/q1/9/.

[8] Wireshark. http://www.wireshark.org/.

[9] AGARWAL, B., AKELLA, A., ANAND, A., BALACHANDRAN, A., CHITNIS, P., MUTHUKRISHNAN, C., RAMJEE, R., AND VARGHESE, G. Endre: An end-system redundancy elimination service for enterprises. In *Proceedings of USENIX NSDI* (2010).

[10] ANAND, A., GUPTA, A., AKELLA, A., SESHAN, S., AND SHENKER, S. Packet caches on routers: The implications of universal redundant traffic elimination. In *Proceedings of ACM SIGCOMM* (2008).

[11] ANAND, A., MUTHUKRISHNAN, C., AKELLA, A., AND RAMJEE, R. Redundancy in network traffic: Findings and implications. In *Proceedings of ACM SIGMETRICS* (2009).

[12] ANAND, A., SEKAR, V., AND AKELLA, A. SmartRE: an architecture for coordinated network-wide redundancy elimination. In *Proceedings of ACM SIGCOMM* (2009).

[13] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM 13*, 7 (July 1970), 422–426.

[14] CHEN, X., WU, Y., XU, L., XUE, Y., AND LI, J. Para-Snort: A Multi-thread Snort on Multi-core IA Platform. In *Proceedings of Parallel and Distributed Computing and Systems (PDCS)* (2009).

[15] COOKE, E., MYRICK, A., RUSEK, D., AND JAHANIAN, F. Resource-aware multi-format network security data storage. In *Proceedings of ACM SIGCOMM workshop on Large-scale attack defense* (2006).

[16] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: a stream database for network applications. In *Proceedings of ACM SIGMOD* (2003).

[17] DERI, L., CARDIGLIANO, A., AND FUSCO, F. 10 gbit line rate packet-to-disk using n2disk. In *Proceedings of IEEE INFOCOM Workshop on Traffic Monitoring and Analysis* (2013).

[18] DESNOYERS, P. J., AND SHENOY, P. Hyperion: high volume stream archival for retrospective querying. In *Proceedings of USENIX ATC* (2007).

[19] FUSCO, F., DIMITROPOULOS, X., VLACHOS, M., AND DERI, L. pcapIndex: an index for network packet traces with legacy compatibility. *ACM SIGCOMM Computer Communications Review 42*, 1 (Jan. 2012), 47–53.

[20] G. VASILIADIS, S. ANTONATOS, M. POLYCHRONAKIS, E. P. MARKATOS, AND S. IOANNIDIS. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)* (2008).

[21] GUGELMANN, D., SCHATZMANN, D., AND LENDERS, V. Horizon extender: long-term preservation of data leakage evidence in web traffic. In *Proceedings of ACM CCS* (2013).

[22] HAN, S., JANG, K., PARK, K., AND MOON, S. B. PacketShader: a GPU-accelerated software router. In *Proceedings of ACM SIGCOMM* (2010).

[23] JAMSHED, M. A., LEE, J., MOON, S., YUN, I., KIM, D., LEE, S., YI, Y., AND PARK, K. Kargus: A highly-scalable software-based intrusion detection system. In *Proceedings of ACM CCS* (2012).

[24] JANG, K., HAN, S., HAN, S., MOON, S. B., AND PARK, K. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of USENIX NSDI* (2011).

[25] MAIER, G., SOMMER, R., DREGER, H., FELDMANN, A., PAXSON, V., AND SCHNEIDER, F. Enriching network security analysis with time travel. In *Proceedings of ACM SIGCOMM* (2008).

[26] NTOP. Libzero for DNA: Zero-copy flexible packet processing on top of DNA. `http://www.ntop.org/products/pf_ring/libzero-for-dna/`.

[27] RABIN, M. O. Fingerprinting by random polynomials. Tech. Rep. TR-15-81, Harvard University, 1981.

[28] RIZZO, L. netmap: a novel framework for fast packet I/O. In *Proceedings of USENIX ATC* (2012).

[29] SPRING, N. T., AND WETHERALL, D. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM* (2000).

[30] VASILIADIS, G., POLYCHRONAKIS, M., AND IOANNIDIS, S. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In *Proceedings of ACM CCS* (2011).

[31] WOO, S., JEONG, E., PARK, S., LEE, J., IHM, S., AND PARK, K. Comparison of caching strategies in modern cellular backhaul networks. In *Proceedings of ACM MobiSys* (2013).