pcapIndex: An Index for Network Packet Traces with Legacy Compatibility

Francesco Fusco
IBM Research, ETH Zurich
ffu@zurich.ibm.com

Xenofontas Dimitropoulos
ETH Zurich
fontas@tik.ee.ethz.ch

Michail Vlachos
IBM Research

Luca Deri
ntop
deri@ntop.org

ABSTRACT

Long-term historical analysis of captured network traffic is a topic of great interest in network monitoring and network security. A critical requirement is the support for fast discovery of packets that satisfy certain criteria within large-scale packet repositories. This work presents the first indexing scheme for network packet traces based on compressed bitmap indexing principles. Our approach supports very fast insertion rates and results in compact index sizes. The proposed indexing methodology builds upon *libpcap*, the defacto reference library for accessing packet-trace repositories. Our solution is therefore backward compatible with any solution that uses the original library. We experience impressive speedups on packet-trace search operations: our experiments suggest that the index-enabled *libpcap* may reduce the packet retrieval time by more than 1100 times.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network monitoring

General Terms

Algorithms, Measurement, Performance, Security

Keywords

packet indexing

1. INTRODUCTION

Diverse network management and network security processes require efficient search and filtering operations over large-scale network-trace repositories. For example, in network-forensics operators need tools that can facilitate the effective scanning of stored packet-traces for mapping intrusion activities, e.g., machines contacted by a compromised node within a company's infrastructure. Similar functionalities are required for validating claims of Service Level Agreement (SLA) violations, for troubleshooting failures, and in general, for performing any traffic analysis task that cannot be performed over a live network stream, but requires access to a historical packet-trace repository.

Packet-trace analysis solutions use the same filtering mechanism when filtering packets either from a network interface (i.e., live capture) or from a packet-trace. The defacto reference packet filtering mechanism is the Berkeley Packet Filter (BPF) [10]. BPF implementations are provided by the large majority of operating systems and packet capture libraries, such as the widely-used *libpcap* [2], on top of which the ubiquitous *tcpdump* is also built. BPF filters packets from a packet trace using *filtering expressions*, which are posed in the BPF language [10]. Filters are evaluated against each packet. Packet filtering operations are costly to perform because they require the complete (linear) scan of the packet-trace repository with the subsequent filtering of packets that satisfy the search criteria. Therefore, search operations are plagued by long execution times. Surprisingly, current packet-trace analysis solutions do not provide support for indexing schemes.

This work presents an index-based packet filtering architecture that enables fast packet searching within packet-trace repositories, e.g., queries like: find all packets from subnet 192.168/16 with destination port 445/udp. The proposed filtering architecture has been designed with legacy compatibility in mind, and does not require existing packet-trace repositories to be re-encoded in a new format. More importantly, the architecture is modular and easily extensible with packet indexing plugins. We augment the popular *libpcap* with our indexing architecture. In this manner, we preserve backward compatibility with the rich pool of packet-trace analysis software already built upon pcap. However, our design of the extended libpcap can be generalized for other packet formats, as well, like the ERF format used by Endace[1]. This can be achieved with limited effort because the majority of packet trace formats share many common characteristics. The **contributions** of this work are as follows:

- We highlight shortcomings of the BPF filters when used for searching packet-traces. We propose the use of compressed bitmap indexes, which are ideally suited for this purpose, because they can effectively retrieve packets without resorting to expensive linear scans.

- We design and implement `pcapIndex`, an indexing scheme for packet-traces based on compressed bitmap indexes. pcapIndex extends the widely-used *libpcap* without modifying the existing API, so that *libpcap*-based applications can benefit without need for reimplementation. We show that our scheme reduces the response time by up to 3 orders of magnitude. The disk consumption is on average less than 7 MBytes per indexed field per GB of trace.

- We evaluate the performance of three state-of-the-art bitmap index compression encodings: WAH, PLWAH, and COMPAX. Our experiments on two large real-world packet traces, suggests that COMPAX, the proposed encoding, exhibits the best performance when indexing packet traces.

2. BACKGROUND

A bitmap index is a structure that accelerates search queries. It maps a sequential set of values into positions in a binary array with as many columns as the range of encountered values. Figure 1 shows an example of sequential data, having a range of 0 to 3 ($n = 4$). Values can be mapped into a bitmap index of 4 columns. For example, the existence of the second data value (3) is indicated by setting the bit to 1 in the last column of the second row. In network applications, if one wishes to map m records of port data ($n = 65535$), the required storage space would be a bitmap of $m \times n$ bits. Such a representation gives rise to efficient search operations, implemented as bitwise operations between different bitmap indexes. As an example, finding all records that used a particular source- and destination-port can be found by performing a bitwise operation between the columns of two respective bitmap indexes.

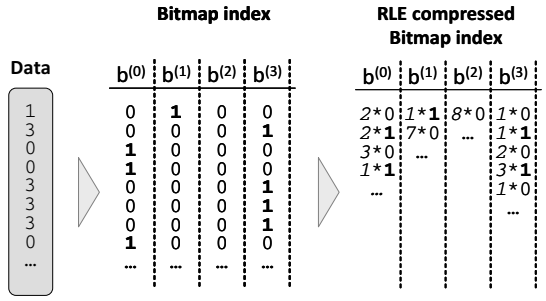

Figure 1: An example of a bitmap index (right) and the corresponding compressed bitmap index using Run-Length-Encoding(left).

A shortcoming of bitmap indexes is that they may require large storage space. This has recently given rise to *compressed bitmap indexes*. The idea is to compress bitmaps column-wise using Run-Len-Encoding (RLE). As shown in Figure 1, consecutive blocks of ones or zeros are replaced with a one or zero and a count. In this manner, disk footprint is substantially reduced. At the same time boolean operations can still be performed fast and efficiently in the compressed space. Recent works suggest that compressed bitmap indexes are expected to be more compact than typical B-Tree structures even for high-cardinality (large n) attributes [11].

WAH [12] and PLWAH [4] are current state-of-the-art compressed bitmap index encodings. In our previous work, we introduced COMPAX [7], a new compressed bitmap index encoding, and we have shown its efficiency when used to index Netflow archives. WAH, PLWAH, and COMPAX use different codeword schemes for implementing the RLE encoding. PLWAH and COMPAX use codeword schemes optimized for sparse bitmaps, whereas WAH focuses on compressing better homogeneous bitmaps. In the experimental section, we show that COMPAX provides the best compression for packet-trace datasets.

Bitmap indexing on packet-traces: The concept of compressed Bitmap Indexes (BI's) is ideally suited for packet-traces and particularly for speeding up packet filtering operations. This is for the following reasons:

i) BI's are tailored for read-only data. When dealing with packet-trace data, information is only appended but never modified.

ii) Compressed BI's are very compact in size [11] and can be memory efficient even for large packet-traces.

iii) BI's are best suited for indexing numerical attributes and the vast majority of network protocol fields are numerical.

iv) BI's do not require rebalancing (something that tree-based indexes require) and can perform fast bitwise operations even in the compressed domain.

v) Both BI's and compressed BI's can be used to answer *existence* ("Did IP W.X.Y.Z access my network?") or *cardinality* queries ("How many packets used port 445?") without access to the packet-trace [3], but using only the index structure. Cardinality queries enable one to seamlessly find and plot frequency distributions of desired attributes.

vi) Finally, compressed BI's support very fast insertion rates, which is desirable for indexing packets from high-speed links.

3. PCAPINDEX

In this section we describe the architecture of `pcapIndex`. The *pcap* packet-trace format consists of a trace header followed by chronologically-ordered tuples. Each tuple has a *pcap* header and a packet. The *pcap* header contains the timestamp when the packet was captured, the length of the packet as seen on the wire, and the portion of the packet that is stored, i.e., the capture length. Other trace formats, such as the ERF format from Endace, have a similar structure.

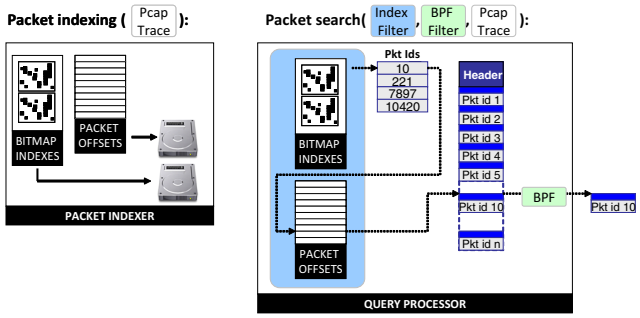

Figure 2: Packet indexing and retrieval overview.

`pcapIndex` is composed of two components: a packet indexer and a query processor as shown in Figure 2. The packet indexer receives a stream of packets from a packet-trace stored on disk or from a live capture interface. Each packet is given a sequential identifier *pkt-id*. When the end of the input stream is reached or after a specified maximum number of packets, the indexer encodes and flushes to the disk two types of files: BI's and an indirection array. The indirection array maps a *pkt-id* to the offset within the packet trace that marks the beginning of the corresponding packet. Given an input stream, the indexer constructs one file storing the indirection array and one index file for each indexed attribute. In this way, we store indexes separately without changing the *pcap* format. The indexing files are created once and are re-used from our index-enhanced pcap library for every query.

The query processor takes as input an *index filter*, a *BPF filter*, and a trace as shown in the right side of Figure 2. In particular, the index filter selects packets that match an

expression of the indexed attributes. The processor extracts from the BI's an ordered list of *pkt-id*s matching a query. The indirection array is then used to map *pkt-id*s to the corresponding packet offsets within a trace. In this way, we skip reading undesired packets from a trace file. An index filter can be chained with a BPF filter. Chaining an index with a BPF filter enables us 1) to support more complex expressions; and 2) to check less commonly queried packet attributes, e.g., ICMP codes, without having to keep an additional index for them. Packets that match the index filter are passed to the BPF filter, which sequentially checks them to produce the final result. In this way, our architecture combines the efficiency of index filters with the flexibility of BPF filters.

3.1 Packet Indexer

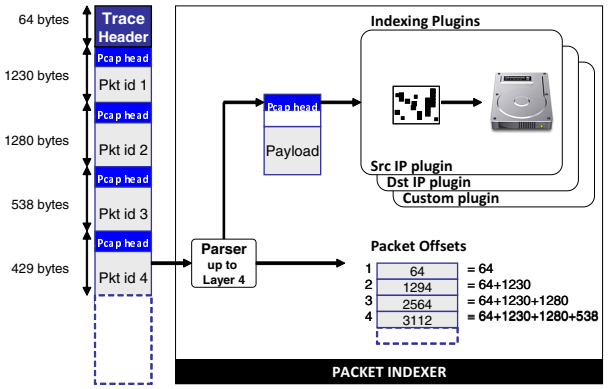

Figure 3: The packet indexer is composed of a parser and a set of indexing plugins. It decodes packets, indexes attributes, and updates the indirection array.

Figure 3 shows the architecture of the packet indexer. The packet indexer has been designed for *modularity* and *extensibility*. It is composed of a packet parser and a number of indexing plugins. Plugins index custom attributes, are enabled on request, and provide seamless extensibility.

Packet parser. The packet parser decodes headers up to the transport layer. It extracts source and destination IP addresses, MAC addresses, the VLAN identifier, source and destination ports, the layer-3 protocol and the offset from the beginning of the transport layer. The extracted information, the *pcap* header, and the packet payload are then passed on to the indexing plugins. In this way, the parsing up to transport layer is done exactly once even when multiple indexing plugins are active. In addition, the parser constructs the indirection array.

Indirection array. The indirection array is used to map *pkt-id*s to offsets. The offset of a packet is equal to the cumulative size of the trace header, the *pcap* headers, and the packets. For example, the 3rd packet in Figure 3 has an offset of 2,564. The size of the indirection array is very small, in our experiments less than 1% of the size of the trace. By encoding each offset in 64 bits we need, for example, 15.2 MBytes for the indirection array of 2 Million packets, which correspond to a trace with size between 1.1 and 1.4 GBytes in our data. The size of the indirection array can be further reduced by using more advanced encodings like gap coding, which requires much less than 64 bits for storing the difference between the offsets of two consecutive packets.

Indexing plugins. Indexing plugins receive from the packet parser the decoded header fields, the *pcap* header, and the packet payload. They optionally decode additional fields, map decoded fields to derived metrics, and then perform indexing. An indexed attribute can be 1) a decoded header field, e.g., a port number or a GTP tunnel identifier; 2) a decoded payload field for performing layer-7 packet filtering, such as filtering all request for a VoIP call to a specific address; or even 3) a derived metric, such as the country code of an IP address or the layer-7 application, e.g., HTTP.

Besides, a plugin may exploit the specifics of a field, e.g., the hierarchical nature of IP addresses, to support more complex query expressions by splitting a field into multiple indexed attributes or by synthesizing an indexed attribute by combining multiple decoded fields. For example, in our implementation we treat IP addresses in a specific way. Each byte of an IP address is indexed separately resulting in four BI's of cardinality 256. In this way, the plugins for the source and destination IP addresses allow us to provide wildcard queries (e.g. 10.10.*.*).

3.2 Query Processor

In Figure 4 we show the query workflow. A query has two optional arguments: 1) an expression, the index filter, of the indexed attributes; and 2) a BPF filter. Introducing the index filter has two important implications. First, attributes that an application is expected to query often should be indexed, e.g., a network forensics application should index IP addresses. This enables to substantially accelerate BPF-based queries involving indexed attributes, since we avoid reading unnecessary packets. Second, we can still exploit the flexibility of BPF, which enables to perform more complex queries even on fields without an index. For example, the BPF filter in Figure 4 matches HTTP GET requests. Chaining it with the index filter of Figure 4 allows the BPF filter to be applied only against the packets with source IP in the subnet 10.4/16 and destination port 80. The query processor of pcapIndex evaluates the index filter. If a BPF filter is defined, then only the matching packets are passed to the BPF engine of libpcap.

An index filter consists of one or more plugin query expressions combined with the AND (:) or OR (;) boolean operators. Parentheses can be used to enclose boolean operations (*"e1 AND (e2 OR e3)"* is allowed). Each plugin query expression has the form:

$$\langle plugin\_id \rangle = \langle plugin\_query\_string \rangle.$$

where $\langle plugin\_id \rangle$ is a unique plugin identifier that has been registered with the query processor. This enables the query processor to pass the query expression $\langle plugin\_query\_string \rangle$ to the right plugin. For example, the following string is an index filter with two plugin expressions combined with the AND operator:

"SrcPort=22:SrcIp=10.4..*"*

The query processor will split the string into plugin query expressions, will extract the query strings, and will pass them to the $\langle SrcPort \rangle$ and $\langle SrcIp \rangle$ plugins.

Each plugin computes a compressed bitmap of the positions of matching packets. For example, the $\langle SrcIp \rangle$ plugin

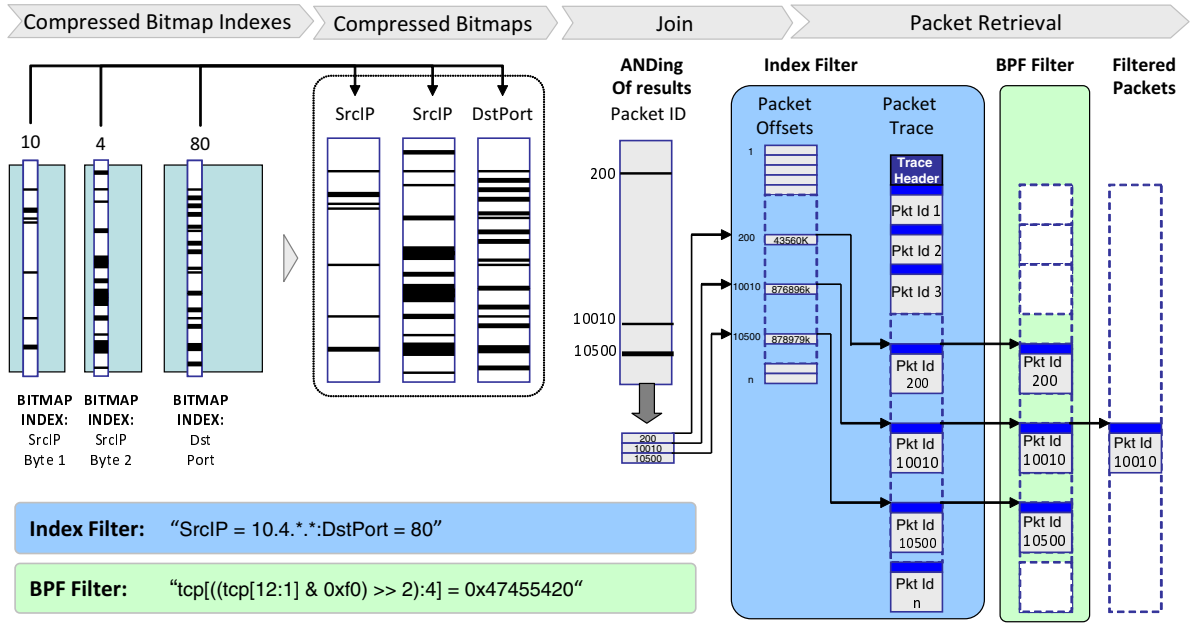

Figure 4: PcapIndex query workflow. The query processor parses the Index filter, retrieves the right BI's, joins BI's, maps *pkt-id*s to packet offsets, reads matching packets, and finally applies the BPF filter.

will retrieve the compressed bitmaps for the address bytes 10 and 4, will perform an AND operation between them, and will return a compressed bitmap encoding the result. The compressed bitmaps returned by all plugins are finally joined with the logical operators defined in the query to derive the final list of matching *pkt-id*s.

3.3 PcapIndex Implementation

PcapIndex extends the *pcap_open_offline* function of libpcap, which is the standard function for opening a packet-trace. A simple naming convention is used for specifying the index filter. In particular, the index filter is appended to the trace filename using the + character. For example, the call:

```
pcap_t = pcap_open_offline("test.pcap+SrcIp=10.10.10.10")
```

opens the pcap trace *test.pcap* and creates an array of *pkt-id*s of the packets matching the filter $SrcIp = 10.10.10.10$. In addition, the modified function memory maps the indirection array file of the trace. The name of the indirection array file is the name of the trace file appended with the ".off" suffix (in this example *test.pcap.off*). Once the array of matching *pkt-id*s is created, e.g., the packet identifiers 200, 10010, and 10500 in Figure 4, the trace traversal process can start. This is performed by calling *pcap_next_ex*, which is the standard libpcap function for tranversing a trace:

```
pcap_next_ex(pcap_t *, struct pcap_pkthdr **, u_char **)
```

We have not changed the API of this function. In the standard libpcap implementation, the function reads all the packets in a trace sequentially. With pcapIndex, the function uses the *pkt-id*s and the indirection array to skip reading unnecessary packets. Therefore, integrating pcapIndex into existing libpcap applications requires only two very simple changes: 1) including the pcapIndex header file; and 2) augmenting the filename of a packet-trace with a desired query using the + character.

In Algorithm 1 we show the pseudo-code of our implementation of the *pcap_next_ex* function. We use a counter, which is initialized to zero when a trace is opened, as a cursor in the *pkt-id* array. The counter points to the actual packet to be delivered to the caller. The function uses the indirection array file and the counter to seek, within a trace, for the pcap header of the packet to be returned. The pcap header and the packet are read from the trace. If a BPF filter is configured, the BPF filter is evaluated on the read packet. If the packet matches, it will be delivered to the user, otherwise, the counter is incremented and the process repeated.

Algorithm 1 pcap_next_ex(pcap_h)

```
Require: an open pcap handle pcap_h
 1: while has_results do
 2:    /* Seek to the packet */
 3:    pktId = pcap_h.PacketID[pcap_h.cursor];
 4:    pktOffset = pcap_h.PacketOffsets[pktId];
 5:    seek(pcap_h.tracefile, pktOffset);
       /* Read the pcap header and the packet itself */
 6:    hdr = read_pcap_header(pcap_h.tracefile);
 7:    pkt = read_packet(pcap_h.tracefile, hdr.caplen);
 8:    pcap_h.cursor +=1;
       /* Execute the BPF filter, if configured */
 9:    if (!has_bpf(pcap_h))
10:     return (hdr,pkt);
11:    else if (bpf_exec(pkt,pcap_h.bpf))
12:     return (hdr,pkt);
13: end while
14: return NULL;
```

4. EVALUATION

In this section, we evaluate critical performance metrics of the index-enhanced *libpcap* library. In particular, we evaluate 1) the disk space required to store compressed BI's, which are built using three state-of-the-art encodings (WAH,

PLWAH and COMPAX); 2) the processing overhead for constructing the indexes; and 3) the query response time for filtering the same packets using BPF or pcapIndex.

In the evaluation, we use two pcap packet traces captured in two distinct locations. The trace *trace1.pcap* has been captured at the border gateway of a university network, whereas *trace2.pcap* at the border gateway of a small ISP. The trace sizes are 1.1 Gb and 1.4 Gb, respectively. Both traces contain 2 Million packets. The traces store the entire traffic as transmitted over the network.

During the evaluation we use a commodity desktop machine with 2GB of DDR3 memory and a 2.66 GHz Intel Core2 Quad processor (Q9400) running Linux. We store traces on a 320GB desktop hard drive and also on an Intel X-25M solid-state drive (SSD).

Index sizes. Our packet indexer can be configured for using WAH, PLWAH or COMPAX. The plugins transparently use the chosen encoding. For our experiments, we enable all the plugins we have developed that capture the following attributes: IP addresses, ports, layer-3 protocol, TCP flags, packet length, packet capture length, and VLAN identifier. Table 1 reports the total size of the indexes when using the different encodings.

Table 1: Index size using WAH, PLWAH and COMPAX.

Dataset	WAH	PLWAH	COMPAX
trace1	99 MBytes	65 MBytes	55 MBytes
trace2	89 MBytes	61 MBytes	51 MBytes

COMPAX indexes are the one providing the lowest disk consumption. WAH indexes are almost twice as big, whereas the PLWAH ones are up to 15% bigger than COMPAX. This is expected as packet trace fields rarely have the same value for long sequences of consecutive packets and COMPAX can compress short sequences of repeated symbols better than WAH and PLWAH. *Indexing (with COMPAX) needs on average less than 7 MBytes per indexed field per GByte of trace.*

Processing overhead. We compare the processing overhead for building the indexes with the three encodings. We measure the time it takes to: 1) solely read a trace and 2) read and index a trace. Table 2 reports the measured time using a desktop hard drive and a solid-state drive for storing the packet traces and the corresponding indexes.

Table 2: Execution time (in milliseconds) for solely reading a trace and for reading and indexing a trace.

	Solid-State Drive (SSD)			
Dataset	**read**	**WAH**	**PLWAH**	**COMPAX**
trace1	7141	13424	13331	10583
trace2	5708	11522	11534	8768
	Hard drive			
Dataset	**read**	**WAH**	**PLWAH**	**COMPAX**
trace1	21406	23796	23069	22380
trace2	17839	19648	19350	18761

For the regular hard drive the indexing time is almost the same as the reading time: *IO operations for reading packets are the bottleneck of the indexing time.* In fact, during our experiments we noticed that the CPU utilization was low and the IO wait percentage high. Even with the faster solid-state drive, the IO is the main bottleneck. Based on this observation, *we highlight that because of the IO bottleneck, indexing has a great potential to reduce query response time as it circumvents reading unnecessary packets.* In addition, we find that *COMPAX has the lowest processing overhead for constructing an index.* We attribute this to the lower disk utilization of COMPAX. Based on this, we use COMPAX as the default encoding for the rest of the experiments.

Query response time. We compare the query response time of BPF and pcapIndex using queries of different selectivity. We choose ports and IP addresses, which are the most commonly-used attributes for searching packets, as query attributes. To create queries of varying selectivity, we first find all distinct source IP addresses and sort them by number of occurences. We query for the 200 most frequent (top 200) and 200 least frequent (bottom 200) IP addresses. In this way, we capture the two ends of the query selectivity spectrum. Using the same methodology, we create 400 additional queries for the source port attribute. Both traces have more than 50 thousand distinct IP addresses and more than 40 thousand distinct ports.

The queries are executed with *pcapidx-query*, a dummy *libpcap* application that reads the packets matching a BPF or index filter and discard them without doing any further processing. For each query, we measured the *pcapidx-query* execution time. In addition to reading packets, the measured time includes the time to compile a BPF filter or the time to retrieve the *pkt-id*s array from the index. Since packet traces can be cached, we unmounted the filesystem every time we ran *pcapidx-query*, which ensures that a trace is not cached. To better understand the performance implications of the disk type, we repeated our experiments using a desktop hard drive and a solid-state drive.

In Table 3 and 4 we report the average running time for each block of 200 queries when traces are stored on the solid-state drive and on the hard drive, respectively.

Table 3: Query response time (in milliseconds) with pcapIndex and BPF on a solid-state drive. The grey cells mark the speedup of pcapIndex.

	Source ports					
	Top 200			Bottom 200		
Dataset	**BPF**	**index**		**BPF**	**index**	
trace1	7076	654	10.8×	7101	6	1183.5×
trace2	5714	983	5.8×	5746	47	122.3×
	Source IP addresses					
	Top 200			**Bottom 200**		
Dataset	**BPF**	**index**		**BPF**	**index**	
trace1	7113	799	8.9×	7125	12	593.8×
trace2	5733	1148	4.9×	5724	8	715.5×

Table 4: Query response time (in milliseconds) with pcapIndex and BPF on a regular hard drive. The grey cells mark the speedup of pcapIndex.

	Source ports					
	Top 200			Bottom 200		
Dataset	**BPF**	**index**		**BPF**	**index**	
trace1	21555	8925	2.4×	21532	28	769.0×
trace2	17886	9142	1.9×	17870	128	139.6×
	Source IP addresses					
	Top 200			**Bottom 200**		
Dataset	**BPF**	**index**		**BPF**	**index**	
trace1	21590	11393	1.9×	21522	50	430.4×
trace2	17849	8764	2.0×	17843	53	336.7×

We first observe that the average execution time of BPF is independent of the query selectivity. This is because without an index a trace is fully scanned for each query. In fact, the

average execution time is very close to the time reported in Table 2 for reading all packets. *Disk IO is therefore the main factor that determines the BPF response time.*

For low selectivity queries (bottom 200), using an index improves the response time by 2-3 orders of magnitude for both types of drives. The impact of the disk type is larger for high selectivity queries (top 200). Even for these queries, the index reduces response time: the speedup is up to 10 times on the solid-state drive and close to 2 on the standard hard drive. This is because indexing profits from the low seek time offered by solid-state drives. This highlights that *in newer disk technologies providing lower seek times the query speedup offered by pcapIndex is larger*, which indicates that in the future pcapIndex has the potential to deliver even higher speedups.

We repeat the same experiments without unmounting the disk to prevent caching before running each query. We observe that traces are fully cached and that indexing always provides better query response time than BPF even for high selectivity queries. This is expected because when a trace is cached seek operations are almost for free, whereas BPF still needs to access and apply filters to every packet.

5. RELATED WORK

Long term historical analysis of massive volumes of network management data is an emerging requirement for increasing reliability, security and performance of modern networks [8]. The Time Machine [9] and Hyperion [6], which are packet archival solutions designed for high-speed links, have identified indexing as a mandatory feature for exploring massive packet repositories. Both architectures save packets in large file segments. Indexes are used for performing existence queries, i.e., for checking the presence of packets satisfying certain conditions in a segment. Since the used indexes do not provide the positions within the segment where the desired packets are stored, the matching segments have to be linearly scanned. In this work, we show that compressed bitmap indexes allow the actual positions of packets of interest within a segment to be retrieved, pose limited disk requirements, can be created efficiently.

Special purpose columnar-databases with built-in support for compressed bitmap indexes have been used for storing and indexing NetFlow records [5, 7]. These solutions do not have strict legacy compatibility requirements, whereas in our case the goal is to provide indexed access to already existing packet-trace repositories without requiring an impractical re-encoding of entire packet repositories, and more importantly, without requiring the already existing commercial or in-house packet analysis applications to be reimplemented.

Despite the growing interest in packet-trace indexing, packet trace analysis libraries, such as *libpcap*, do not provide any indexing functionality. In particular, the BPF packet filtering mechanism [10], which was originally designed for filtering packets from streams, has been used for searching packet-traces. In this work, we show that a widely-used packet-trace analysis library can be extended to support index-based searching with legacy compatibility.

6. CONCLUSIONS

In this work we have built upon the popular *libpcap* packet-capture and trace-analysis library and extended it to support fast filtering using compressed bitmap indexes. We have designed `pcapIndex`, the first indexing scheme for packet-traces based on compressed bitmap indexes, and have made it backward compatible with *libpcap*. Our work allows the rich pool of trace analysis applications implemented on top of *libpcap* to benefit from indexing, with no need for reimplementation or for re-encoding existing pcap-based repositories. Our evaluation suggests that indexing packet traces imposes minimal disk overheads and provides impressive speedups, particularly when few packets have to be retrieved from large packet-traces. We compare the performance of three state-of-the-art compressed bitmap encodings on packet-traces and show that COMPAX exhibits the best performance. Finally, we demonstrate that our index-based *libpcap* significantly benefits from modern solid-state hard drive technologies, which highlights that in the future `pcapIndex` has the potential to deliver even better performance.

7. REFERENCES

[1] Endace Measurement Systems. http://www.endace.com.

[2] TCPDUMP/LIBPCAP public repository. http://www.tcpdump.org/.

[3] E. W. Bethel, S. Campbell, E. Dart, K. Stockinger, and K. Wu. Accelerating Network Traffic Analysis Using Query-Driven Visualization. In *Proc. of 2006 IEEE Symposium on Visual Analytics Science and Technology*, pages 115–122, 2006.

[4] F. Deliège and T. B. Pedersen. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *Proc. of the 13th Int. Conf. on Extending Database Technology*, pages 228–239, 2010.

[5] L. Deri, V. Lorenzetti, and S. Mortimer. Collection and Exploration of Large Data Monitoring Sets Using Bitmap Databases. In *2nd Int. Workshop on Traffic Monitoring and Analysis(TMA)*, pages 73–86, 2010.

[6] P. J. Desnoyers and P. Shenoy. Hyperion: high volume stream archival for retrospective querying. In *Proc. of the USENIX Annual Technical Conf.*, 2007.

[7] F. Fusco, M. P. Stoecklin, and M. Vlachos. Net-fli: on-the-fly compression, archiving and indexing of streaming network traffic. *Proc. VLDB Endow.*, 3, 2010.

[8] C. R. Kalmanek et al. Darkstar: Using exploratory data mining to raise the bar on network reliability and performance. In *7th Int. Workshop on Design of Reliable Communication Networks*, 2009.

[9] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider. Enriching network security analysis with time travel. In *Proc. of the ACM SIGCOMM 2008 Conf. on Data communication*, pages 183–194, 2008.

[10] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *Proc. of the USENIX Winter Conf.*, pages 2–2, 1993.

[11] K. Wu, E. Otoo, and A. Shoshani. On the Performance of Bitmap Indices for High Cardinality Attributes. In *Proc. of the 13th Int. Conf. on Very Large Data Dases (VLDB)*, pages 24–35, 2004.

[12] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31:1–38, March 2006.