NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic

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

Marc Ph. Stoecklin
IBM Research - Zurich
Ecole Polytechnique
Fédérale de Lausanne (EPFL)
mtc@zurich.ibm.com

Michail Vlachos
IBM Research - Zurich

ABSTRACT

The ever-increasing number of intrusions in public and commercial networks has created the need for high-speed archival solutions that continuously store streaming network data to enable forensic analysis and auditing. However, "turning back the clock" for post-attack analyses is not a trivial task. The first major challenge is that the solution has to sustain data archiving under extremely high-speed insertion rates. Moreover, the archives created need to be stored in a format that is compressed but still amenable to indexing. The above requirements make general-purpose databases unsuitable for this task, and, thus, dedicated solutions are required.

In this paper, we describe a prototype solution that satisfies all requirements for high-speed archival storage, indexing and data querying on network flow information. The superior performance of our approach is attributed to the on-the-fly compression and indexing scheme, which is based on compressed bitmap principles. Typical commercial solutions can currently process 20,000-60,000 flows per second. An evaluation of our prototype implementation on current commodity hardware using real-world traffic traces shows its ability to sustain insertion rates ranging from 500,000 to more than 1 million records per second. The system offers interactive query response times that enable administrators to perform complex analysis tasks on-the-fly. Our technique is directly amenable to parallel execution, allowing its application in domains that are challenged by large volumes of historical measurement data, such as network auditing, traffic behavior analysis and large-scale data visualization in service provider networks.

1. INTRODUCTION

Corporate and service provider networks, financial institutions and high-security data centers are increasingly interested in tools that allow them to archive network traffic information for post-mortem analysis. Imagine for example, the case of an identified data breach at a financial institution where the system administrator would like to quickly pinpoint the accessed nodes from the list of suspected IP addresses to isolate any additional compromised nodes. A similar scenario is encountered during the outbreak of a computer worm (cf. Fig. 1), in which all network nodes being contacted by the infected host need to be discovered to understand the propagation pattern of the worm.

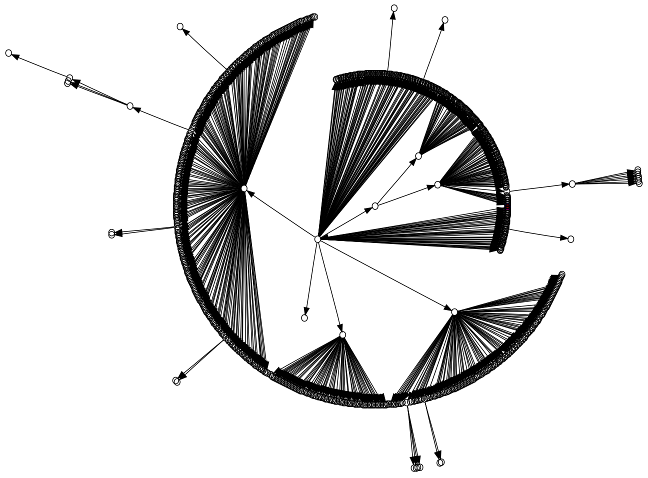


Figure 1: Propagation graph of a worm epidemic used to identify potentially compromised nodes in a network.

To address the above cases, all inbound and outbound traffic can be recorded in order to recreate the original breach conditions. To limit disk consumption, most companies typically only record historical *network flow* data. In that way one can still capture information, such as source and destination IP addresses, ports, protocols and time, but avoid recording the actual packet content, something that would result in prohibitive repository sizes and also severely compromise user privacy. However, even with a flow-based approach, huge repositories will be accumulated over time. Currently, a typical service provider network may exhibit flow export rates as high as 50,000 flows per second; this would amount to more than 8 GB of raw flow information per hour. Therefore, even with data compression enabled, the resulting data repository could easily reach the order of terabytes on a yearly basis.

General-purpose databases are not able to scale to the required high insertion rates. Even sifting through such enormous databases is not trivial, particularly when interested in identifying "needles in the haystack". To effectively address the above issues, two mechanisms need to be in place: i) a low-latency data storage mechanism that will capture and archive all streaming network information of interest, and ii) a search mechanism over the archival data, potentially with the help of an indexing scheme.

This work introduces **NET-FLi** (NETwork FLow Index), a highly optimized solution for real-time indexing and data retrieval in large-scale network flow repositories. By exploiting the nature of network flow data, we introduce adaptive indexing and data compression mechanisms which adapt the principles of bitmap indexing and Locality Sensitive Hashing (LSH). Our methodology offers real-time record processing, with high compression rates and interactive query response times. Both data compression and indexing are performed *on-the-fly*. The low response time for data retrieval from the repository is attributed to our indexing and selective data-block decompression strategies.

Our solution can be utilized in the following applications:

1) **Network forensics**, for auditing and compliance purposes. Compressed network flow archives capture digital evidence that can be used for retrospective analysis in case of information abuse and cyber-security attacks. We illustrate such an example in Fig. 1, which explores the past connectivity graph of a potentially compromised node. Such an analysis would require extended recursive exploration of the inbound/outbound traffic through access of the stored archive. Using traditional approaches such a task would require hours to complete; using our framework the task can be completed in only a few seconds.

2) **Network troubleshooting**, as an indispensable tool for system administrators and data analysts to better support network management. Consider the case of an operator requiring a visualization of the dependencies between a set of servers; this is a necessary step for preparing a server migration plan and requires extensive retrieval of historical traffic usage and access patterns in one's domain. The end-user therefore requires fast *drill-down* capabilities for querying the archive system. In addition, the fast search functionalities over the archival data as provided by our solution can assist in expedient resolution of network anomalies, identification of network performance issues and bottlenecks, and lead to better load balancing of the network.

3) **Behavior analysis**, with focus on traffic classification. It is of general interest to identify the application that generated a particular network communication. This is a challenging problem because nowadays a huge amount of network traffic is transported through widely used ports, such as port 80 (e.g., Skype or torrent file transfer) [19]. Accurate application-deciphering rules can potentially be distilled by examining the cardinality of flows, packets and bytes exchanged, as recorded in the network flow records.

4) **Streaming data indexing**. Even though the solution presented has been created with network data in mind, the indexing and compression strategies that we present can also be used for archival and search over any streaming numerical data, when high efficiency and low response time are of essence.

Our work makes several important **contributions**:

- We present a unique on-the-fly solution for archiving and indexing of network flow data based on the synergy of an alphabet-optimized bitmap index variant, along with an online LSH-based reordering scheme to boost the space savings of our approach even more.

- The compressed columnar approach that we propose presents compression ratios on par with that of a typical gzip compression, with the added benefit of providing indexing functionality, and partial and selective archival-block decompression.

- Typical data insertion rates of our approach can reach **1.2 million flow records/sec** on a commodity desktop PC. High-bandwidth networks currently experience bursts of up to 50,000 flows/sec traffic, so our approach offers an order of magnitude higher processing rates than these required to capture all flows from a typical network device.

- The data index achieves **interactive retrieval time** while searching through gigabytes of compressed flow repositories.

- The architecture actively exploits the parallelism offered by multi-core and multi-processor systems.

In the remainder of the paper, we go into more detail on the distinct advantages of our solution. We begin by surveying related work and elementary concepts in Sections 2 and 3. Section 4 presents the overall architecture of our solution. We evaluate our approach in Section 5, provide a comprehensive case study in Section 6 and discuss limitations and future work in Section 7.

2. RELATED WORK

Network forensics involves topics that need to be efficiently addressed from both the network and the database perspective. Below we review some solutions relevant to the task at hand and, when possible, highlight the differences to our approach:

a) *Network Traffic Recording Systems* [2, 5] are deployed by financial and security agencies interested in keeping a sliding window of the traffic, which can be replayed entirely for post-mortem analysis. Filters for supporting on-the-fly queries can be enabled , but interactive queries over the archived data are not supported.

b) *Data Stream Management Systems* (DSMS), such as Gigascope [12], TelegraphCQ [10], and System S [6, 34] have been introduced to perform online queries over data streams. The idea behind the stream database approaches is to support static queries over online data without storing the entire stream on disk. Only the query results (i.e., data aggregations) are kept on secondary storage. Those systems usually provide SQL-like languages augmented with stream-aware operators. Plagemann et al. provide examples of stream databases deployed in the context of traffic analysis [28].

c) *Flow-Based Systems* attempt to lift some of the limitations of stream-based systems by archiving entire network streams. Silk [16], nfdump [18] and flowtools [30] are the most commonly used tools for storing flows. They all store flow records as flat files, with optional data compression using tools such as gzip, bzip2 and lzop [25]. Important shortcomings of those software suites are that: i) they do not provide any indexing facility, and ii) the entire data set must be scanned linearly (and decompressed) even for retrieving only a few records. In contrast, our flow storage solution has been designed for minimizing the amount of data to be decompressed at query time.

Reiss et al. [29] propose a flow storage solution implemented on top of Fastbit [3], a column-oriented library with built-in support for compressed indexes when performing queries over historical data. Contrary to NET-FLi, their flow storage solution does not provide any compression mechanism for the data, which is appended uncompressed (and unsorted) to the disk. In addition, indexes are built offline and in batch. NET-FLi can handle twice the incoming flow rate with compression rates equivalent to that of gzip, leading to 40% smaller indexes.

Distributed architectures for flow storage have also appeared in MIND [22] and DipStorage [24], which distribute flow records among different peers to decrease the response time of multi-dimensional queries. Our work is orthogonal, as our flow storage solution can be used as a back-end system for distributed storage architectures.

d) *Flow Aggregation Databases* such as as AURORA [1] and nTop [4] use stream-based approaches to analyze high-speed networks and provide statistics (e.g., IP addresses with highest usage) at different time scales (i.e., hourly, daily, weekly, or yearly). However, unlike stream databases which are programmable with SQL-like languages, network aggregation databases provide a predefined set of analysis tasks, trading flexibility for performance. In addition, they do not offer drill-down capabilities, which are essential for our application.

In our solution, we adopt a columnar data representation. Related to our work are therefore column-oriented database systems such as CStore [31], MonetDB [9] or BigTable [11]. Organizing the data in columns rather than in rows offers several advantages: i) it provides more opportunities for compression, ii) it allows operators to be implemented directly over the compressed data, and iii) it reduces the I/O bandwidth for queries with high selectivity on the attributes.

In NET-FLi we introduce a specialized compressed bitmap index variant, which we call COMPAX. Existing bitmap indexes, such as Word-Aligned-Hybrid (WAH) [33] or Byte-Aligned Bitmap Codes (BBC) [8], have been shown to offer indexes that are many times smaller than traditional tree-based indexing data structures [32]. Our evaluations suggest that COMPAX is superior to the state-of-the-art index WAH in terms of compression rate, indexing throughput, and retrieval time.

3. BACKGROUND

Before we start describing our solution in detail we briefly revisit the concept of network flow, which is essential for our framework.

The *network flow* structure has been introduced in the area of network monitoring to capture a variety of information related to network traffic. A flow is defined as a unidirectional sequence of network packets sharing the following 7-tuple key of equal values: source and destination IP address, source and destination port, transport proto-

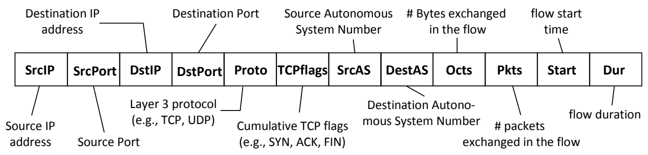

Destination IP address
Destination Port
Source Autonomous System Number
# Bytes exchanged in the flow
flow start time
SrcIP
SrcPort
DstIP
DstPort
Proto
TCPflags
SrcAS
DestAS
Octs
Pkts
Start
Dur
Source IP address
Source Port
Layer 3 protocol (e.g., TCP, UDP)
Cumulative TCP flags (e.g., SYN, ACK, FIN)
Destination Autonomous System Number
# packets exchanged in the flow
flow duration

Figure 2: Attributes that can be present in a flow entry record

col, ingress interface, and service type. Network equipment, such as routers, is equipped with monitors (referred to as flow meters) that maintain statistics of the flows observed, such as the number of bytes, packets, or TCP flags. Once a flow is considered terminated, the statistics are exported as a flow record to a collector device; flow records consist of a pre-defined data structure used to represent the fixed-size attribute fields. Over the years, different export protocols (e.g., Netflow v5, Netflow v9, IPFIX) have been proposed. Netflow v5, the most widely used protocol, uses 48 Bytes (including three padding bytes) to encode 19 flow attributes of a single flow. In Figure 2 we depict the network flow attributes that we utilize in our setting, along with a simple description of the fields.

By archiving collected flow records over time, a large repository documenting all end-to-end network communication patterns can be created. This repository provides a valuable source of information for many analysis tasks conducted on a daily basis by administrators, such as the examination of anomalous traffic patterns, forensic investigation, usage accounting, or traffic engineering.

4. ARCHITECTURE

In this section, we describe our approach, discuss its inner workings, and explain our design choices. The storage solution comprises two logical components, which are depicted in Fig. 3:

1. The *archiving backend*, which compresses incoming flow data, and

2. The *compressed index*, which encodes the flow information using the COMPressed Adaptive indeX format (or COMPAX), using a pre-specified codebook of words.

The tasks of the two components are executed on-the-fly and in parallel, which is the reasons for the high-performance of our solution. An additional online pre-processing step can be executed, that reorders the flows using a variation of a Locality Sensitive Hashing technique, with the purpose of achieving better compression in both the archived data and the data index. This approximate sorting approach is computationally lightweight, allowing an on-the-fly execution over the streaming network data. Although this optional process reduces the flow processing rate, it results in more compact archives and indexes, and eventually leads to faster response times of the overall system.

The last component in the system is the *query processor* which, given a search query and using the bitmap index structures created, identifies historical flows matching the criteria given and retrieves them by uncompressing only the relevant portions of the archived data. Below we provide more details on each of those components.

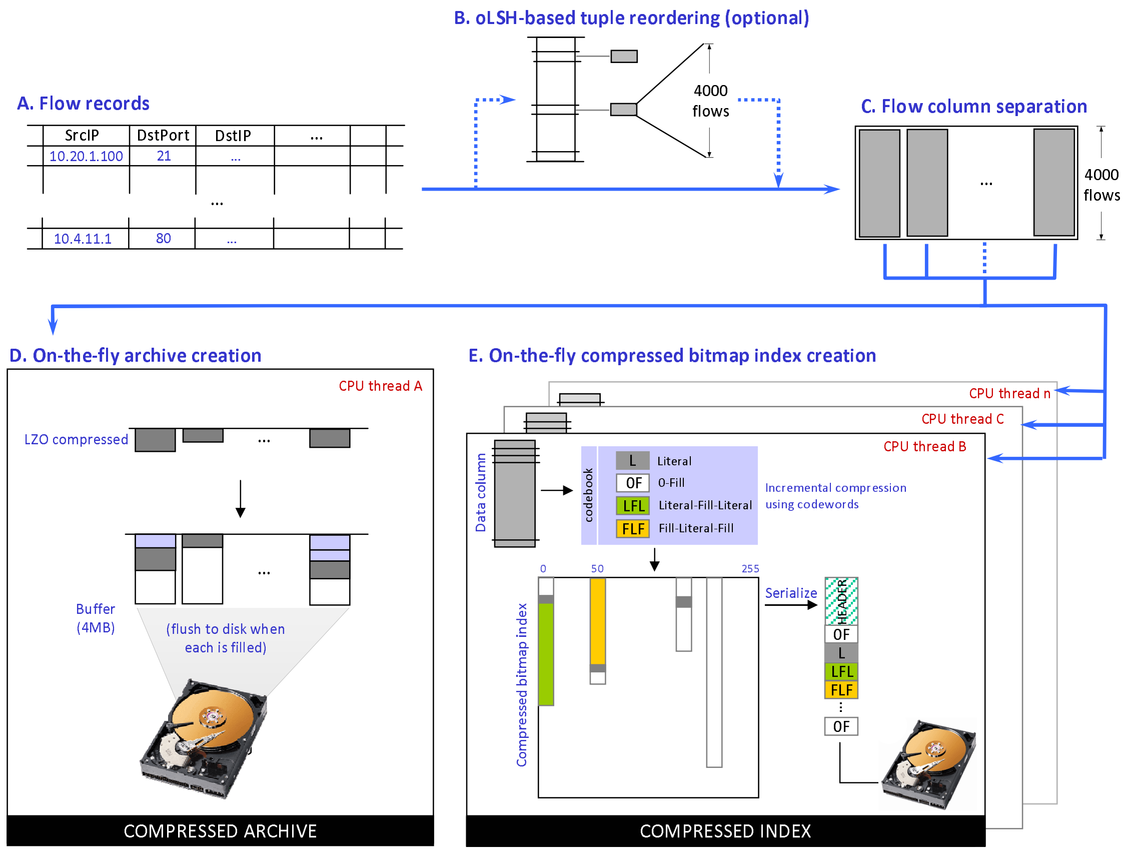


Figure 3: The flow archival process consists in two parallel tasks: creation of compressed columnar archive (D) and on-the-fly construction of compressed bitmap indexes (E).

4.1 Compressed Archive of Flow Data

The input to our system consists of streaming network flow records, which are exported by flow meters. As mentioned, flow records can record a number of predefined communication attributes; in our system we record the 12 attributes listed in Fig. 2.

Incoming flow records are packetized and processed using a tumbling data window of 4000 flows (Fig. 3.B). The size of the window was not selected arbitrarily, but reflects the amount of processed data (12 attributes $\times$ 4000 records) that fit into the L2 cache of the system. We also experimented with larger flow record block sizes, but did not observe that it significantly affected the compression rate. In addition, keeping packetization level low by using 4000 records provides finer-grained access on the archived data and promotes decompression of fewer archive blocks when querying the data.

For all subsequent phases, the data are treated in a columnar form, and each attribute (column) of the flow records is processed independently (Fig. 3.C). Each of these conceptual data columns of length 4000 is compressed using a user-defined compression scheme. Ideally, algorithms that can support fast compression and even faster decompression are preferred. For this reason in our prototype implementation we chose the Lempel-Ziv-Oberhumer (LZO) compressor [25], which corresponds to one the fastest algorithms available, particularly for decompression[1].

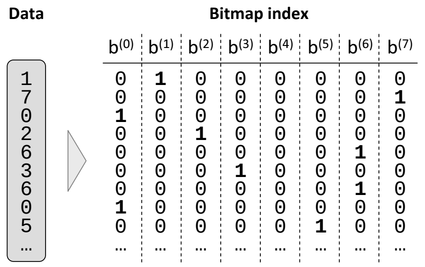
The compressed columnar data archives are created on-the-fly and stored on disk. To reduce the number of random I/Os, the column archives are not written to disk as soon as they are created: the compressed blocks are initially buffered and only flushed to disk when the buffer is full. In our implementation, we allocate 4 MB for each column buffer (Fig. 3.D).

The task described in this section, consisting of packetization, reordering, and compression, is executed within one CPU core of the system. Optionally, the compression of the different data columns can be distributed in different CPU's or cores, when heavy-weight compression algorithms are utilized.

4.2 COMPressed Adaptive indeX - COMPAX

Concurrently with the creation of the compressed flow data archive, a compressed bitmap index is constructed, which facilitates the speedy location of relevant portions in the archived data during the querying process. We call the new index *COMPressed Adaptive indeX* or *COMPAX*. It is constructed using a codebook of words that significantly reduces the bitmap index size. The entire process is performed online. We begin by elucidating the creation and usefulness of traditional bitmap indexes.

[1]LZO is four to five times faster in decompression than the fastest zlib compression level [26].



Data	$b^{(0)}$	$b^{(1)}$	$b^{(2)}$	$b^{(3)}$	$b^{(4)}$	$b^{(5)}$	$b^{(6)}$	$b^{(7)}$
1	0	1	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0
6	0	0	0	0	0	0	1	0
3	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0
…	…	…	…	…	…	…	…	…

Data
Bitmap index

Figure 4: An example of a bitmap index.

4.2.1 *Bitmap Indexes*

The concept of *bitmap indexing* has been applied with great success in the areas of databases [3] and information retrieval [15]. The basic notion is to keep k bitmaps (columns), one for every possible value that an attribute can assume (k refers to the attribute cardinality), and to update them at every insertion by appending a "1" to the bitmap corresponding to the inserted value and "0" otherwise. An example of a bitmap index with $k = 8$ is shown in Fig. 4. In addition to fast updates, bitmap indexes allow inexpensive bitwise AND/OR operations between columns. *Compressed* variants of bitmap indexes have appeared in the literature [32, 14], with the most popular variant being the World-Aligned-Hybrid index (WAH) [33], which has been used for indexing and searching on a multitude of datasets including scientific simulations results and network flow repositories.

4.2.2 *COMPAX encoding*

We compress each bitmap index in a column manner *on-the-fly* using a codebook[2] of four word types. The following four 32-bit wide word types are used to encode the incoming bit stream processed in *chunks* of 31 bits:

1. A literal **[L]** represents a chunk consisting of a bitwise mix of zeros and ones; it is encoded as a 32-bit word identifying the codeword type with the first bit (`1`), for example:

   ```
   1 0011100 11000010 00110000 00000000
   ```

2. A 0-Fill **[0F]** encodes a sequence of consecutive chunks of zero bits by means of run-length encoding.

   For example, a portion of a bitmap index column consisting of a sequence of 3×31 zero-bits is encoded within a single 32-bit word as:

   ```
   000 00000 00000000 00000000 00000011
   ```

   where the first three bits (`000`) encode the codeword type and the remaining 29 payload bits encode the number of 31-bit-chunks from the original sequence.

3. An **[LFL]** word represents a sequence of [L]-[0F]-[L] words after applying null-suppression. In particular, if in each of the three words in the sequence only one of the payload bytes is non-zero ("dirty") and the dirty byte in the [0F] word is at position 0,[3] the three dirty bytes and the two positions (0 to 3) from the [L] words can be packed into a single [LFL] word. An example of how the [LFL] codeword is formed is shown in Fig. 5.

4. The **[FLF]** word represents a sequence of words that follow a [0F]-[L]-[0F] paradigm. Similarly, to the [LFL] codeword, when three consecutive words exhibit the [0F]-[L]-[0F] pattern, with only one dirty byte in each payload and the dirty bytes of the [0F] words are both at position 0, then they can be condensed into an [FLF] codeword while retaining the position of the dirty byte in the [L] word payload.

[2]We borrow the term "codebook" from information theory, which we feel is appropriate in this context, because the encoded words help us compress the incoming information.

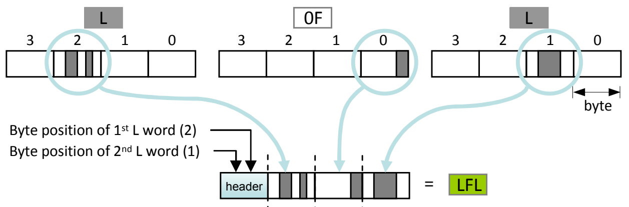

L
0F
L
3
2
1
0
byte
Byte position of 1st L word (2)
Byte position of 2nd L word (1)
header
=
LFL

Figure 5: Example of the creation of a [LFL] codeword from a [L]-[F]-[L] sequence.

The bit encoding for all four codewords in shown at the bottom right-hand side of Fig. 6.

Difference to WAH: The WAH encoding utilizes two word types to encode bitmaps: 0-Fill and 1-Fill. In our work, we introduce the [LFL] and [FLF] codewords because we noticed that such patterns were very predominant in network flow traffic. In addition, we omit the [1-Fill] from the codebook because contiguous bit blocks of 1's are quite uncommon in our setting.

The practical implications of our coding scheme is that it results in significant space savings. In the experimental section, we show that, compared to WAH, COMPAX encoding can result in space reduction of more than 60%, particularly when combined with the tuple reordering phase. For example, an 8.1 GB WAH-compressed bitmap index is reduced to 3.3 GB when compressed with COMPAX.

Figure 6 depicts the difference of the COMPAX encoding to WAH. The bits of a column are shown row-wise for presentation reasons, and the original uncompressed sequence comprises of 155 bits (indicated as 5 verbatim 31-bit chunks **[V]**). In this example we illustrate COMPAX' ability to condense three WAH words into one. COMPAX packs more information because of the carefully selected codebook and as such offers superior compression. In the same figure, we also depict how the different words are encoded and the meaning of the various bits.

On-the-fly Bitmap Index creation: The COMPAX-encoded bitmap indexes can be constructed on-the-fly in an incremental fashion to reduce memory consumption. Note that the construction is performed in a single pass on the observed data and requires a look-back of at most two words. We briefly elucidate the online incremental creation of the bitmap by an example.

Assume that a 0-Fill codeword has been created in a certain column, encoding 3×31 bits of zeros (length 3), as

[3]The fill length of the [0F] word must be less than 256 31-bit chunks.

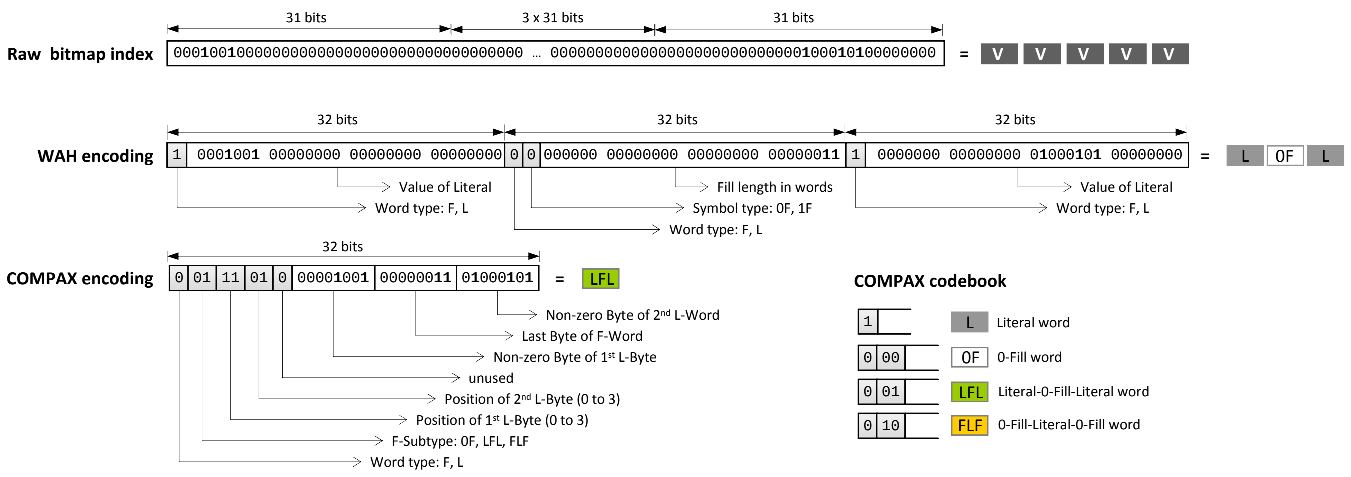


Figure 6: Example encoding of a raw bitmap vector (top) using the WAH (middle) and the COMPAX method (bottom). The COMPAX codebook is shown at the bottom right.

shown in the example for [0F] above. Now, suppose that for the same column an additional sequence of 31 zeros is observed. This will result in the creation of an additional 0-Fill word of length 1. By examining the previously produced codeword, we are able to merge them and produce a single 0-Fill word encoding 4×31 zeros (length 4). In binary format this is encoded as:

```
000 00000 00000000 00000000 00000100
```

In a similar manner, we maintain the codewords per column on-the-fly. In addition to 0-Fill words, we distinguish the following cases:

- [L]: Unlike the case for 0-Fill words, when a new 31-bit *literal* chunk is observed, it cannot be merged with a previously produced codeword, so another literal [L] word is created.
- [LFL]: To form an [LFL] codeword, a lookback examines the two previously produced codewords. If they are [L] and [F] and the current word is a literal—and all three codewords have only one dirty byte (and the dirty byte in [F] is at position 0)—then these three codewords are merged into a single [LFL] word.
- [FLF]: Similarly treated as [LFL].

The advantage of this incremental encoding is the significant reduction in memory requirements; at no point does the entire bitmap have to be materialized but it is always stored and updated in a compressed format. An illustration of this incremental bitmap creation is shown in Fig. 3.E.

Bitmap Index serialization: The COMPAX compressed bitmap indexes are serialized to disk by appending the compressed columns sequentially. Every index is prepended with a header that contains pointers to the beginning of each compressed column. In such a way, random access to specific compressed column within a bitmap index is accommodated. To preserve disk space, the header is also compressed, with the Simple9 algorithm [7], which represents one of the fastest integer coding techniques, particularly during decompression.

In our implementation, we create bitmap indexes for the most commonly queried attributes, such as *source and destination IP addresses*, *source and destination ports*, *protocol*, *tcpflags*, *duration* and *start time*. Particular consideration is taken for the IP fields: a separate index is maintained for each 8-bit block of a 32-bit IP address. In this manner one can accelerate wild-card queries over networks (e.g., 10.1.*.*) by combining compressed bitmaps belonging to different byte indexes using boolean AND operations.

4.3 Querying the System

The proposed architecture can answer very efficiently common types of queries that are posed by system administrators and security analysts. The system users can utilize both equality or range queries on attributes contained in the index, such as source and destination IP addresses, ports, protocols, and the time-span of interest. The query execution consists of the following steps (see Fig. 7):

1. Columns/attributes involved in the query are determined. Relevant columns from the appropriate compressed bitmap indexes are retrieved.
2. Boolean operations among compressed columns are performed directly *without explicit decompression*. Flow record positions on the compressed archive are resolved.
3. The appropriate portions of the archive (relevant compressed 4K blocks) are decompressed, and the results are furnished to the user.

We will explain the above process with an example. Consider the case of an worm attack that occurred the previous evening. The system administrator would like to find all destination IP addresses contacted on port 137 by nodes in the range 10.4.0.0/16. The corresponding query is:

> **Query:** "Find all destination IP addresses contacted by the source IP address range 10.4.*.* at destination port 137."

The various substeps are depicted in Fig. 7. As bitmap index files are created on an hourly basis, first the bitmap

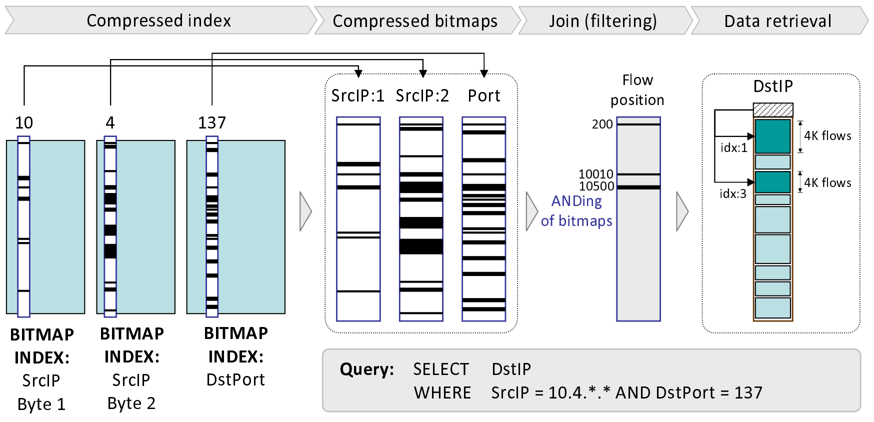


Figure 7: Example of a query execution.

index within the time range of interest in the query is retrieved. The bitmap indices for *SrcIP.byte1*, *SrcIP.byte2* and *DstPort* are retrieved, and from those the relevant *compressed* columns 10, 4, and 137, respectively, are fetched. Note that we do not need to uncompress the columns. Subsequently, an AND operation is performed on the three compressed columns. This represents an inexpensive operation that can be done directly in the *compressed domain* as one does not need to AND the portions containing the 0-Fill codewords.

Suppose the join result of the three compressed columns indicates that there are matches at flow row numbers of $\{200, 10010, 10500\}$. Because the user wants to retrieve the destination IP addresses, the query processor will need to uncompress the relevant blocks in the archive of column *DstIP*. Recall that each of the compressed blocks on the archive contains 4000 flow records. Therefore, to access the 200th, 10010th and 10500th flow record, we need to retrieve only the first and third compressed blocks in the archive. The start position of those blocks is provided in the header of the archive. Finally, the result set of the three destination IP addresses, contacted by the specified range of source IPs in the query, is returned to the user.

4.4 Online-LSH (oLSH) Stream Reordering

We also introduce an online stream reordering mechanism, which is based on the principle of Locality Sensitive Hashing (LSH) [17, 23]. We call this variant online-LSH or oLSH for simplicity. It implements an intelligent buffering mechanism with the purpose of packing together "similar" records from a data stream. It has been shown that data sorting leads to smaller and faster bitmap indexes [27, 21]. Fastbit [3], the reference implementation of the WAH algorithm, accommodates an optional off-line column sorting to decrease index sizes. In this work, we rely on sorting to decrease the disk consumption of both indexes *and* columns.

The general characteristics and benefits of oLSH are:

- It reorders the incoming data records in a fast and effective way, resulting in data blocks with lower entropy on average.
- It improves the compression rate leading to smaller bitmap index and archive sizes.
- By placing similar records in close-by positions it eventually leads to faster response times of the system, because fewer data blocks need to be decompressed from the data archive.

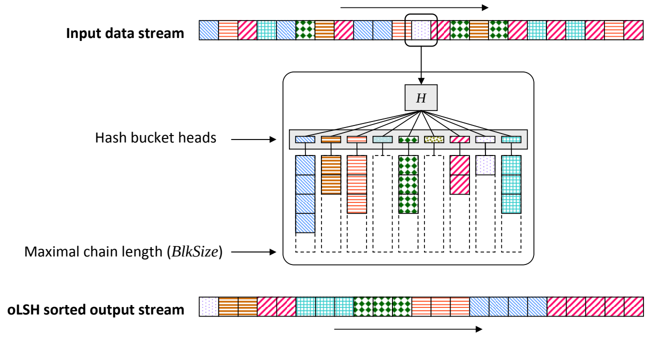


Figure 8: Mechanism of the approximate on-the-fly LSH (oLSH) sorting.

The stream reordering is implemented by a hash-based buffer, which, by using several LSH-based functions, groups flow records by content. In fact, the basic premise of LSH is to use hash functions that cause vectors which are close according to a distance function to collide with high probability to the same hash bucket. The sorted output stream is then built by constantly removing similar records from the hash (cf. Fig. 8).

We consider each flow record as a vector $\vec{r} \in \mathbf{N}^d$, where d is the number of the attributes relevant to the sorting process. The purpose of the LSH functions is to aggregate "similar" flows to the same hash bucket. Each hash bucket eventually contains a chain of "similar" records. We employ n LSH functions based on p-stable distributions, such as the ones proposed by Datar et al. [13]. Each LSH hash function $h_{\vec{a},b} : \mathbf{N}^d \to \mathbf{Z}$ maps a vector $\vec{r}$ into a single value ("bin") and is defined as:

$$h_{\vec{a},b}(\vec{r}) = \left\lfloor \frac{\vec{a}^T\vec{r} + b}{W} \right\rfloor$$

where $\vec{a}$ is a d-dimensional random vector with each component chosen independently from a Gaussian distribution[4],

[4]The Gaussian distribution is 2-stable; it can be shown that elements, which are close in the Euclidean distance sense, will be mapped to the same value with high probability and to distinct values otherwise [13].

W is the width of a bin, and b is a real number chosen uniformly from the range $[0, W]$.

In our scenario, we wish to pack together flow records based on the most often queried attributes. Therefore, we chose $\vec{r}$ to be an 11-dimensional vector ($d = 11$) consisting of the attributes source and destination IP addresses (2×4 bytes, i.e., 8 dimensions) as well as source and destination ports and protocol numbers (3 dimensions).

We reduce the probability of collisions of unrelated flows by computing the hashtable index value as a sum of many LSH functions. In detail, $H_1(\vec{r}) = \sum_{i=1}^{n} h_{\vec{a}_i,b_i}(\vec{r}) \mod P$ from the n mappings of $\vec{r}$, where P is the hash size. In addition, as collisions of unrelated records may still occur within each hash chain, chains are kept ordered using an InsertionSort algorithm. The key used for sorting in the InsertionSort is computed using a different combination of LSH functions H_2 that utilizes different projection spaces: $H_2(\vec{r}) = \sum_{i=1}^{n} h_{\vec{a}'_i,b'_i}(\vec{r}) \mod Q$.

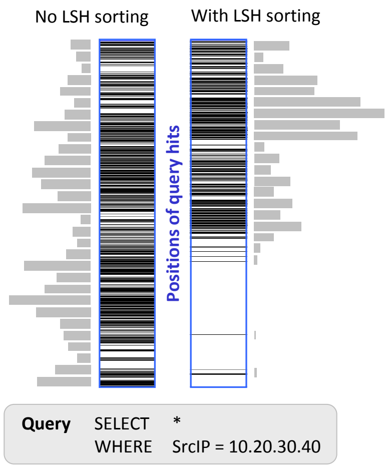


Figure 9: Comparison of positions of a query without (left) and with (right) LSH reordering.

The stream reordering process consists of inserting incoming flow records into the hash and dispatching new blocks to the compression and indexing components. Whenever the length of a chain reaches a configurable maximum threshold (*maxBlockSize*), the chain is removed from the hash and its content used to fill a block (or several blocks in case of collisions). We also employ two thresholds, *MMax* and *MMin*, to limit the number of records to be buffered (i.e., the memory budget). When the total number of flows stored by the hash reaches *MMax*, blocks are created by packing (and purging) the longest chains. The process stops when *cnt* reaches a value lower than *MMin*. A pseudocode of the online reordering process is shown in Program 1.

Figure 9 illustrates the benefits of online record reordering. We compare the row positions (matching records) returned by an IP query lookup when executed over two NET-FLi flow repositories storing exactly the same traffic and built without and with oLSH reordering enabled. In the unsorted version, matching positions are spread all over the column, whereas in the version using the online-LSH the matching records are concentrated in the first half. The data retrieval from compressed columns benefits from this because fewer blocks from the archive must be accessed and decompressed.

Program 1 oLSH reordering of the streaming records

```
processElement(hashtable hash, flow record r){
   P = hash.length(); // size of hashtable

   h1 = sum( h(a[i], b[i], r) ) mod P; // hashtable index
   h2 = sum( h(a'[i], b'[i], r) ) mod Q; // used for
                                       // insertionSort

   chain = hash[h1].insertionSort(r, h2);
   if (chain.length() > maxBlockSize)
     emitBlock(chain, archive, index); // send to archive
                                     // and index
   maxCount = hash.totalNumBuckets();
   if (maxCount > MMax){ // memoryBudget
     do{
       chain = longest_chain(hash);
       emitBlock(chain, archive, index);
     } while(hash.totalNumBuckets() > MMin); // minimum
   }                                         // threshold
}
```

5. EVALUATION

In this section we evaluate the performance of our solution and investigate critical performance metrics of the archiving and querying process. We use two datasets in the evaluation:

- Six days of NetFlow traces of access traffic from a *large hosting environment* (HE).
- A two-month NetFlow trace of internal and external traffic in an average-sized enterprise *production network* (PN).

The characteristics of the two datasets are listed in Table 1. The traffic of the two network environments differs significantly in terms of the distribution of IP addresses and service ports. The flow attributes included in the index and archived data columns are presented in Table 2.

Table 1: Datasets utilized

Dataset	# flows	Length	Size
Hosting Environment (HE)	231.9 million	6 days	6.9 GB
Production Network (PN)	1.2 billion	62 days	37 GB

Table 2: Flow attributes present in the index and archived data columns.

Attribute	Size	Index	Archive
Source IP address	4 Bytes	✓	✓
Destination IP address	4 Bytes	✓	✓
TCP/UDP source port	2 Bytes	✓	✓
TCP/UDP destination port	2 Bytes	✓	✓
Layer 3 protocol	1 Byte	✓	✓
TCP flags	1 Byte	✓	✓
Source AS number	2 Bytes	-	✓
Destination AS number	2 Bytes	-	✓
Number of Bytes	4 Bytes	-	✓
Number of packets	4 Bytes	-	✓
Flow start time	4 Bytes	✓	✓
Duration	4 Bytes	-	✓

We have implemented the indexing, storage, and querying techniques as a C++ library of 25 000 lines of code. The library comes with two similarly optimized implementations of both WAH and COMPAX compressed bitmap indexing algorithms. In both cases, boolean operations do not require any explicit bitmap decompression. The software does not require any external library except for LZO [25],

Table 3: Disk space needed to store the two datasets for different compression algorithms and storage approaches.

	Flat files			NET-FLi archive	
Dataset	**Raw**	**GZIP**	**LZO**	**LZO**	**LZO+oLSH**
HE	6.9 GB	2.5 GB	3.5 GB	3.7 GB	2.6 GB
PN	37 GB	8.1 GB	13.2 GB	13.8 GB	8.2 GB

which provides us the LZO1X-1 [26] algorithm implementation used for compressing data columns. Among the algorithms belonging to the LZO family, LZO1X-1 does not offer the best compression rates but it has instead been designed for achieving the best compression and decompression speed instead.

All experiments have been executed on a commodity desktop machine equipped with 2 GB of DDR3 memory and an Intel Core 2 Quad processor (Q9400) running GNU/Linux (2.6.28 kernel) in 32-bit mode. The processor has four cores running at 2.66 GHz and 6 MB of L2 cache. We store flows on a 320 GB desktop hard drive[5] formatted with a single Ext3 partition.

5.1 Disk utilization

We measure the disk consumption of our methodology for both the *archive* and the COMPAX encoded *bitmap index.*

Archive size: We illustrate the disk utilization of the raw flow data, of their compressed counterpart (the common case for non-indexed archives), and finally of our compressed columnar approach. To compress the raw data, we applied two widely used compression utilities: *gzip* and *lzo.* Moreover, we are interested in measuring the space saving when enabling the oLSH reordering of flows in our approach. We report our findings in Table 3.

As expected, the space required to store flow files compressed with LZO (which is optimized for speed rather than compression) is larger than that required for same *flat files* compressed with GZIP of the two datasets.

We observe that the total size of the column-oriented NET-FLi archive, when compressed with LZO (3.7 GB and 13.8 GB), requires 5 % and 6 % more space than the flat files compressed with LZO (3.5 GB and 13.2 GB). This was to be expected, because by compressing the data in small blocks (up to 4000 records each) the overall compression rate decreases. However, at the same time, the block-based compression method enables partial data decompression.

When enabling the oLSH on-the-fly record reordering, our approach can drastically reduce disk space consumption: up to 40 % for the PN dataset (2.6 GB) and up to 30 % for the HE dataset (8.2 GB). In both test datasets, the combination of oLSH with LZO compression allows our methodology to reach similar compression rates as the ones archived with GZIP when applied to the raw flow files.

Index size: We now measure the disk space savings induced on the bitmap index created when using the COMPAX encoding and compare it with the space size required by the WAH bitmap index methodology.

[5]The hard drive is a 7200 rpms Hitachi HDP725032GLA380 equipped with 8 MB of cache. The system is capable to perform cached reading at 2400 MB/s and unbuffered disk reads at 80 MB/s (measured with `hdparm`).

Table 4 reports the disk space consumption of WAH indexes, COMPAX indexes, and COMPAX indexes when oLSH-based sorting is enabled. Compared to WAH, the COMPAX-compressed indexes are 30% smaller for dataset PN and 40% for dataset HE. In the next section, we show that this is achieved *without any impact on the CPU load.* Enabling the oLSH component allows the disk consumption of the index to be further reduced. In fact, the combination of COMPAX with oLSH reordering results in indexes that are up to 60% smaller.

Table 4: Comparison of index sizes built using WAH, COMPAX and COMPAX+oLSH.

Dataset	WAH	COMPAX	COMPAX+oLSH
HE	8.1 GB	4.9 GB	3.3 GB
PN	26.3 GB	18.6 GB	12.8 GB

Aggregate savings: Now, we consider the aggregate storage savings, on both the archive and the bitmap indexes. By combining our compressed index bitmap and oLSH sorting, we see that the total disk consumption decreases from 40.1 GB to 21 GB for dataset PN and from 12.8 GB to 5.9 GB for dataset HE. **Thus, we managed to almost halve the total storage requirement**. Even with oLSH reordering disabled, we can still save space thanks to the enhanced compressed bitmap indexes. The total disk consumption of the archive and COMPAX-based indexes requires 20 % to 38 % less space than the same columns and WAH based indexes.

5.2 Stream Record Processing Rates: Archive and Index

NET-FLi has been designed to handle high-speed streams of flow records. In this section, we evaluate the average sustainable insertion rate, expressed in flows per second (f/s). We test our storage solution using as data-feeds historical uncompressed flow traces, which are stored on a mainstream solid state drive[6]. The drive provides a sustained reading speed of 170 MB/s corresponding to more than 5 million f/s. Archives and bitmap indexes are stored on a regular hard drive. This simple setup allows us to reproduce flow rates that can only be observed in very-large ISP networks.

We measure the insertion rate of our solution with and without enabling oLSH flow reordering. Indexes are built using the WAH and our COMPAX encoding (without and with stream reordering enabled). Table 5 reports the record-processing rates for building both the index and the archive.

Without enabling oLSH reordering, our storage solution can handle up to 1.2 million f/s. To put these number into perspective, we mention that medium-sized service provider network typically exhibit peak rates of 50,000 f/s.

Table 5: Record processing rates of our system when building both the index and the archive.

Dataset	WAH	COMPAX	COMPAX+oLSH
HE	768K f/s	936K f/s	474K f/s
PN	1150K f/s	1255K f/s	513K f/s

One important topic that we need to mention is that bitmap compression using COMPAX is actually more lightweight than WAH and results in higher record processing rates. Without reordering, our storage solution can handle

[6]Intel X-25M G1, 80 GB model

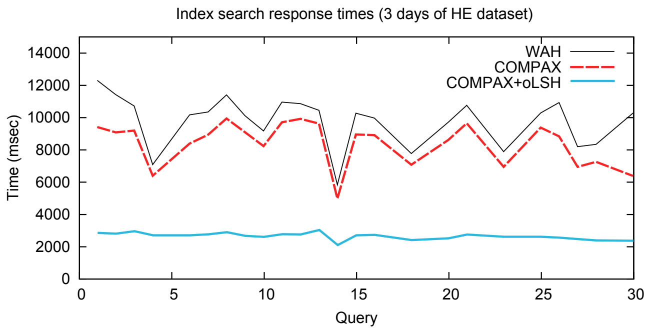


Figure 10: Comparison of index performance for three bitmap index variants

up to 0.93 million f/s (21 % more than WAH) for dataset HE and 1.26 million f/s (9 % more) for dataset PN. By using a profiler, we realized that COMPAX is much more cache friendly than WAH. Indeed, COMPAX performs its online compression steps by packing three WAH-words into a single word ([FLF] or [LFL]), resulting in fewer L2 cache misses on average, thus allowing more effective stream record processing.

When enabling the oLSH component, as expected, the insertion rate drops down to 474K f/s for dataset HE and 513K f/s for dataset PN. However, the reordering process eventually achieves as much as 55 % space savings. In addition, as we will demonstrate in the following section, the record reordering results in improved response times of the system.

5.3 Index Performance

First we compare the performance of the bitmap index only (without access to the archive), when the index is encoded using WAH, COMPAX, and COMPAX+oLSH. We pose queries on the IP address field, because because IP addresses are typically the most often queried attribute. By querying exact IP addresses, we evaluate the performance of boolean operations over compressed bitmaps as each IP lookup requires the bitwise ANDing of four different bitmaps, each corresponding to the four bytes of an IPv4 address.

For this experiment, we use a smaller subset of the data, which can be cached entirely by the system, by building the index using three days of flow data from the HE environment. The bitmap index sizes are 1.5 GB for WAH, 845 MB for COMPAX and 314 MB for COMPAX+oLSH. As queries we use 3000 random distinct IP addresses, which we grouped into 30 sets of 100 addresses each. For every set, we report the time for performing the 100 independent and sequentially executed IP address lookups. The time measurements are reported in Fig. 10.

The response times reported are determined exclusively by the performance of boolean operations on IP address byte indexes. Queries on the COMPAX-based index are on average 15% faster compared with WAH-based index. This result indicates that opting for the more elaborate COMPAX bitmap encoding does not necessarily penalize performance, because boolean operations are still performed directly on the compressed bitmaps (i.e., without explicit decompression). In fact, on the COMPAX-encoded index, the additional decoding complexity is compensated by the smaller working set, which is a consequence of the improved compression rate.

Finally, COMPAX+oLSH is observed to be four times faster on average than WAH for the same IP address lookups. In this case, the improvement is not just attributed to the increased cache locality, but rather to the sorting itself. The result of the sorting is that literal words are more dense whereas fill words can represent longer sequences. In this way, the number of bitwise instructions is substantially reduced [20].

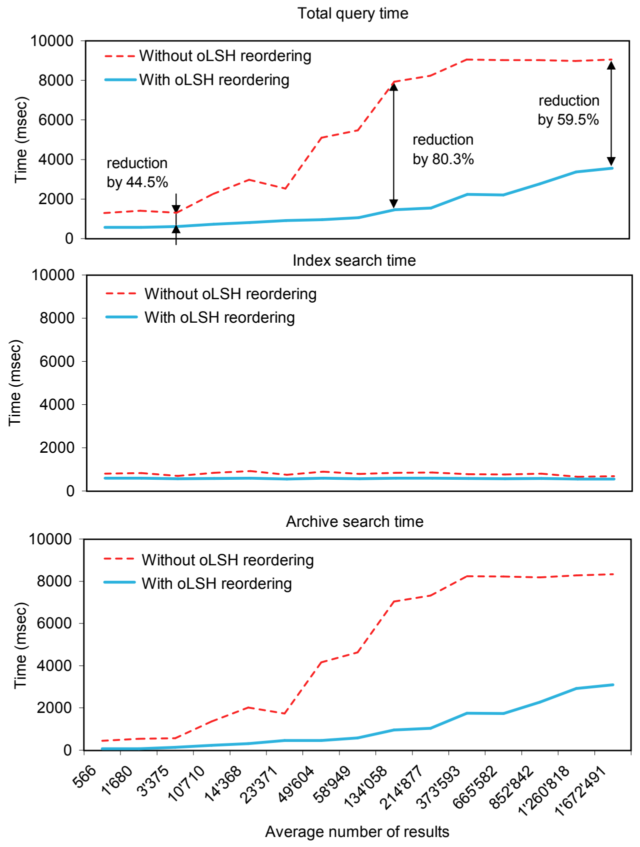


Figure 11: Query time vs. number of results: Comparison of the total query time (top) and its separation into time on the index access (middle) and data retrieval time from the archive (bottom), with and without oLSH reordering.

5.4 Query Performance: Index and Archive

Lastly, we measure the complete system performance under query operations. We create the index and the archive with and without the oLSH reordering. We measure the cumulative response time required when executing queries over the 1.2 billion flows of dataset PN. The disk consumption of the index and archive without the oLSH option amounts to approximately 31 GB. With oLSH enabled, the size of the resulting repository is 20 GB.

We pose random queries with increasing number of wildcards such as:

```
srcIP = 10.4.5.*, dstPort=X, dstIP = 10.5.5.*
srcIP = 10.4.5.*, dstPort=X, dstIP = 10.5.*.*
srcIP = 10.4.*.*, dstPort=X, dstIP = 10.5.*.*
...
```

to create queries with decreasing selectivity, leading to an increasing number of results. We sort and "bin" the number of results into a histogram format. Each bin is labeled with the average number of results returned by all the queries in this bin. In Fig. 11, we report the following three measures:

Index time The time needed to search the index and join the resulting indexes.

Archive time The time needed to retrieve the data from the archive.

Total query time The sum of the two above measures.

For each measure, we report the runtime with and without oLSH reordering. We notice that by using oLSH the total query time decreases by more than 40 %. This result is attributed to the smaller index and archive size of the oLSH variant. For the queries with high selectivity, the results are returned in less than a second, whereas for queries with lower selectivity the response time can be 3 seconds. This outcome is very encouraging because the query covers a time-span of two months.

The middle and lower parts of Fig. 11 segregate the total query time into its components. The time to retrieve results from the index and join them (middle) is nearly constant and approximately on the order of 500 msec. At the bottom, it can be observed that the data retrieval process from the archive contributes most to the total response time. At the same time, this graph highlights the significant performance boost provided by the reordering component. The careful pre-processing of flow records in both the index and the archive results in substantial query performance improvements. In particular, the effective record reordering decreases the system response time by up to 80 %. This is attributed to the better packing of similar records, reducing the number of data blocks to be decompressed and increasing the decompression speeds of individual blocks.

6. APPLICATION OF NET-FLi

In this section, we demonstrate the utility of NET-FLi in a typical use case scenario for network monitoring. A machine has been detected to be infected with a computer worm that exploits a vulnerability of an application protocol. To quarantine any other possibly infected machine, the administrator needs to know all machines that communicated with the infected machine using the vulnerable protocol.

Filtering a large flow repository for a small subset of flows is a tedious task, as the entire repository needs to be scanned linearly. Using the index capabilities of NET-FLi, we expect to achieve a significant reduction of the time to identify possibly infected machines.

We perform an analysis on the two-months PN dataset worth of 1.2 billion flows and focus on machine M that is suspected to spread a worm using a vulnerability on service port 445. We query the system for all machines that have been contacted by M on port 445. Therefore the query has the form:

```
SELECT DstIP
WHERE SrcIP = M AND DstPort = 445
```

In the introduction, in Fig. 1, we depict an propagation example of a similar worm epidemic; the center node represents a suspicious machine whereas an edge signifies a communication link on vulnerably port between the two machines. Such a graph can be created with NET-FLi by recursively querying for flows emitted by suspectedly infected machines in the network.

We measure the time needed to find all 2225 relevant graph connections and to retrieve the corresponding *dstIP* addresses using COMPAX+oLSH, with a completely empty cache (unmounting the disk), as well as with a "warm" cache at the operating system level, when other queries have been previously posed. We repeat the experiment 100 times. We discover that the uncached query response time is 62.314 sec (with standard deviation of $\sigma = 0.798$), on average. When reissuing the same query with a 'warm' cache, the response time drops down to 2.345 sec ($\sigma = 0.051$), on average. In comparison, the identical query executed on a conventional flat flow file repository using linear scanning over all records takes as much as 6062 sec, i.e., more than two orders of magnitudes longer.

Our results show that the NET-FLi approach exhibits low response times to locate the candidate records and return the flow data. In addition, NET-FLi can exploit the cache capabilities offered by the operating system, without requiring a dedicated cache/buffer manager. The last observation is particularly attractive for interactive query refinement: for example, in a network investigation, typically a number of queries are used to narrow down the root cause. Subsequently, refined queries on previously cached indices can be answered almost for free. In contrast, linear scan approaches cannot benefit significantly from the LRU-oriented cache system of the operating system.

7. LIMITATIONS AND FUTURE WORK

In the evaluation, we focused on indexing and archiving flow records with a fixed structure of fields without addressing records with varying structure, e.g., defined with templates. Nevertheless, our approach is still applicable under such conditions. By maintaining columns for all attributes and a template index, we can compensate for missing values in some of the records.

NET-FLi is modular and can accommodate different algorithms for column data compression. We plan to evaluate more efficient compression algorithms, such as PFOR [36], and to design algorithms optimized for specific attributes (e.g., start and end timestamps).

As NET-FLi is inherently parallel by design, we plan to develop a distributed architecture of a cluster of commodity PCs using NET-FLi as a back-end. Our experiments suggest that the compressed indexes exhibit specific patterns depending on the traffic behavior. We plan to study these patterns for visualization tasks [35] or for detecting traffic anomalies and inferring traffic properties directly on the compressed bitmaps.

8. CONCLUSION

We introduced NET-FLi, a high-performance solution for high-speed data archiving and retrieval of network traffic flow information. Our solution achieves:

- Data record insertion rates in the range of **0.5M to 1.2M flows per second** depending on the desired compression level.

- An adaptive, compressed bitmap indexing technique, called COMPAX, which **outperforms the state-of-the-art encoding** WAH in terms of CPU and compression efficiency.

- An on-the-fly stream reordering approach based on locality sensitive hashing (LSH) which renders the data compression rate of flow records **equivalent to that of gzip**.

- **Interactive response times** for typical queries, when sifting through gigabytes of compressed flow data.

Our solution can be used to drive a wide spectrum of applications including iterative hypothesis testing in network anomaly investigation, on-demand traffic profiling, and customized reporting and visualization. Moreover, we believe that it can be applied to many other domains challenged by high data-volumes and exceptionally high insertion rates.

9. REFERENCES

[1] AURORA: Traffic Analysis and Visualization. http://www.zurich.ibm.com/aurora/.

[2] Endace Measurement Systems, NinjaProbe Appliances. http://www.endace.com.

[3] FastBit: An Efficient Compressed Bitmap Index Technology. https://sdm.lbl.gov/fastbit/.

[4] Network Top. http://www.ntop.org/.

[5] Niksun NetDetector. http://niksun.com.

[6] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu. Scale-Up Strategies for Processing High-Rate Data Streams in System S. In *Proc. of ICDE*, pages 1375–1378, 2009.

[7] V. N. Anh and A. Moffat. Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval*, 8(1):151–166, 2005.

[8] G. Antoshenkov and M. Ziauddin. Query processing and optimization in Oracle Rdb. *The VLDB Journal*, 5(4):229–237, 1996.

[9] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Communications of ACM*, 51(12):77–85, 2008.

[10] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *Proc. of SIGMOD*, pages 668–668, 2003.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):1–26, 2008.

[12] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *Proc. of SIGMOD*, pages 647–651, 2003.

[13] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. of Symposium on Computational Geometry*, pages 253–262, 2004.

[14] F. Deliége and T. B. Pedersen. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *Proc. of EDBT*, pages 228–239, 2010.

[15] K. Fujioka, Y. Uematsu, and M. Onizuka. Application of bitmap index to information retrieval. In *Proc. of WWW*, pages 1109–1110, 2008.

[16] C. Gates, M. Collins, M. Duggan, A. Kompanek, and M. Thomas. More Netflow Tools for Performance and Security. In *Proc. of USENIX conference on System administration*, pages 121–132, 2004.

[17] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. of VLDB*, pages 518–529, 1999.

[18] P. Haag. Nfdump. http://nfdump.sourceforge.net/.

[19] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: Multilevel Traffic Classification in the Dark. In *Proc. of SIGCOMM*, pages 229–240, 2005.

[20] O. Kaser, D. Lemire, and K. Aouiche. Histogram-aware sorting for enhanced word-aligned compression in bitmap indexes. In *Proc. of International Workshop on Data Warehousing and OLAP*, pages 1–8, 2008.

[21] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *Data and Knowledge Engineering*, 69(1):3–28, 2010.

[22] X. Li, F. Bian, H. Z. 0002, C. Diot, R. Govindan, W. Hong, and G. Iannaccone. MIND: A Distributed Multi-Dimensional Indexing System for Network Diagnosis. In *Proc of INFOCOM*, 2006.

[23] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proc. of VLDB*, pages 950–961, 2007.

[24] C. Morariu, T. Kramis, and B. Stiller. DIPStorage: Distributed Architecture for Storage of IP Flow Records. In *Proc. of the 16th Workshop on Local and Metropolitan Area Networks*, 2008.

[25] M. F. Oberhumer. The Lempel-Ziv-Oberhumer Packer. http://www.lzop.org/.

[26] M. F. Oberhumer. Lzo documentation. http://www.oberhumer.com/opensource/lzo/lzodoc.php.

[27] A. Pinar, T. Tao, and H. Ferhatosmanoglu. Compressing Bitmap Indices by Data Reorganization. In *Proc. of ICDE*, pages 310–321, 2005.

[28] T. Plagemann, V. Goebel, A. Bergamini, G. Tolu, G. Urvoy-Keller, and E. W. Biersack. Using Data Stream Management Systems for Traffic Analysis - A Case Study. In *Proc. of Passive and Active Network Measurement (PAM)*, pages 215–226, 2004.

[29] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, and J. M. Hellerstein. Enabling real-time querying of live and historical stream data. In *Proc. of SSDBM*, page 28, 2007.

[30] S. Romig, M. Fullmer, and R. Luman. The OSU Flow-tools Package and CISCO NetFlow Logs. In *Proc. of USENIX conference on System administration*, pages 291–304, 2000.

[31] M. Stonebraker et al. C-store: a column-oriented DBMS. In *Proc. of VLDB*, pages 553–564, 2005.

[32] K. Wu, E. Otoo, and A. Shoshani. On the Performance of Bitmap Indices for High Cardinality Attributes. In *Proc. of VLDB*, pages 24–35, 2004.

[33] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(1):1–38, 2006.

[34] K.-L. Wu et al. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S. In *Proc. of VLDB*, pages 1185–1196, 2007.

[35] X. Yin, W. Yurcik, and A. Slagell. The Design of VisFlowConnect-IP: A Link Analysis System for IP Security Situational Awareness. In *Proc. of IEEE International Workshop on Information Assurance (IWIA)*, pages 141–153, 2005.

[36] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proc. of ICDE*, page 59, 2006.