Doctoral Thesis

High-speed indexing and archival of network measurement data

Author[s]:
Fusco, Francesco

Publication Date:
2012

Permanent Link:
https://doi.org/10.3929/ethz-a-007583321

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
High-speed indexing and archival of network measurement data

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
FRANCESCO FUSCO
Dottore Magistrale in Technologie Informatiche,
Università degli Studi di Pisa
born May 19, 1982
citizen of Italy

accepted on the recommendation of
Prof. Dr. Bernhard Plattner, examiner
Prof. Dr. Ernst W. Biersack, co-examiner
Dr. Xenofontas Dimitropoulos, co-examiner
Dr. Michail Vlachos, co-examiner

2012
Abstract

The Internet has become a global IT infrastructure providing ubiquitously accessible, interactive, and secure services used by a large fraction of the global population. To meet users’ expectations, network administrators require sophisticated monitoring infrastructures for detecting misconfiguration and faults, for measuring the performance, and for enabling timely reactions to security threats.

Passive monitoring has rapidly become the de-facto monitoring approach for getting deep insights into the actual status of production networks. Nowadays networks rely on network probes, which are embedded in network equipments or deployed as special purpose monitoring devices, that constantly monitor important network aspects. Therefore, current monitoring infrastructures are able to create large volumes of monitoring data.

Industrial and academic research mostly focused on the generation, collection, processing and analysis of network monitoring data streams with the primary goal of providing live views of diverse network aspects. These efforts have led to mature technologies for processing high-speed data streams in real-time. Nowadays, stream processing represents the foundation for the large majority of software and hardware based monitoring infrastructures deployed for operating current production networks. In a nutshell, the stream processing approach consists of applying a predefined set of queries to one or more data streams in a way that summaries of the data are continuously computed. This approach allows one to have a predefined set of information about the network streams without requiring the streams to be entirely recorded, hence the name single-pass analytics. Unfortunately, this also means that the information not captured by the current query set is lost forever.

In many emerging contexts including, but not limited to, cyber-security, this trade-off is undesirable. In particular, large corporations, financial insti-
tutions and high-security data centers are increasingly interested in efficient solutions enabling the collection of exact data streams, and the expedient analysis of large-scale repositories of historical network measurements particularly in case of security breaches. Enabling long-term historical analysis of massive volumes of network monitoring data is required to enable forensics, anomaly detection, and information leakage analysis tasks.

Advanced data collection systems are required to enable the archival of high-speed streams of network monitoring data and, most importantly, to enable fast explorations of large-scale repositories. Such systems have to support data archiving under extremely high-speed insertion rates and to produce archives still amenable to indexing and search. Current solutions that address the challenge of lossless storage of massive network monitoring data streams use off-the-shelf compression techniques, like GZIP and BZIP2. The main shortcoming of these solutions is that they do not offer efficient query processing, especially for queries targeting a small part of the dataset, as large data blocks are compressed and then retrieved using expensive decompression operations and serial scans of the archives.

In this thesis, we first focus on the storage, indexing, and data querying of high-speed streams of network flow information and we propose an architecture built upon novel lossless indexing and compression algorithms carefully optimized for the network monitoring domain. The architecture is capable of compressing high-speed streams of network flow records in real-time while achieving higher compression ratios than popular general-purpose compressors, and, more importantly, produces compressed archives that support partial decompression. Then, we describe an indexing architecture for packet traces that has been integrated into libpcap, the de-facto reference library for accessing packet trace repositories.

We make the following important contributions: (a) we propose a novel compressed bitmap index encoding that outperforms the current state-of-the-art both in terms of CPU load and disk consumption when indexing network flow traces and packet traces, (b) we introduce an online stream reordering mechanism that boosts both compression ratios and retrieval time of modern compressors and compressed bitmap indexes, and, (c) we describe RasterZIP, a novel lossless compressor that leverages indexes for providing fine-grained decompression granularity. RasterZIP achieves higher compression ratios than general purpose compressors by exploiting data patterns, such as the shared prefixes of reordered IP addresses, that are commonly present in networking data.
Internet è divenuta un’infrastruttura globale che fornisce servizi interattivi, sicuri, accessibili da ogni luogo usati da una larga frazione della popolazione mondiale. Per soddisfare le aspettative degli utenti gli amministratori di rete hanno bisogno di sofisticate infrastrutture di monitoraggio per rilevare problemi di configurazione e guasti, per misurare le prestazioni e per rendere possibili reazioni tempestive a minacce di sicurezza.

Il monitoraggio di tipo passivo è di fatto diventato il paradigma di monitoraggio per ottenere informazioni dettagliate sullo stato corrente delle reti in produzione. Attualmente le reti si affidano a sonde integrate in apparati di rete o installate sotto forma di apparati di monitoraggio dedicati, che analizzano costantemente importanti parametri di rete. Pertanto le infrastrutture di monitoraggio attuali sono capaci di creare grandi volumi di dati.

La ricerca industriale ed accademica si è concentrata in larga misura sulla generazione, sulla raccolta, sull’elaborazione e sull’analisi dei flussi di dati di monitoraggio con l’obiettivo principale di fornire viste aggiornate di diversi parametri di rete. Queste attività hanno creato tecnologie mature per elaborare flussi di dati in tempo reale. L’elaborazione di flussi rappresenta oggi la fondamenta per la grande maggioranza delle infrastrutture di monitoraggio sia software che hardware, che sono installate per gestire le reti in produzione. In sintesi, l’elaborazione dei flussi consiste nell’applicare un insieme predefinito di queries ad uno o più flussi in modo tale che riassunti dei dati sono continuamente calcolati. Questo approccio permette di avere un insieme predefinito di informazioni riguardo ai flussi di rete senza richiedere che i flussi siano completamente archiviati. Da questa proprietà nasce il nome di single-pass analytics. Sfortunatamente questo significa anche che le informazioni non catturate dall’insieme corrente di query è perso per sempre.

In molti contesti emergenti, come per esempio la sicurezza informatica,
Questo compromesso non è accettabile. Grandi organizzazioni, istituti di finanza e operatori di centri di calcolo ad elevato grado di sicurezza sono infatti sempre più interessati a soluzioni efficienti che abilitino il salvataggio di interi flussi di dati e l’analisi di grandi moli di dati storici specialmente nel caso di violazioni di sicurezza. Analisi storiche di grandi volumi di dati di monitoraggio sono necessarie per effettuare investigazioni informatiche e per rilevare anomalie e fughe di informazioni.

Per archiviare flussi ad alta velocità di dati di monitoraggio e soprattutto per permettere esplorazioni interattive di grandi moli di dati servono sistemi di archiviazione avanzati. Questi sistemi devono permettere l’archiviazione di dati ad alta velocità e nel frattempo produrre archivi che forniscono funzionalità di indicizzazione e ricerca. Le attuali soluzioni che affrontano il problema di archiviazione lossless di flussi di dati di monitoraggio sono basate su compressori comunemente disponibili, come GZIP o BZIP2. Il più grande limite di queste soluzioni è che non offrono metodi efficienti per la ricerca, soprattutto quando i dati richiesti sono una piccola frazione dell’archivio, perché comprimono i dati in grandi blocchi che devono essere compressi e poi acceduti utilizzando costose operazioni di decompressione e scansioni lineari.

In questa tesi ci concentriamo inizialmente nell’archiviazione, nell’indicizzazione e nell’interrogazione di flussi di dati di monitoraggio ad alta velocità e proponiamo un’architettura basata su algoritmi di compressione e indicizzazione di tipo lossless ottimizzati per il monitoraggio di rete. L’architettura può comprimere flussi di rete in tempo reale ottenendo livelli di compressione migliori di quelli che possono essere ottenuti con compressori generici e, soprattutto, crea archivi compressi che possono essere decompressi anche solo in parte. In seguito descriviamo un’architettura di indicizzazione per tracce di pacchetti che è stata integrata in libpcap, che, di fatto, rappresenta la libreria di riferimento per accedere ad archivi di pacchetti.

Introduciamo i seguenti importanti contributi: (a) proponiamo un nuovo encoding per gli indici bitmap compressi che è migliore dello stato dell’arte sia in termini di carico di CPU che di spazio disco quando usati per indicizzare flussi di rete e pacchetti, (b) introduciamo un meccanismo di riordinamento dei flussi che migliora il livello di compressione e i tempi di risposta di moderni compressori e indici bitmap compressi, (c) descriviamo RasterZIP, un nuovo compressore lossless che sfrutta gli indici bitmap compressi per garantire una decompressione a grana fine dei dati. RasterZIP fornisce livelli di compressione migliori dei compressori generici sfruttando caratteristiche, come prefissi comuni in indirizzi IP, generalmente presenti nei dati di rete.
Table of Contents

Abstract iii

Sommario v

Table of Contents vii

List of Figures xi

List of Tables xiii

1 Introduction 1

1.1 Network measurements .......................... 1

1.2 Motivation ........................................... 2

1.3 Problem statement ................................. 4

1.4 Scope of Work and Contributions .................. 4

1.5 Outline .............................................. 5

2 Background 7

2.1 Network Traffic Monitoring ........................ 7

2.2 Stream Processing ................................... 8

2.3 Traffic monitoring technologies .................... 10

2.3.1 Packet level traffic analysis ................... 10

2.3.2 Flow level traffic analysis .................... 14

2.4 Data Stream Warehousing ........................ 19

2.4.1 The settings ..................................... 20
## Table of Contents

2.4.2 Compression ...................... 22
2.4.3 Indexing .......................... 23

3  Related work ......................... 27
   3.1 Data management systems ............ 27
   3.2 Traffic recording systems .......... 29
   3.3 Network Flow collectors ............ 29
   3.4 Cloud-based network traces analysis 30
   3.5 Compression ........................ 31
   3.6 Indexing ............................ 33

4  NET-FLi: indexing and compression of streaming flow records 35
   4.1 Architecture ......................... 36
       4.1.1 Compressed Archive of Flow Data 38
       4.1.2 Compressed Index of Flow Record Attributes 39
       4.1.3 Querying the System ............ 41
   4.2 COMPRESSED Adaptive index - COMPAX 43
       4.2.1 Encoding ......................... 43
       4.2.2 Comparison with other bitmap indexing schemes 47
       4.2.3 On-the-fly Bitmap Index Creation 50
   4.3 Online-LSH (oLSH) Stream Reordering 51
   4.4 Evaluation ........................... 56
       4.4.1 Disk Utilization ................. 58
       4.4.2 Stream Record Processing Rates: Archive and Index 61
       4.4.3 Extended Comparison with PLWAH 63
       4.4.4 Index Performance ................ 66
       4.4.5 Query Performance ............... 67
   4.5 NET-FLi at work ..................... 72
   4.6 Summary ............................ 75

5  RasterZip: a real-time compressor with partial decompression 77
   5.1 Overview of the Approach ............ 78
       5.1.1 Online Record Reordering ......... 80
   5.2 Algorithm ........................... 81
       5.2.1 Compression ....................... 82
       5.2.2 Encoding properties .............. 86
List of Figures

2.1 Pcap packet trace format ................................. 13
2.2 Traffic monitoring architecture based on flow records. 16
2.3 NetFlow v5 format. ...................................... 17
2.4 Bitmap index example ................................. 25

4.1 NET-FLi overview ................................. 37
4.2 The flow record we use in our setting. 38
4.3 An example of a bitmap index using equality encoding. 40
4.4 Example of a query execution. ......................... 42
4.5 Compressed bitmap indexes encodings ................. 44
4.6 Example of bit packetization in the [LFL] codeword. 46
4.7 online-LSH (oLSH): an approximate data stream sorting 52
4.8 oLSH reordering: positions ......................... 55
4.9 Bitmap indexes visualization ......................... 56
4.10 Visual representation of the oLSH reordering .... 62
4.11 Comparison between COMPAX and PLWAH .......... 65
4.12 Index lookup time .................................... 66
4.13 Query time vs. number of results .................... 69
4.14 Decompression speed in MB/sec. ..................... 71
4.15 Connectivity graph of a worm ....................... 73
4.16 Graphical application for exploring flows .......... 74

5.1 Compression methodology overview .................. 79
5.2 Cumulative difference between consecutive values .. 80
5.3 Row-wise and column-wise compression .......................... 82
5.4 Flowgraph of RasterZip compression ............................... 83
5.5 RasterZip encoding: V-blocks ........................................ 84
5.6 RasterZip encoding: B-blocks ........................................ 85
5.7 Full-Decompression (FD) ............................................. 89
5.8 Byte traversal technique ............................................. 90
5.9 Partial Decompression (PD) .......................................... 91
5.10 Grid Bitmap .......................................................... 93
5.11 Adaptive decompression component .................................. 95
5.12 A 10 × 10 grid model .................................................. 96
5.13 Adaptive decompression strategy ..................................... 100
5.14 RasterZip over LZO query response time: PN dataset ........ 101
5.15 RasterZip over LZO query response time: HE dataset .......... 102

6.1 Packet indexing and retrieval overview ......................... 106
6.2 Packet indexer .......................................................... 108
6.3 PcapIndex query workflow ........................................... 110
## List of Tables

4.1 Used data sets .............................................. 57  
4.2 Flow attributes present in the index and archived data columns. 57  
4.3 Disk space requirements when compressing with gzip and lzop 59  
4.4 Data compression with and without oLSH .................... 59  
4.5 Index sizes without the oLSH component enabled ............. 60  
4.6 Index sizes with the oLSH component enabled ............... 61  
4.7 NET-FLi record processing rates ............................ 62  
4.8 Index sizes for uniform distribution ........................ 64  

5.1 Utilized Datasets ....................................... 97  
5.2 Storage space requirements for different compression algorithms. ................................................. 98  
5.3 Record processing rates using RasterZip and LZO .......... 98  

6.1 Index size using WAH, PLWAH and COMPAX. ............... 113  
6.2 Indexing times vs reading times ............................ 114  
6.3 pcapIndex query response time on a solid state drive .... 115  
6.4 pcapIndex query response time on a regular hard drive .... 115
Chapter 1

Introduction

In the digital era, large volumes of data are created every day by digital devices and applications. Internet has quickly become the preferred networking infrastructure for distributing, sharing, moving and accessing information across the globe. Many technological advances made this feasible. Among them, compression and search technologies played an important role by making the data easily accessible; network monitoring made the Internet operational, highly available and capable of enabling the deployment of heterogeneous services used on a daily bases by millions of users.

The aim of this research is to focus on technologies that enable the collection and expedited search of network measurement data, which is continuously produced by sensing devices or meters deployed nowadays in the large majority of networks to constantly analyze their status.

This chapter introduces the issues addressed, the motivations and the research contributions of this work.

1.1 Network measurements

Network monitoring involves the analysis of network measurement data collected over time from networks by network equipment, networked devices and applications.

Two approaches are predominant for network monitoring: active and passive monitoring. Active monitoring relies on the capability of injecting traffic
into the networks to be monitored to retrieve measurements that capture how the network reacted to the synthetic traffic. Passive network monitoring relies on sensors, usually referred to as *network meters* responsible of observing and analyzing the traffic as it passes by. Meters can be implemented in dedicated monitoring devices and appliances, or can be embedded into network equipment such as routers and switches.

By using devices capable of analyzing real network traffic in real-time, network operators have detailed *live* views of their networks and can understand the behaviour of real services offered through the monitored network. Passive monitoring enables security applications (e.g., intrusion and anomaly detection), network monitoring applications (e.g., billing) and service monitoring applications (e.g., Service Level Agreements (SLAs) analysis).

Passive monitoring devices are deployed nowadays in almost every network. Router and switches from major network equipment vendors, such as Cisco, come with built-in network meters capable of producing streams of measurement records, referred to as *flow records*. Flow records are structured data types that store fields, such as IP addresses and bytes exchanged, that allows conversations between network endpoints to be summarized. Standalone network appliances responsible to monitor specific network aspects (e.g., network security) through the analysis of network packets are also deployed in corporate networks.

### 1.2 Motivation

Network monitoring has been the driving force for a significant research on stream processing of network measurements. Stream processing research has laid important foundations for capturing, manipulating, and analyzing streaming data directly as it arrives, without requiring the data stream to be stored. However, in the network monitoring context many emerging applications require not only the continuous processing of streams of measurements data, but also the exact storage of incoming data streams over long time windows (e.g., years). This stems from the need to perform historical data analyses on the data streams, particularly for tracing back the causal nexus of events that led to a particular event.

Corporate and service provider networks, financial institutions, high-security data center operators and industries relying on SCADA (supervisory control and data acquisition) systems [83] for monitoring and controlling in-
1.2 Motivation

Industrial systems, 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: the system administrator would like to quickly pinpoint the accessed nodes from the list of suspected IP addresses to isolate additional compromised nodes. A similar scenario is also encountered during the outbreak of a computer worm, when one would like to identify the computer nodes that have been contacted by a compromised system.

To support the above functionality, all inbound and outbound traffic can be recorded in order to recreate the original breach or attack conditions. By storing the traffic it is also possible to provide the evidence for network forensics, to show that sensible performance metrics have satisfied the expected Service Level Agreements (SLAs), and to troubleshoot complex network issues.

An approach for storing network traffic is to use packet loggers storing a recent window of the traffic (e.g., the last day of traffic). When packet loggers are deployed on high-speed networks, they must be able to store millions of network packets in real-time. For this reason, high-speed traffic recording is in many cases accomplished by deploying packet-to-disk products commercialized by specialized companies [48, 91, 103]. Packet loggers are also embedded into Intrusion Detection Systems (IDSs), such as Snort [113], for storing network traffic that matches specific network patterns or traffic marked as malicious or anomalous.

A less intrusive and more commonly adopted approach for keeping historical network traffic data consists of collecting flow records produced by the existing network infrastructures instead of full packets. In that way, one could still capture information such as source and destination IP addresses and ports, transport protocol and the amount of data transferred between Internet endpoints, but avoid recording the actual packet content, something that would require order of magnitude larger storage repositories and would severely compromise user privacy.

In both cases, large volumes of measurement data are accumulated over time. Collecting large volumes of network measurement data and warehousing the data for long time windows poses several, potentially orthogonal, challenges:

- **Real-time collection:** High-speed streams of network measurements have to be collected in real-time. The stream arrival rate can reach the order of tens of millions per second in the case of packets and hundreds of thousands in the case of flow records.
- **Real-time compression**: Network measurements have to be compressed for reducing data volumes and disk bandwidth requirements both at collection and retrieval time.

- **Efficient searches over compressed data**: To be of practical utility, network measurement repositories should provide mechanisms enabling fast search and retrieval of desired data from large volumes of compressed data.

## 1.3 Problem statement

Domain-optimized compression and indexing technologies have been developed in many information technology fields where large-scale collections of digital content has to be archived and efficiently retrieved. Mature compression and search technologies have been researched, developed and sometimes standardized for domains including, but not limited to, bioinformatics, biochemistry, and web information retrieval. These experiences have shown that substantial gains can be obtained by designing algorithms tailored to the specific domain. Contrary to what happened to these domains, the traditional approach to reduce volumes of measurement data used by monitoring applications consists of applying general purpose compressors. However, standard compression algorithms and tools are not designed to compress high-speed data streams in real-time. In addition, once network measurements have been stored, eventually in a compressed format, they have to be easily and quickly accessible to enable interactive exploration of the collected data. Sifting through large volumes of network measurement repositories storing data compressed using general purpose compressor can be very expensive, particularly when network operators are interested in identifying *needles in the haystack*.

## 1.4 Scope of Work and Contributions

The goal of this research is to explore new exciting avenues in the field of stream indexing and compression to enable the implementation of efficient software architectures capable of storing high-speed network traffic streams in real-time using off-the-shelf hardware and domain-specific indexing and compression algorithms tailored for the network monitoring domain.
In this thesis, we propose indexing and compression algorithms for enabling collection and retrieval of large-volumes of network measurement data. In particular, the contributions of this work are as follows:

- a novel encoding scheme for compressed bitmap indexes that outperforms the current state-of-the-art compressed bitmap indexing encodings, WAH [126] and PLWAH [39], when applied to the context of network measurements indexing.

- a novel stream-based technique that approximately sorts a stream of network flow measurements in a way that higher compression ratios can be achieved when compressing the data or when indexing it with compressed indexes.

- the design of a novel domain-specific data compressor that can exploit external indexes to provide fine-grained decompression granularity.

- the design and the implementation of an architecture built upon the aforementioned algorithms, that is capable of indexing and compressing high-speed streams of measurement data in real-time.

- the design and implementation of the first indexing scheme for packet traces based on compressed bitmap indexes. We also built this index into the reference packet capture library libpcap [10].

- an extensive comparison of different bitmap indexing encodings in the context of network monitoring.

1.5 Outline

The thesis is structured as follows:

- **Chapter 2** introduces basic concepts and settings of passive monitoring infrastructures and describes the types of measurement data usually collected.

- **Chapter 4** presents an architecture designed for storing and indexing high-speed streams of network measurements tailored for network flow records.

- **Chapter 5** describes a novel domain-specific compressor carefully engineered for compressing network flow traces.
Chapter 6 shows that the proposed architecture can be easily embedded into the de-facto reference library for accessing packet traces for providing indexed packet searches.

Chapter 3 discusses related works in these fields.

Chapter 7 concludes the thesis.
Chapter 2

Background

In this chapter we describe concepts from the network monitoring domain that are relevant to this dissertation.

2.1 Network Traffic Monitoring

Network traffic monitoring is a fundamental aspect of network management. Understanding the traffic behaviour is necessary for assisting network operators and network managers to accomplish several management tasks. Depending on the task, several metrics have to be measured (or computed) from the monitored network traffic.

Traffic engineering (TE): The goal of traffic engineering is to maximize the network resource utilization, while being able to meet Quality of Service (QoS) requirements. Traffic engineering techniques require the measurements of many network performance metrics in order to implement timely reactions and to perform capacity planning.

Billing: Traffic monitoring infrastructures enable the creation and the application of cost models for selling network resources. Simple cost models can be based on the actual bandwidth used by each user. By having more advanced traffic monitoring architectures in place, more complex business model can be defined and adopted. The network usage cost can be defined not only in terms of bandwidth utilization, but also in terms of other parameters such as the time of utilization (e.g., day versus night), expected quality of service (e.g.,
low vs high latency) and location (e.g., surfing the Internet might be more expensive in busier cells of a cellular network).

**Network troubleshooting:** Monitoring the actual network traffic is important for enabling proactive detection and implementing early reactions to network issues caused by misconfigurations, hardware/software defects, and security threats (e.g. Denial of Service (DoS) attacks).

**Security analysis:** Traffic monitoring is performed by tools for security management. Intrusion Detection and Prevention Systems (IDSs and IPSs) tools are nowadays widely deployed for recognizing malicious traffic automatically generated by malware software or for detecting network attacks.

**Service monitoring:** In the Internet of Services (IoS) era, traffic monitoring infrastructures are deployed by network operators for having fine-grained information about the services deployed. The required information can be derived by inspecting and dissecting packets up to the application level protocols to measure service specific metrics, such as the Jitter for Voice Over IP calls [56, 57].

**Lawful interception (LI) and forensic analysis:** Analyzing the network traffic is required in forensic contexts to provide digital evidence of illegal activities. Legal interception capabilities are in place in almost every country and are implemented using standards developed by institutions such as the European Telecommunications Standards Institute (ETSI).

### 2.2 Stream Processing

Advances in interconnection technologies, together with the spread of the Internet as a universal communication network, have favored and founded an active research in stream processing. This research has led to a rich family of advanced single-pass algorithms for counting distinct elements of the streams, for detecting heavy hitters, for measuring the stream entropy. Single-pass analytics are nowadays used not only for processing streams of network data, but also for analyzing streams of URLs generated by web users (i.e. web logs), streams of clicks that can be used for implementing campaigns for targeted advertisements, and streams of financial data (i.e. FIX streams [11]).

A data stream is an unbounded sequence of \((\text{record}, \text{timestamp})\) pairs. A stream can be raw or derived. A raw stream is made of data records continuously produced over time (e.g., sensor measurements), whereas a derived
stream is composed of metrics continuously computed by a stream processor engine (SPE) starting from a set of running data streams.

A stream processor engine can be a software application or a hardware device whose task is to continuously support a static set of *continuous* queries over on-line data. A continuous query is defined in a way that allows specific properties of one or more data streams to be summarized without storing the entire (and possibly infinite) stream on disk. A stream processing engine can keep the state locally and be regularly polled or produce another data stream as a result of its analysis. In this case, the stream processor is both a sink and a source of data. Stream processing engines can be combined such that modular and possibly distributed stream processing architectures can be implemented.

Network traffic monitoring has been the natural playground for stream processing. Stream processing has been introduced in the context of network traffic monitoring for being able to process high-speed network traffic in real-time. In a traffic monitoring context, stream processing engines have to be able to process millions of packets per second. The common approach for satisfying the performance requirements is to organize the analyses in layers and to minimize the data transfers across the layers: the analyses are performed as close as possible to the data itself (i.e., at the lower layers) and intermediate results are propagated to the upper layers. Bottom layers, usually mapped to hardware, provide high performance and low flexibility, whereas upper layers are usually mapped to software, which offers lower performance but higher flexibility. The key to meet the performance is to design a balanced data path where each layer can perform its task within the performance constraints. This is a challenging task that is hard to accomplish unless specific domain knowledge is used.

Stream processing applications for traffic analysis are implemented using the following approaches:

- **Ad-Hoc Stream Processing Solutions** are preferred for the contexts where performance requirements are critical (e.g. traffic analysis on backbone links) and for specific applications. The performance requirements are met by implementing ad-hoc architectures created with an integrated hardware, firmware and software co-design.

- **Stream Processing Frameworks** [56] assist and facilitate the development of traffic analysis applications by providing dedicated components for performing and accelerating common tasks, such as traffic capture, packet filtering, IP defragmentation and TCP reassembly.
Stream processing engines are implemented with standard programming languages (usually C/C++ for performance reasons).

- **Data Stream Management Systems (DSMS)** are special purpose databases where stream processing engines can be defined with SQL-like languages augmented with stream-aware operators (e.g. data stream windows).

### 2.3 Traffic monitoring technologies

#### 2.3.1 Packet level traffic analysis

Passive monitoring at the packet level consists of analyzing network packets as they are transmitted over the network. Recent advances in processors and interconnection technologies made off-the-shelf hardware attractive for processing packets at high-speed.

*Packet capture* is the process of feeding a monitoring application with a copy of the packets as seen on the wire together with additional information (e.g., the time when the packet is captured) that is not included in the packet itself, but is known at capture time and can be useful for analyzing the traffic. The most popular packet capture library is *libpcap* [10]. Libpcap is available for almost every operating system and it is used by the large majority of packet-level analysis software, such as the ubiquitous *tcpdump*, and the *de-facto* reference protocol analyzer *Wireshark* [13].

Capturing packets without losses from high-speed links (Gbit and beyond) requires sophisticated high-performance packet capture solutions. Special-purpose network interface cards, referred to as packet capture accelerators, are extensively used to accelerate the packet capture task. Significant research efforts have been focused on improving the packet capture performance offered by general purpose operating systems and commodity hardware [24, 25, 27, 40, 41, 55, 57, 106].

A monitoring application analyzing the traffic at the packet level might be interested in accessing the full packet stream, or only a subset of it (e.g., only packets on a specific application port). *Packet filtering* can be implemented by the monitoring application running in user space, or at lower levels for efficiency reasons. Implementing packet filtering functionalities at the lowest possible hierarchy level allows computing resources and bandwidth to be saved.
2.3 Traffic monitoring technologies

The de-facto reference packet filtering mechanism is the Berkeley Packet Filter (BPF) [86]. It is adopted by the large majority of operating systems and packet capture libraries, including the widely-used libpcap. BPF filters packets matching a filtering expression, which is specified using the BPF language and evaluated against each network packet by a register-based virtual machine (VM). To do so, the filtering expression has to be first compiled by the BPF compiler into instructions that the BPF Virtual Machine can interpret. Program 1 shows how a simple BPF filter gets compiled into instructions to be executed by the BPF Virtual Machine.

Program 1 BPF instructions for the filter ”ip6 and tcp and port 80”

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Value</th>
<th>Jump Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ldh</td>
<td>[12]</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>jeq</td>
<td>#0x86dd</td>
<td>jf 9</td>
</tr>
<tr>
<td>002</td>
<td>ldb</td>
<td>[20]</td>
<td></td>
</tr>
<tr>
<td>003</td>
<td>jeq</td>
<td>#0x6</td>
<td>jf 9</td>
</tr>
<tr>
<td>004</td>
<td>ldh</td>
<td>[54]</td>
<td></td>
</tr>
<tr>
<td>005</td>
<td>jeq</td>
<td>#0x50</td>
<td>jf 6</td>
</tr>
<tr>
<td>006</td>
<td>ldh</td>
<td>[56]</td>
<td></td>
</tr>
<tr>
<td>007</td>
<td>jeq</td>
<td>#0x50</td>
<td>jf 9</td>
</tr>
<tr>
<td>008</td>
<td>ret</td>
<td>#96</td>
<td></td>
</tr>
<tr>
<td>009</td>
<td>ret</td>
<td>#0</td>
<td></td>
</tr>
</tbody>
</table>

The program starts by loading the Ethernet-type value (16 bits starting from the byte at position 12 of the packet) into the accumulator using the ldh instruction. If the retrieved Ethernet-type value is not the one corresponding to IPv6 (0x86dd), the execution jumps to the line 009 for returning false. Otherwise, the program continues with the line 002 for fetching, and storing into the accumulator, the 8 bits value where the protocol type is specified (the value is stored in the byte at position 20 within the packet). In line 003, the value stored in the accumulator is compared with the protocol type corresponding to TCP (0x6). If the value is the same, the instruction ldh of line 004 gets executed in order to load the source port field (16 bits starting from the byte in position 54). The rest of the program is interpreted as explained above.

A monitoring application that receives packets from a packet capture component can either perform the traffic analysis on-the-fly in a streaming manner (e.g. without storing the packet stream) or save the packet stream to disk. In the latter case, the monitoring application is usually referred to as packet logger producing packet traces of the captured traffic. Packet loggers are configured to keep a rotating window of the traffic (e.g., the last day of traffic). When deployed on high-speed networks, packet loggers must be able to store
millions of network packets in real-time. The width of the window is limited by the capacity of the storage subsystem whereas the maximum sustainable packet rate depends both on the packet capture performance and on the bandwidth provided by the storage facility.

Packet logging can be performed by special purpose appliances, usually referred to as packet-to-disk appliances, commercialized by companies specialized in high-speed network monitoring such as Endace [48] and Niksun [91]. Those appliances come with dedicated hardware for accelerating packet capture and are capable of logging at wire-rate from multi Gigabit links. Therefore, packet-to-disk appliances are able to log several Terabytes of packet traces per day. Packet logging functionalities are also embedded into Intrusion Detection Systems (IDS), such as Snort [113], for storing only the traffic that has been considered malicious or anomalous.

Packet traces are flat files containing packets and the metadata information that allow the repository to be explored. To reduce the disk consumption, the metadata information is usually limited to few fields (e.g. packet lengths and timestamps) that allow the original packet stream to be reconstructed. The de-facto packet trace format is the one used by libpcap. Commercial packet loggers either allow one to capture the traffic in pcap format or provide tools for converting the traces into the pcap format.

Figure 2.1 depicts the format of a pcap encoded packet trace. The trace starts with a trace header followed by a record for each captured packet. The record is composed of a record header, the pcap packet header, and the packet itself as captured from the network link. The pcap packet header stores a timestamp, corresponding to the time when the packet was captured, the length of the packet as transmitted on the link, and the capture length, which is the portion of the packet that is actually retrieved.

The packet retrieval process can be implemented using a standard libpcap application such as the one sketched in Algorithm 1. The application linearly scans every packet in the packet trace, and, if a Berkeley Packet Filter has been specified, it applies the filter against each packet.

Surprisingly, current packet-trace analysis solutions do not provide support for indexing schemes. Widely used formats for packet traces, such as pcap or the Endace Record Format (ERF) [48], do not include metadata information (e.g. indexes) that would accelerate searches over the trace repository. As a result, when network operators are interested in identifying a needle in a haystack the entire repository has to be linearly scanned.

To overcome the lack of indexing mechanisms, commercial products from
leading companies [48, 91] rely on hardware-assisted traffic transmission and filtering capabilities offered by capture accelerator devices. Instead of using an index to speed up searches of packets matching a rule set, the entire traffic stored on disk as a set of packet traces is replayed at wire-rate on the management network where another monitoring station is responsible for capturing the traffic matching the rule set (e.g. all the traffic on port 123). By combining hardware accelerated transmission and filtering provided by capture accelerators and the high sequential reading speed of disk arrays, a mechanism that has the same effect as indexing can be offered to network operators without
Background

paying for the additional disk space required to store indexes.

Therefore, the main advantage of this approach is the reduced storage footprint whereas the major drawback is the need of a management network and dedicated devices for traffic transmission and filtering. The approach described above is best suited for the cases where the selectivity is low\(^1\). If the selectivity is high and only a small fraction of packets are needed, having a true disk-based index may provide substantial benefits in terms of response times.

The most popular packet trace processing tools, *tcpdump* and *Wireshark*, use the same mechanisms offered by *libpcap* [10] for filtering packets within a packet trace file. This means that BPF filtering expressions have to be evaluated against each packet. There are two major performance bottlenecks when doing so. First, the entire packet trace has to be read even when the selectivity is extremely high (e.g., when a single packet has to be extracted from a trace storing millions of packets). Second, every time a filtering expression is used, all the packets are implicitly parsed by the BPF engine. The 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. This could have been avoided with the use of an index.

There are three main reasons that prevented indexing facilities being included into packet trace formats. First, packet trace formats should be as simple as possible to simplify encoding and decoding, and therefore their acceptance. Second, auxiliary data structures for indexing require additional disk space that may represent a serious concern in many environments. Third, indexing the traffic on the fly may be too computationally expensive if performed while dumping the traffic on disk.

\subsection{2.3.2 Flow level traffic analysis}

Network flow monitoring is an application of the stream processing model introduced in the network monitoring context in order to allow the traffic flowing through high-speed network links to be analyzed. Flow monitoring has become the key technology for operating large networks as it provides deep insights into the network traffic while satisfying the scalability requirements. Flow monitoring enables applications such as network troubleshoot-

\(^1\)The selectivity is low when the number of desired packets is a big fraction over the entire repository.
2.3 Traffic monitoring technologies

A network flow can be broadly defined as a network conversation between end points. A flow record is structured data type storing information that summarizes an individual network flow. A network conversation can be defined as TCP connection, a video stream, or even as a Voice-Over-IP (VoIP) call. According to Cisco’s standards, a network flow is an unidirectional sequence of packets observed at a vantage point, having the same values in all the following fields: source address, destination address, source and destination ports, IP protocol, and type of service.

Network traffic analysis architectures implementing the network flow monitoring approach are made of flow meters and flow collectors. A flow meter is essentially a stream processing engine that receives as input a stream of network packets and produces as output a stream of flow records summarizing each individual observed network flow. A flow collector receives the stream of flow records and performs some further processing. Flow meters maintain statistics on each observed flows, such as the number of packets and bytes transmitted. Once a flow is considered expired, the statistics are exported as a flow record to a centralized flow collector, using an export protocol, such as Cisco’s ubiquitous NetFlow [33]. A flow record is expired when a TCP connection is closed by one of its pairs, or when the flow meter does not see any packet on the network belonging to the flow for a configurable amount of time, or at fixed time intervals in order to have information about flows that stay active for a long time. To optimize the network bandwidth usage, flow records are not individually exported. Instead, export protocols allow multiple flow records to be packed into a single packet when UDP is used as transport protocol.

Traffic monitoring architectures based on network flow analysis implement the push model: the flow meter continuously sends a stream of flow records to a collector machine, without allowing the collector to read information directly from the flow meter (see Figure 2.2). This is in contrast with polling based monitoring technologies, such as the Simple Network Management Protocol (SNMP) [30].

Historically, flow meters have been embedded into network devices for performance reasons. However, recent advances in microprocessor technologies and in research have favored the development of software based flow meters that are deployed on standard PCs [6] or on specialized network appliances [48] and are fed with packets through mirroring ports or network tap
devices. Software flow meters are attractive because they are extensible and offer higher flexibility than the one built in networking devices.

**Flow records format**

The large majority of modern network equipment, namely routers and switches, commercialized by companies such as Cisco, HP, and Juniper, are able to produce network flow records according to well established standards. In the next paragraphs, we briefly describe the most common flow record standards.

- **NetFlow.** NetFlow, which has been introduced by Cisco, is by far the most widely used flow export protocol. Since its introduction, nine versions of the protocol have been proposed. Currently, the most commonly used are version 5 and 9.

  NetFlow v5 uses a static structure for exporting 19 flow attributes per flow record. NetFlow v5 uses UDP as transport protocol. Each NetFlow v5 packet starts with a header followed by flow records. The structure of each flow record and the header format are depicted in Figure 2.3.

  NetFlow v9 is more flexible than v5 as it allows the definition of custom flow records carrying user defined fields. Additionally, it supports both IPv4
and IPv6 addresses. Netflow v9 uses templates for defining the flow record structure. Flow templates are regularly sent from the flow meter to the collector, which uses that information for decoding flow records. Flow records and flow record templates can be interleaved in the same export packet.

**IPFIX.** NetFlow v9 led to the IPFIX specification [34]. IPFIX extends NetFlow v9 with additional features such as the possibility to use variable length fields and to define Enterprise-specific field types [119]. Due to the higher industry interest on monitoring networks with a service-centric mindset, IPFIX has been chosen as it allows application-specific fields to be exported. Commercial collectors, are nowadays deployed together with advanced flow meters, to export service-specific information and metrics such as: i) voice quality in Voice Over IP (VoIP) networks, ii) the latencies of database operations, and iii) accessed URL [99].

**sFlow.** sFlow [109] is a flow technology embedded within switches and routers based on packet sampling. According to the latest version (ver. 5) can be used for monitoring not only network metrics, but also host and application metrics. sFlow combines interface counters and flow samples into sFlow
Flow record processing

Flow collectors must be able to process high-speed streams of flow records. A typical service provider network exhibits peak flow export rates as high as 50,000 flows per second for each flow meter; this would amount to more than 8 GB of raw flow information per hour. The inter-arrival rate observed by a collector depends on the number of flow meters that use it as the destination, on the export policy at the flow meter side and on the nature of the traffic observed by each flow meter. By exporting flows more frequently, a flow meter can provide finer grained information about the observed traffic, which causes higher loads for the collector. Two approaches can be used for reducing the export rate of a flow meter: i) increase the timeout parameter, and ii) apply statistical flow sampling techniques at the collector side. Sampling techniques are more commonly used when the number of distinct flows is extremely high, as happens in the case of Internet Service Providers (ISPs) networks.

Flow collectors can perform different aggregations of fields contained in the received flow records. By aggregating flows together it is possible to have summaries over time of specific network aspects. For example, one could aggregate flows to report all the exchanged bytes on a specific port or merge together flows generated by a specific Autonomous System (AS). However, by producing network summaries without storing the entire flow stream, important information, such as the IP addresses, will be discarded.

Instead, by archiving the entire stream of flow records, large repositories documenting all end-to-end network communication patterns over time can be created. Storing high-speed data streams of flow records using Relational Database Management Systems (RDBMSs) is not feasible in practice, unless a cluster of many database servers are used to balance the load [79, 87]. Therefore, the de-facto standard way of storing flow records is to dump raw flow packets or eventually flow records on disk using a multiple-file based approach. This approach, adopted by many collectors [1, 6, 65, 105], introduces performance bottlenecks when performing historical analysis on the collected data as raw flow packets have to be decoded for accessing individual flow records. However, for large, multi-gigabyte datasets, a multiple-file based ap-
2.4 Data Stream Warehousing

The prerequisite of stream processing is to know in advance the stream properties to be captured, and, then, to let the data stream flow through an appropriate set of stream processing engines that analyze data streams for extracting the information of interest. Once the raw data streams have been processed by the stream processing engines, they are no longer available for further analysis. Stream buffering techniques can be applied for making a short-term window of the stream accessible from stream engines. The window length that can be kept is limited by the available memory.

However, many network traffic monitoring contexts require access to long-term historical streaming data to enable after-the-fact root cause analyses. By keeping long-term repositories of network traffic data the following applications can be enabled:

- **Network Forensics**, for auditing and compliance purposes. Compressed network flow archives capture digital evidence that can be used for retrospective analysis in cases of information abuse and cyber-security attacks. Such an analysis would require deep recursive exploration of the inbound/outbound traffic through access to the stored archive. Traditional approaches typically require expensive searches and data decompression in large data repositories.

- **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 the laborious retrieval of historical traffic usage and access patterns in one’s domain. **Drill-down** capabilities for querying the archive system are also needed.

- **Behavior Analysis**, with focus on traffic classification. Identifying the application that generated a particular network communication is a challenging task and even modern traffic classifiers are not able to classify 100% of the traffic [84,96]. Therefore, packet traces of the undetected traffic can be stored to enable a posteriori reverse engineering. Recent work suggests that network...
flow information, such as connectivity graphs, the cardinality of flows, packets and bytes exchanged, can be exploited to recognize network applications by studying their behaviour [108].

Ideally, an architecture capable of creating and maintaining large-scale network traffic repositories should provide the following features:

- **High-Throughput** required for storing all the data without losing any information. A system should be able to store the incoming data with no data loss.

- **Volume Reduction** is the ability to reduce the data volumes being persistently collected in order to reduce the capacity requirements of the storage facilities.

- **High Usability** allowing the user to quickly access the data of interest. Once network events have been stored, possibly in a compressed format, they have to be easily and quickly accessible to enable interactive exploration of the collected data.

### 2.4.1 The settings

There are several aspects making network packet repositories substantially different from network flow record repositories. Understanding these aspects is crucial when designing new storage repositories:

**Throughput requirements.** Flow records have been mostly introduced for scalability reasons. In fact, monitoring architectures based on network flow records use flow meters to reduce the data stream arrival rate at the collector side. A stream engine that processes network traffic at the packet level receives up to 1.48 Million packets per second (1.48 Mpps) on a single 1 Gbit link\(^2\). The flow record arrival rate is significantly lower. Commercial flow collectors are able to process up to 100,000 flow records per second (e.g., [8]). Higher flow record rates can be observed when a single collector is processing flow record streams generated by many routers, when monitoring edge routers at large institutions, or during denial of service attacks.

**Compressibility.** Packet trace data is usually hard to compress because encryption and compression technologies are often enabled at different network

\(^2\)Assuming 64 bytes as the minimum packet size.
layers and because most of the digital content exchanged or made available through the Internet is already compressed (e.g. multimedia content). Recent research suggests that the volume of already compressed or encrypted network traffic is going to increase [74, 84]. For this reason, even commercial packet loggers do not have compression enabled by default, as the achieved compression ratio is marginal in most of the cases. In contrast, network flow traces are easily compressible with general purpose compressors, such as gzip.

**Presence of well established storage formats.** The large majority of packet trace repositories store packet traces in a well defined and widely accepted format, which is the \texttt{pcap} format. A rich family of existing applications are capable of processing \texttt{pcap}-encoded packet traces. The definition of a new packet trace format would cause significant legacy compatibility issues. The first issue is that the legacy software is no longer able to read the data encoded with a new format. This issue can be solved by providing proper adapters to \texttt{pcap}. The second and more serious issue has to do with existing packet repositories. To support the additional functionalities (e.g. built-in compression or indexing) when analyzing traces encoded in \texttt{pcap} the existing repositories have to be re-encoded in the new proposed format. This is not acceptable for large-scale historical packet trace repositories. The case of flow record repositories is substantially different. Even if a standard for storing flow traces is highly desired for increasing the interoperability between flow processing tools [120], the industry has not yet agreed on common format for flow record repositories. For this reason, proposing a new way of storing flow traces is acceptable, as the legacy compatibility requirements are not as severe as in the case of packet traces.

**Access patterns.** Network flow record analysis rarely involves accessing the entire set of attributes stored in a single flow record. Additionally, the cardinality of attributes stored within a flow record is small (on the order of 10). Flow records store timestamps that specify when the flow started and ended, but raw flow record streams are not ordered by those timestamps. In contrast, a packet trace is a time ordered sequence of network packets. This has an impact on the packet trace access pattern. The majority of packet processing tools process time ordered sets of packets (e.g., for performing flow monitoring).

In this thesis work, we focus on two key technologies, namely compression and indexing, to enable the creation of large-scale network traffic records
repositories. In the next two sections, we describe how these two technologies are challenged by the network traffic monitoring domain and we highlight issues not yet solved by current approaches and technologies.

2.4.2 Compression

Real-time lossless data stream compression is a challenging research topic. In a network monitoring setting with high incoming stream rates, real-time data compression is a desirable capability because by compressing the data in memory as soon as it arrives, disk bandwidth can be saved. Current approaches to flow record compression rely on general purpose lossless compressors. Compared to lossy compression techniques, which reduce the data by discarding or approximating part of it, lossless compression allows the exact original data to be reconstructed from the compressed representation. The most commonly used general purpose lossless compressors are gzip and bzip2, which provide higher compression ratios than light-weight compressors, such as LZO, but lower compression and decompression throughput.

In a stream-warehouse setting, even more important than real-time compression is to provide high-speed data decompression. In fact, a compressed repository that offers poor decompression speed is not usable in practice as sifting through the data requires time consuming decompression operations. Alternative approaches are required to have similar or better compression ratios than general purpose compressor, while providing substantially higher access speed.

In a network forensic setting, where security experts are more interested in looking for needles in a haystack, the data of interest is a substantially small fraction of the collected data. The ideal compression technology to address this requirement is the one that minimizes the volume of data to be decompressed.

The main shortcoming of many general purpose compressors when applied in a stream warehouse setting is that they are not properly able to capture and exploit data repetitions that occur in large data stream windows. In fact, several lossless compressors, including the large number of commonly used compressors inspired by LZ77 [130], such as ZIP and Gzip, assume that patterns in the input stream occur close to each other.

Since LZ77 principles are used by a large number of compressors, we provide the intuitions behind this algorithm. LZ77 is a lossless, dictionary-based compressor that achieves compression by replacing repeated occurrences of
data with references to a copy already seen earlier in the input stream. A *sliding window*, which is a few thousand bytes long in practical implementations, is used for matching the current data with strings of symbols already seen. References are appended to the (compressed) output stream as compression tokens, which are triplets in the form of (offset, length, next symbol). Intuitively, a triplet \((x, y, z)\) tells the decoder to copy \(y\) bytes starting from \(x\) bytes back. The last element of the triplet, \(z\) is used in case there are no matches in the sliding window. This leads to three types of entities: literals (unmatched bytes), offsets, and lengths.

In what follows, we provide a high level overview of the compression algorithms adopted by widely used compressors: Gzip, Bzip2 and 7-Zip. A complete description of the internal details of the algorithm used in these tools can be found in [107].

- **GZIP** is based on the *Deflate* compression method [45]. Deflate combines LZ77 with Huffman codes [107] to reduce the space used for representing literals, offsets and lengths. According to the deflate standard, the search buffer can be up to 32Kb.

- **Bzip2** offers higher compression ratios than gzip, but lower compression speed. Bzip2 is not based on the LZ77 principles. It compresses data in blocks that can be 100 to 900 Kb in size. The encoder uses a combination of several compression techniques stacked on top of each other (among them Run-Length-Encoding (RLE), Burrows Wheeler Transform (BWT), Move to Front (MTF), Huffman coding) to achieve high compression ratios [107].

- **7-Zip** is known for providing an implementation of the LZMA algorithm, which is a sophisticated variant of LZ77. The principle is similar to Deflate, but range encoding is used instead of Huffman coding. Due to LZMA, 7-Zip offers even better compression ratios than bzip2 at the cost of a lower compression speed.

### 2.4.3 Indexing

There are several issues one has to consider when designing indexing schemes for large-scale repositories.

**Disk space requirements.** Indexes provide alternative data views more suitable for speeding up certain data searches. Therefore, additional disk space is used for the purpose of reducing the time spent in retrieving the desired
data. Increasing the disk footprint has many implications in addition to the increased capacity requirements for the storage subsystem. In fact, bigger indexes pose higher disk bandwidth requirements (for read and write operations) and increase the working set of the operating system block cache, making caching less effective.

**Usability.** In a stream warehousing context, indexes should be as small as possible, while being able to accelerate commonly posed queries. For example, in a forensic context, one has to quickly establish if a specific IP or MAC address appeared during a given time frame. Existence queries must be assisted by indexing mechanisms without requiring the non-relevant data to be accessed.

**Indexed attributes.** Increasing the set of fields to be indexed corresponds to higher disk usage and indexing times, but, on the other side, can speed up a larger set of queries. To minimize the response time given a disk budget the indexing technique must be extensible and allow fields to be indexed at different points in time (e.g. include TCP flags to the set of indexed fields if necessary). A natural consequence of this property is that indexes have to be kept separately, and not embedded within the data itself, in order to avoid the rewriting of the data if the index changes or has to be deleted.

**Efficiency.** If the indexing has to be performed on a live stream, the time required to index a single field must be lower than the inter-arrival time between packets. Several factors influence the insertion performance of an indexing mechanism. The most important ones are: the CPU time, the I/O bandwidth and the I/O operations per second (IOPS). Traditional tree based indexing algorithms have been designed for situations where the data is often updated, and, therefore, the index itself has to be modified in order to accommodate the data changes. These updates are usually the cause of a higher IOPS value. In the context of network traffic indexing, network measurements are written once and never modified. This property must be exploited in order to achieve higher insertion rates.

**Flexible ageing strategies.** For a large scale data repository storing months or even years worth of traffic, disk consumption is an issue. Disk space used for indexing aged data that is no longer accessed must be saved. To save space, indexes for newer data only (e.g. last month of traffic) can be kept. However, re-indexing aged data must be an efficient procedure that only requires the indexed attributes to be accessed.
2.4 Data Stream Warehousing

Given the aspects described above, it is worth describing two techniques that have been used for indexing large-scale network traffic repositories.

**Bitmap Indexing.** A bitmap index is a structure that accelerates search queries. It maps a sequential set of values into positions in a binary map having as many columns as the number $n$ of distinct values that the attribute can assume (the attribute cardinality). Figure 2.4 shows an example of sequential data, corresponding to an attribute having values in the range 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 ($n = 65536$), the required storage space would be a bitmap of $m \times n$ bits. Such a representation gives rise to efficient search operations. For example, to find all records that used a specific source port number $p$ it is sufficient to fetch the bitmap index column corresponding to $p$ and, then, to compute the row positions where the column has a bit set to 1. In addition, bitmap indexes enable efficient logical operations between distinct bitmap indexes. As an example, finding all records that used a particular source- and destination-port can be obtained by performing a bitwise operation between the columns of two respective bitmap indexes. Bitwise operations can be efficiently implemented as bitwise instructions.

A shortcoming of bitmap indexes is that they may require large storage space especially when high cardinality attributes have to be indexed. This give rise to compressed bitmap indexes. The idea is to compress bitmaps column-wise using CPU-efficient compression techniques, such as Run-Len-
Encoding (RLE). As shown in Figure 2.4, RLE replaces omogeneous sequences of bit values (runs of 1’s or 0’s) with a 1 or 0 associated with a count. In this manner, the disk footprint can be substantially reduced. In addition, column-wise compression is beneficial for accelerating bitwise operations between distinct bitmap index columns. In fact, by compressing sequences of homogeneous symbols, the number of bitwise instructions required to implement logical operations can be reduced (e.g., when performing an AND between a column $c_1$ and a column $c_2$ storing 0s only). Modern compressed bitmap indexing encodings are designed to support boolean operations directly, and more efficiently, in the compressed domain [127].

In general, compressed bitmap indexes are expected to be more compact than typical B-Tree structures even for high-cardinality (large $n$) attributes [125]. WAH [126] and PLWAH [39] represent the current state-of-the-art encodings for compressing bitmap indexes.

- **Probabilistic indexing.** When designing an indexing structure for large scale repositories one could consider the option of having a probabilistic index, such as the ones based on Bloom filters [22], which may provide false positives (i.e. the index signals the existence of a record matching the rule even if it does not match) but guarantee that there are no false negatives. The reason why a probabilistic index may be preferred to a lossless index is the storage space requirement. In fact, an index based on Bloom filters may provide a small percentage of false positives and still be an order of magnitude smaller than deterministic indexes. Bloom filters can be used for indexing data blocks, however, contrary to bitmap indexes, they do not provide the capability of indicating the exact position where the data of interest is located within the data block.
Chapter 3

Related work

This research work is interdisciplinary as it combines concepts from the database and networking domains. In this chapter, we review technologies that relate to our work and we highlight, when possible, the main differences with our approaches.

3.1 Data management systems

Data Warehouses

Columnar physical layout of the data [17, 66, 68] has been extensively used for reducing response times and increasing the query throughput of read optimized databases. Organizing the data in columns rather than in rows offers several advantages: i) it provides more opportunities for compression (e.g. compressors optimized for specific data types can be used); 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. CStore [116] and MonetDB [23] are high-performance general purpose columnar databases proposed in the literature. Many commercial data warehouses use columnar approaches [2, 29].

Columnar databases have been used to store network measurements. NetStore [61] is a columnar database optimized for storing flow records with built-in support for compression. Each data column is horizontally partitioned into data blocks, which are individually compressed using one of the sup-
ported compression algorithms. The compression algorithm and the block sizes are decided to favour the access time rather than the maximum sustainable insertion rates. The main design goal of NetStore is to be able to sustain a target insertion rate, which is fixed to 10,000 flow record per second in the paper [61], while creating read-optimized repositories. However, the peak flow record arrival rate that can be reached in a given network is not easy to predict, as it can drastically change, for example, during Distributed Denial of Service (DDoS) attacks.

**Data Stream Management Systems**

Data Stream Management Systems (DSMS), like Gigascope [35], TelegraphCQ [31], Borealis [16], Tribeca [117], and System S [128], compute pre-determined queries on streams on-the-fly and store only the output of the queries. DSMSs usually provide SQL-like languages augmented with stream-aware operators for decreasing the development time of stream processing engines. The feasibility of using such systems for network traffic analysis tasks has been demonstrated in previous studies [98]. However, DSMSs provide low utility for custom post-mortem analysis of network traffic data as only the query results (e.g. data aggregations) are kept on secondary storage.

Distributed stream processing frameworks have been implemented for introducing real-time processing of web data [63]. Major web players developed their own stream processing framework to complement their big data infrastructures with real-time processing capabilities. Among them are Yahoo [90], Google [95], Facebook, Linkedin, Twitter and Cloudera. However, those systems are designed for facilitating the implementation of web analytics applications and offers lower performance than general purpose stream processing systems [37].

**Stream Data Warehouses**

Stream Data Warehouses (SDW) have been introduced to bridge the gap between Data Warehouses and Data Stream Management Systems, and are becoming an active research topic [114]. One of the major challenges that Stream Data Warehouses have to address is the implementation of highly efficient and continuous database updates [62, 70, 100, 118]. Even if several initiatives have emerged that recognize the need to support lossless storage of data streams [26, 46], real-time compression techniques optimized for streaming contexts have not been widely studied. In fact, these efforts focus primar-
ily on general frameworks for stream storage, and searching on the archive is based on generic pattern matching paradigms. Real-time data compression or even search in the compressed data are not covered by these research works.

3.2 Traffic recording systems

Long term historical analysis of massive volumes of network management data is an emerging requirement for increasing reliability, security and performance of modern networks [71]. The Time Machine [85] and Hyperion [44] are packet-to-disk solutions designed for storing packets captured from high-speed links. Both architectures save packets in large file segments and have identified indexing as a mandatory feature for exploring massive packet repositories. Indexes are only 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.

Packet-to-disk appliances are in the product portfolio of companies specialized in high-speed traffic monitoring and cybersecurity, such as Riverbed [103], Endace [48] and Niksun [91]. Those products provide high-speed traffic capture and replay and mechanisms for filtering the traffic to be captured from network links, but do not provide mechanisms for enabling interactive queries over the archived data.

3.3 Network Flow collectors

*Flow aggregation databases*, such as as AURORA [1] and nTop [89], use stream-based approaches to analyze high-speed streams of network flow records. They 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 efficient drill-down functionalities, which is the scope of this research.

*Network Traffic Databases* attempt to lift some of the limitations of stream-based systems by archiving entire flow record streams. Silk [59], Nf-Dump [65] and Flow-tools [105] are 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 [92]. Important shortcomings of those software suites are that: i) they do not provide indexing facilities, and ii) the entire dataset must be scanned linearly (and decompressed) even for retrieving only a few records.

Network traffic databases still represent a better alternative to Relational Database Management Systems (RDBMSs) for storing and querying large amounts of network flow records. Hofstede et al. [67] compare NfDump with MySQL, a very popular RDBMS, when performing queries over multi-gigabyte flow record datasets. Their benchmarks show that NfDump always outperforms MySQL, and that MySQL indexes are not effective in reducing the response times when the dataset largely exceeds the available memory.

Reiss et al. [102] propose a flow storage solution implemented on top of FastBit [124], which is a library for developing column-oriented databases that provides built-in support for compressed indexes to accelerate queries over historical data. The proposed 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 off-line and in batch.

3.4 Cloud-based network traces analysis

The MapReduce programming model (MR) has been largely used by Google for simplifying the development of data intensive applications running on large clusters of commodity servers [38]. The Hadoop project [12] created an open-source infrastructure for processing data at scale using MapReduce, that includes a fault-tolerant distributed filesystem, HDFS [111], with built-in support for data replication. Hadoop quickly became the industry standard for processing large volumes of data with the MapReduce paradigm.

Hadoop has also been adopted in the context of large scale network trace analysis. In particular, Lee et al. show that Hadoop can be used for analyzing NetFlow traces [77]. As Hadoop is mostly text-oriented, flow records are converted to text records for simplifying the data processing. The work does not address the problem of data volume reduction and high-speed real-time ingestion of flow records. In a more recent work [76], they focus on packet trace processing. They implemented Hadoop adapters that allow packet traces to be processed directly in pcap format (i.e. without requiring an impractical pcap to text conversion).
3.5 Compression

Distributed architectures for storing flow records, such as MIND [80] and DIPStorage [87] are based on a federation of database peers. By assigning flow records to different peers the maximum sustainable insertion rate can be increased and the query response time reduced.

In this research, we do not take into account distributed architectures to enable packet and flow record searches.

3.5 Compression

In this section, we describe relevant work in the field of time series compression and high-speed lossless compressors. We conclude the section by positioning our contributions against recent trends in compression research.

Time series compression

A time series is a series is a sequence of data points measured at uniform time intervals. Lossy compression has been used for reducing the volume of time series coming from the network monitoring context. Round Robin Database (RRD) [94], the most widely used open-source time series database used in the industry, implements lossy compression by means of aggregations. More advanced lossy compression techniques have been proposed in the literature for compressing time series while minimizing (or providing guarantees on) the reconstruction error and for allowing the computation of certain operations (e.g. correlation) directly on the compressed representation [88, 101, 110, 121]. However, as pointed out in [81] many of these techniques are not suitable when time series coming from monitoring contexts, such as performance monitoring, are dominated by spikes and high jitter. In those cases, a lossless compression is preferred. In this research, we do not focus on lossy compression, as our goal is to store exact streams for the purpose of historical post-mortem analysis particularly in the case of forensics and auditing.

OpenTSDB [9] and Tsdb [43] are time series databases that store multiple time series with a lossless compression approach. OpenTSDB uses a distributed architecture, based on the Hadoop infrastructure, and has been designed for storing metrics continuously collected from nodes in large cloud environments. Tsdb, instead, has been designed to enable the collection of a large number of distinct time series on a single node.
High-speed lossless compression

Advances in commodity processor technologies and computer architectures, such as multi-core processing, introduced new opportunities for increasing system throughput by applying lossless compression techniques. Compressors optimized for compression/decompression speed rather than for achieving high compression ratios have been developed and proficiently applied for boosting the performance of many data processing systems including databases, information retrieval systems and message passing frameworks.

The Lempel-Ziv (LZ) [130] family of lossless compression algorithms uses a dictionary to encode a window of data replacing phrases with pointers within the window. The Lempel-Ziv-Oberhumer (LZO) compressor [92] is the fastest member of the family standing out for its very fast compression and even faster decompression speed [123,129]. For this reasons, LZO-compression is used in many contexts where real-time compression and high-speed decompression is required. In fact, LZO is the preferred compressor for the Hadoop infrastructure and can also enabled in NfDump. In this thesis work, we apply LZO compression in the context of flow record compression and we describe a stream-based flow record reordering technique that further boost its performance. Additionally, we describe a novel compressor optimized for flow record, RasterZip, that offers the same compression speed as LZO, but outperforms it in terms of compression ratio and decompression speed.

Compression algorithms have been extensively used to boost the performance of columnar databases [32,116,131]. Abadi et al. [15] compared the performance of different compression algorithms for columnar databases and provided guidelines for a database designer on how to select the appropriate compression algorithm.

Recent research trends in compression

The algorithms described in this dissertation are in line with active research topics in compression. We conclude the section with a brief description of recent research trends in compression to position our work against them.

- Modern hardware exploitation. High-speed compression algorithms optimized for modern hardware, such as superscalar CPUs or Graphics Processors...
3.6 Indexing

Compressed bitmap indexing technologies have been used for indexing network flow records. Bethel et al. [21] adopted Word-Aligned-Hybrid (WAH) [126] compressed bitmap indexes to reduce duty cycles of queries in the context of network traffic visualization. Deri et al. [42] rely on the compressed bitmap indexing implementation of WAH provided by Fastbit to
store, and query, flow record attributes. Compared to probabilistic indexes such as the Bloom filters used in [104], compressed bitmap indexes enable the efficient implementation of existence queries. In fact, contrary to what happens in the case of Bloom filters, where the possibility of having false positives requires expensive data scans, compressed bitmap indexes allow existence queries to be answered without requiring the data to be accessed. An additional property that makes compressed bitmap indexing a suitable technology for indexing network traffic records is the storage space requirement. In fact, compressed bitmap indexes, such as Word-Aligned-Hybrid (WAH), Run-Length Huffman (RLH) [115] or Byte-Aligned Bitmap Codes (BBC) [19], have been shown to offer indexes that are many times smaller than traditional tree-based indexing data structures [125].

Several studies [20, 73, 78, 97] have shown that sorting techniques can be used to increase the average length of run-length-encoded sequences and thus to improve the compression rate offered by compressed bitmap indexes. Contrary to previous works, we do not focus on finding an optimal sorting strategy. Instead, we propose a sorting strategy that can be performed on-line using a stream based approach. The sorting strategy we propose is of general interest and can be used in similar contexts where high-speed streams of multi-attribute numerical records are indexed using compressed bitmap indexing technologies.

Despite the growing industry interest in high-speed packet logging, packet trace analysis libraries, including the most commonly used, libpcap [10], do not provide any indexing functionality. In particular, the BPF packet filtering mechanism [86], which has been originally designed for filtering packets from streams, has also been (ab)used for searching packets within packet traces. Libtrace [4] allows to retrieve packets belonging to a specific time range. It uses an array of tuples (timestamp, offset) to position the reading cursor at the beginning of the specified time range, but it does not leverage an index for speeding up the process.
Chapter 4

NET-FLi: indexing and compression of streaming flow records

In this chapter we describe 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 based on the synergy of bitmap indexing and Locality Sensitive Hashing (LSH) [112] technologies. Our approach 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 effective indexing and selective data-block decompression strategies.

Our solution has been primarily created to support archiving of network flow data, and to enable applications for forensics and network troubleshooting. However, the indexing and compression strategies that we present can also be used for archiving and searching over any stream of multi-attribute, numerical records, when high throughput and low latency are essential.

Our work makes several important contributions:

- We present a novel solution for on-the-fly 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 that further boosts the space savings of our approach.

- The compressed columnar approach that we propose achieves compression ratios on par with that of a typical GZIP compression, with the added benefit of providing indexing functionality as well as 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 those required to capture all flows of typical networks.

- The combination of compressed bitmap indexes and compressed data columns enables the selective decompression of the desired data, and therefore guarantees interactive retrieval times while searching through gigabytes of compressed flow repositories.

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

This chapter is structured as follows. Section 4.1 presents the archival and bitmap indexing approach of our solution. Section 4.2 describes a compressed bitmap index encoding optimized for network data. Section 4.3 discusses the online stream reordering methodology. We evaluate our approach in Section 4.4 with respect to other existing approaches. Section 4.5 provides a case study and introduces the visualization layer of our solution. We conclude the chapter with Section 4.6, which summarizes the achievements and analyzes issues that are the motivations for additional work.

## 4.1 Architecture

In this section, we describe our approach and explain our design choices. An overview of our solution is depicted in Figure 4.1. Our solution comprises the following parts:

1. A data pre-processing phase reorder the incoming flow record stream in order to achieve better locality and boost compression (Figure 4.1B). This step is optional.
2. Separation of the flow record attributes in a columnar manner (Figure 4.1C).

3. An archiving backend that compresses blocks of the incoming flow data (Figure 4.1D).

4. A compressed index encoding the flow information into the novel COMPressed Adaptive indeX format (or COMPAX) (Figure 4.1E).

**Figure 4.1:** NET-FLi consists of the following steps: A) Streaming multi-attribute data are input in the system B) Optionally, a fast packet reordering based on Locality-Sensitive-Hashing principles is performed to improve both compression and retrieval. C) Attributes from the streaming records are separated. D) The compressed columnar archives are created. E) Compressed bitmap indexes are created on-the-fly, in parallel.
The above tasks are executed on the fly and in parallel, which imparts the high performance to our solution. The optional online stream reordering step, described in detail in Section 4.3, modifies the processing order of streaming flow records in order to achieve significantly better compression ratios for both archived data and the data index. This approximate sorting, based on an online implementation of a Locality Sensitive Hashing based technique (online LSH or oLSH), is computationally light, therefore enabling an on-the-fly execution over the streaming network data. Although this optional process slightly reduces the number of processed flows, it results in reduced archive and index storage, and, as shown in our experiments, leads to significantly better query response times.

Finally, a component outside the archival solution is the query processor. Given a search query and using the indexes created, the query processor identifies historical flows that match the criteria given and retrieves them by uncompressing only the relevant portions of the archive data.

Below we provide more details on each of the components. We first cover the archival and indexing solution and later the stream reordering methodology.

### 4.1.1 Compressed Archive of Flow Data

The input to our system consists of streaming network flow records. As mentioned, flow records can record a number of predefined communication attributes; in our system we utilize 12 such attributes (cf. Figure 4.2).

![Figure 4.2: The flow record we use in our setting.](image)

The incoming flow records are packetized and processed using a data window of $M$ flows (Figure 4.1C). Even though windowing the data reduces the effective compression rate, this allows us to provide selective decompression of archive blocks, hence significantly speeding up the query execution time.
Practical considerations at this stage are the size of the data window. Larger windows will offer a better compression ratios, but less selective data retrieval during search time. In an effort to reduce cache misses, we use $M = 4000$ records in our implementation; this 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 and realized that changing the block size does not significantly affect the compression ratio. The overall query response time of the system was best when using blocks of 4000 records, because this provides a finer-grained access on the archived data. In addition, fewer data have to be decompressed at query time.

For all subsequent phases, the data are treated in a columnar manner, and each attribute (column) of the flow records is processed independently. Each of the blocks of conceptual data columns is compressed using a specific compression scheme. Compression algorithms for data columns are interchangeable. We choose the Lempel-Ziv-Oberhumer (LZO) compressor [92] as our default compression algorithm, because it provides a nice compromise between compression speed and compression ratio. In particular, it has been empirically demonstrated that LZO is four to five times faster in decompression than zlib [58], even when using zlib at the fastest compression level [93].

To be able to exploit sequential writes to the disk, the compressed block of streaming data are not appended to the data archive as soon as they are created. Instead, the compressed columnar blocks are initially buffered and only flushed to disk when the buffer is full \(^1\) (Figure 4.1D).

### 4.1.2 Compressed Index of Flow Record Attributes

Concurrently with the creation of the compressed flow data archive, compressed bitmap indexes are constructed. The creation process is performed on-the-fly and in parallel. During the querying process, bitmap indexes facilitate the rapid localization of the data of interest from the compressed archive.

We begin by elucidating the creation and usefulness of traditional bitmap indexes. Then we show how we index network attribute fields.

### Bitmap Indexes

The concept of bitmap indexing has been applied with great success in the areas of databases [124] and information retrieval [54]. The basic notion is

\(^1\)In our implementation we allocate 4MB for each column buffer
to keep $k$ bitmap vectors (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 $i$ corresponding to the inserted value and “0” otherwise. An example of a bitmap index with $k = 8$ is illustrated in Figure 4.3. In addition to fast updates, bitmap indexes allow inexpensive bitwise AND/OR operations between columns. For example, given the bitmap index of Figure 4.3, to retrieve the row numbers where the record was either 5 or 7 it is sufficient to perform the bitwise OR between columns 5 and 7. The desirable rows are indicated by the positions where the resulting bitmap is 1. Another key property of bitmap indexes is that subsequent insertions do not require reorganization of the existing index, as new bit values can be appended easily to the end of the index.

Compressed variants of bitmap indexes have appeared in the literature [125], with the most popular variant being the World-Aligned-Hybrid (WAH) [126] which has been used for indexing and searching on a multitude of datasets, including scientific simulations results and network flow repositories. WAH uses Run-Length-Encoding principles to compress homogeneous sequences of sets of the same symbol. Compression is not only beneficial for reducing space, but also for improving the performance of boolean operations. Logical bitmap operations can be executed directly on compressed bitmaps by using run-length-encoded sequences to decrease the number of bitwise comparisons. More recently, the Position List Word Aligned Hybrid (PLWAH) [39] has been introduced. It represents a variant of WAH offering better compression performance for uniformly distributed data.

The architecture provides a modular indexing infrastructure where diverse
compressed bitmap indexing algorithms can be plugged in. We compress each bitmap index in a columnar manner on the fly. In our implementation, we support the two recent state-of-the-art bitmap indexing technologies, WAH and PLWAH. In addition, we also design and implemented our compressed bitmap indexing variant, called COMPAX, which we describe in Section 4.2. In the experimental section, we compare COMPAX against both WAH and PLWAH, and we show that COMPAX provides better performance than the state of art when indexing network flow record attributes.

Indexed Attributes

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. A particular indexing strategy is taken for IPv4 addresses: a separate index is maintained for each 8-bit block of a 32-bit IP address. In this way we accelerate wildcard queries over networks (e.g., 10.1.*.*) by combining compressed bitmaps belonging to different byte indexes using boolean AND operations.

4.1.3 Querying the System

The proposed architecture can very efficiently answer common types of queries posed by system administrators and security analysts. The system users can use 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, which are also captured in Figure 4.4:

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 data blocks) are decompressed, and the results are supplied to the user.

---

2The generalization to IPv6 addresses is straightforward.
Figure 4.4: Example of a query execution.
We will explain the above process with a sample query that a network administrator can issue:

**Query:** “Find all destination IP addresses contacted by the source IP address range 10.4.*.* at destination port 22.”

The various substeps are depicted in Figure 4.4. As bitmap index files are created on an hourly basis, first the bitmap 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 and 4, and 22, for $SrcIP.byte1$, $SrcIP.byte2$ and $DstPort$, respectively, are fetched. Note that we do not need to uncompress the columns; an AND operation can be performed directly on the three compressed columns, 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 $\{200, 10010, 10500\}$. 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 $DstIP$. Assume that each of the compressed blocks on the archive contains 4000 flow records. Therefore, to access the $200^{th}$, $10010^{th}$ and $10500^{th}$ 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.

### 4.2 COMPressed Adaptive indeX - COMPAX

In this section we describe a new bitmap index “COMPressed Adaptive indeX” or COMPAX. It is based on a codebook of words that significantly reduce the bitmap index size when applied to the context of network flow record indexing.

#### 4.2.1 Encoding

We compress each bitmap index in a column manner on the fly using our COMPressed Adaptive indeX (COMPAX). There are two variants of COMPAX: one-sided (or COMPAX) and two-sided (or COMPAX2). We adapt this
Figure 4.5: Example encoding of a row bitmap vector (top) corresponding to a single bitmap index column (as in Figure 4.3) using the WAH (middle) and the one-sided COMPAX method (bottom). The one-sided COMPAX codebook is shown at the bottom right.
terminology from [127], in which BBC bitmap index encoding offers a two-sided variant that compresses both sequences of zeros and ones, whereas the one-sided variant focuses on sequences of zeros only. COMPAX2 offers an extended codebook, which allows better compression but introduces more complex bitwise operations. More details will be provided below.

COMPAX uses a codebook of four word types and compresses sequences of zeros with run-length encoding. 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 31-bit-long non-zero sequence of bits; it is encoded as a 32-bit word having the first bit as one (1), e.g.,

   \[
   \begin{array}{c}
   \hline
   1 & 0011100 & 1100010 & 00110000 & 00000000 \\
   \hline
   \end{array}
   \]

2. A 0-Fill [0F] encodes a sequence of consecutive chunks of zero bits only by means of run-length encoding. For example, a portion of a bitmap index column consisting of a sequence of \(3 \times 31\) zero bits is encoded within a single 32-bit word as

   \[
   \begin{array}{c}
   \hline
   000 & 00000 & 0000000 & 00000000 & 0000011 \\
   \hline
   \end{array}
   \]

   where the first 3 bits encode the codeword type (0-Fill) and the remaining 29 payload bits encode the number of 31-bit chunks from the original sequence.

3. An [LFL] word condenses sequences of [L]-[0F]-[L] words after applying null suppression. In particular, if in each of the two literal words in the sequence only one of the payload bytes is non-zero (“dirty”) and if the 0F represents a sequence shorter than \(256 \times 31\) bits, the 3 dirty bytes are packed into a single [LFL] word. An example of how the [LFL] codeword is formed is shown in Figure 4.6. The 8-bit header encodes the word type (3 bits) and the positions from 0 to 3 of the literals’ dirty bytes (4 bits in total). The last bit is unused.

   A real example of [LFL] encoding is depicted in Figure 4.5. The raw uncompressed bitmap vector depicted on top consists of five 31 bits chunks, where only the first and the last one contain some non-zero bits. As mentioned, they can be represented as literals [L]. As for both literals the non-zero bits are present in only one of the 4 bytes, they exactly have one “dirty” byte (in position 3, and 1 for the first and
Figure 4.6: Example of bit packetization in the [LFL] codeword.

the last, respectively). The three remaining chunks do not contain any non-zero bit and can be encoded with a single [OF] word, where the payload encodes the number of 31 bit chunks (3 in the example, which can be represented as a single byte). Therefore, the technique depicted in Figure 4.6 can be applied to encode the two literals and the FILL as a single [LFL] word. Its one-byte header specifies the word type and, in addition, encodes the positions (3 and 1, respectively).

4. The [FLF] word condenses a sequence of words following a [0F]-[L]-[0F] paradigm and having a single non-zero byte. As fill words are representing sequences shorter than $256 \times 31$ bits, the header includes the position of the literal non-zero byte only (2 bits).

- COMPAX2 extends the COMPAX codebook with a 1-Fill (1F) word that compresses sequences of ones. In addition, the FLF and LFL headers include bits specifying the fill type (0F or 1F):

1. The 1-Fill [1F] encodes a sequence of consecutive chunks of one bits by means of run-length encoding.
   For example, a portion of a bitmap index column consisting of a sequence of $3 \times 31$ one bits is encoded within a single 32-bit word as:
   
   \[
   \begin{array}{c}
   011 \\
   00000 0000000 0000000 0000011 \\
   \end{array}
   \]

2. The [LFL] word is built as in COMPAX. However, both sequences of [L]-[0F]-[L] and [L]-[1F]-[L] can be compressed. The word carries an 8-bit header and the three payload bytes. The first three bits of the header encode the word type (001 for LFL), whereas the remaining
5 bits encode the two positions (2 bits each) and the fill type (1 bit representing 1F or 0F). For example, a sequence such as the one in Figure 4.6, but with a 1F word instead of a 0F, is encoded as

```
001 01101 00000000 00000000 00000011
```

3. The [FLF] compresses sequences following the [F]-[L]-[F] pattern. Contrary to COMPAX, fill words can be both 0F or 1F. The 8-bit header encodes the word type (3 bits), the two fill types (2 bits) and the position of the non-zero byte (2 bits) within the literal. The last bit of the header is unused.

```
010 11000 00000000 00000000 00000011
```

### 4.2.2 Comparison with other bitmap indexing schemes

In this section, we briefly outline the main differences between COMPAX and the state-of-the-art bitmap indexing approaches.

**Comparison with WAH:** The WAH encoding uses three word types to encode bitmaps: Literal [L], 0-Fill [OF] and 1-Fill [1F]. In our work, we introduce the [LFL] and [FLF] codewords because we observed that such patterns were predominant in network flow traffic. The one-sided version of COMPAX omits the [1-Fill] word from the codebook, because such patterns are quite uncommon in network traffic data. In addition, it provides a simpler implementation of bitwise operations because of the reduced codebook size. However, in both cases, our coding scheme leads to significant space savings. In the experimental section, we show that compared to WAH, even the simplest COMPAX encoding can result in a space reduction of more than 60%, particularly when combined with the optional tuple reordering phase (described in detail later on).

Figure 4.5 depicts the difference of the one-sided COMPAX encoding compared with WAH. The bits of a column are shown row-wise for presentation reasons and the original uncompressed sequence comprises 155 bits (indicated as 5 verbatim 31-bit words [V]). In this example, we illustrate COMPAX’s ability to condense three WAH words into one. In general, the COMPAX encoding packs more information because of the carefully selected codebook and as such offers superior compression.
**Comparison with PLWAH:** Similarly to COMPAX, the PLWAH encoding scheme attempts to achieve better compression rates than WAH in the presence of very sparse bitmaps. In the original paper [39], the authors of PLWAH show empirically and analytically the improved compression rates when encoding data that follow a uniform distribution. PLWAH boosts the compression rates by exploiting the fact that in WAH, 0-Fill and 1-Fill words do not usually represent long sequences and literals themselves contain only few bits set to 1. Therefore, by storing the positions of those non-zero bits into the preceding 0 or 1-Fill word, it is possible to pack a FILL word and the following sparse literal into a single codeword. When the word size is 32 bits, each position can be encoded with 5 bits (0 to 30). The number of positions that can be stored in the preceding FILL is a tunable parameter, and, in fact, increasing the number of positions kept corresponds to decreasing the maximum length of the sequence of homogeneous symbols encoded by the FILL word. By keeping a single position, the longest sequence of homogeneous symbols that PLWAH can represent is $(2^{25} - 1) \times 31$ instead of $(2^{30} - 1) \times 31$ as in WAH. In the example of Figure 4.5, a PLWAH implementation that keeps a single position can encode the row bitmap vector in two words instead of three as WAH does. In fact, the last literal containing a single bit set to 1 and the preceding FILL word can be merged by storing the position (9 in the example of Figure 4.5 when counting from the least significant bit) as a 5-bit number just after the two bits header of the FILL word.

**Implementation of boolean operations in the compressed domain:** WAH, PLWAH and COMPAX enable the implementation of the boolean operations in the compressed domain. In practice, operating in the compressed domain means that: i) the result of a logical operation $op$ (e.g., AND/OR) between two distinct bitmap index columns, $B^X_i$ and $B^X_j$, compressed with an encoding $X$ and corresponding to the attribute value $i$ and $j$, is a new column $R^X_{ij} = B^X_i \ op \ B^X_j$ compressed with the encoding $X$, and, ii) computing $R^X_{ij}$ does not require explicit decompression of $B^X_i$ or $B^X_j$.

We first provide an overview on how boolean operations can be implemented in WAH in the compressed domain, and, then, we show that COMPAX and PLWAH can use similar approaches. WAH is word-aligned to efficiently exploit the bitwise instructions provided by microprocessors. The benefits of word-alignment are clear when performing AND/OR logical operations between WAH-incompressible bitmap index columns, which are columns made of literal words [L] only. Each literal word stores a chunk of 31 bits and a single bit header that indicates the symbol type (L). In this case,
the AND/OR between two literal-only columns $B_j$ and $B_i$, can be done by performing the bitwise AND/OR between the two columns, word by word.

[OF] and [1F] words allow the number of bitwise operations to be reduced. Let suppose, for example, to perform the AND between the WAH-encoded bitmap column shown in Figure 4.5 and a literal-only column, made of 5 literals. In this case, WAH can leverage the [OF] word to skip three AND operations as the [OF] word compresses three 31-bit chunks containing 0s only. In fact, to build the resulting column, only two pair of WAH words have to be ANDed: the two literal words storing the first 31-bit chunk in each column, and the two literal words storing the last 31-bit chunk. If they contain at least a bit set to one\(^3\), the resulting column consists of three words: the AND between the two literal words storing the first 31-bit chunk in each column, the unmodified [OF], and, the AND of the literal words storing the last 31-bit chunk. However, the results of the AND operations between literal words have to be taken into account when creating the resulting compressed column. In fact, in case they do not contain any bit set to 1, except the one corresponding to the header, they can be removed, and the length stored in the adjacent [OF] word increased accordingly. By processing the columns from the beginning (starting from the left in Figure 4.5) the resulting column can be built, incrementally, by appending, merging, or splitting, WAH words. A comprehensive description on how to implement boolean operations in the compressed domain is provided by the authors of WAH [127].

In the case of COMPAX and PLWAH, boolean operations can be implemented using similar approaches. However, the more complex encodings require additional bitwise instructions to extract the literal(s) from a Fill word in case of PLWAH, or from a [FLF] or [LFL] word, in the case of COMPAX. In addition, for both encodings, the merging operations are slightly more complex. For example, in PLWAH, appending a new Literal word [L] requires to check if it is possible to store it in the previous Fill word. In COMPAX, up to two words have to be checked to determine if an LFL word can be created with the current literal. However, as we show in the experimental section, a more complex encoding that provides better compression ratios, does not necessarily reduce the performance of boolean operations. In fact, higher compression ratios correspond to higher memory locality and better cache utilization. In the experimental section, we show that by compressing the data better than WAH, the COMPAX encoding provides higher lookup performance than the WAH encoding.

\(^3\)not considering the single bit header
4.2.3 On-the-fly Bitmap Index Creation

The COMPAX encoded bitmap indexes are created on the fly to minimize the space consumption, without having any adverse impact on the index size (see also Figure 4.1E).

For example, assume that already a 0-Fill codeword has been created on some column, encoding $3 \times 31$ bits of zeros, as in the previous example. Now, suppose that for the same column one more chunk of 31 zeros is observed, then another 0-Fill word of length 1 can be created. By looking at the previously produced codeword, we are able to merge them and produce a single 0-Fill word of length 4, which in binary format will be encoded as

```
000 00000 0000000 0000000 00000100
```

In a similar manner, we maintain the codewords per column on the fly. 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 literal, so another literal [L] word is created.

- [LFL]: In order to form an [LFL] codeword a look-back examines the two previously produced codewords. If they are [L] and [F] and the current word is a literal [L] and all three codewords have only one dirty byte (the remaining bytes are all zero), then these three codewords can be merged into a single [LFL] word.

- [FLF]: Treated similarly as [LFL].

The advantage of encoding the bitmaps on the fly is the memory consumption, which, during the entire creation phase, is almost the same as the compressed bitmap index size. An example of this encoding is shown at the bottom right of Figure 4.1.

**Bitmap Index Serialization:** The COMPAX-encoded bitmap indexes are serialized to disk by sequentially appending the compressed columns. In addition, each index is prepended with a header that contains offsets corresponding to the beginning of each compressed column. In this way, random access to specific compressed column within a bitmap index is accommodated. Headers are also compressed. In fact, offsets are compressed using gap coding and the Simple9 algorithm [18], which is among the fastest available integer compression/decompression schemes.
4.3 Online-LSH (oLSH) Stream Reordering

In this section, we introduce an online stream reordering mechanism based on the principle of Locality Sensitive Hashing (LSH) [60]. We call this variant online-LSH or oLSH for simplicity. It has been shown that data sorting leads to smaller and faster bitmap indexes [78, 97]. FastBit [124], the reference implementation of the WAH algorithm, accommodates an optional off-line column sorting to decrease index sizes. In this research, we reorder streams of multi-attribute numerical records (i.e. the flow records) in real-time with the purpose of decreasing the disk consumption of both indexes and data archives. Intuitively, since we compress (and index) each attribute separately in a columnar manner, we are interested sorting the flow records in such a way that columns store homogeneous sequences of values.

oLSH is a stream processing technology that, given a continuous stream of flow records, produces a data stream carrying the same flow records, ordered in a way that records with homogeneous attribute values tend to be in adjacent positions. oLSH implements an intelligent hash-based buffering mechanism that uses several LSH-based hash functions in order to group flow records by content. The sorted output stream is then built by flushing sets of homogeneous records from the hash table (cf. Figure 4.7).

Our approach is consonant with the permuting, partitioning, and compression (PPC) principle [51]: after permuting the order of incoming stream records, records are partitioned into groups of “similar” records with the goal of boosting the compression ratio of general-purpose compressors.

The characteristics and benefits of oLSH are as follows:

- It reorders the incoming data records in an efficient and effective way, resulting in data blocks with lower entropy.

- 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 better query response times, because fewer data blocks need to be decompressed from the data archive.

Locality Sensitive Hashing is a technique for dimension reduction of high-dimensional data. The goal of LSH is to use hash functions to direct vectors that are close to each other according to a distance function into the
same hash bucket with high probability. This property is in contrast with traditional hashing that has the goal of separating inputs that are close together, but not equal, into distinct buckets. This also means that unlike traditional hashing, which has the purpose of accelerating exact matches, locality sensitive hashing enables approximate matching and, therefore, has its natural applications in nearest neighbors search problems [82].

In this research, we rely on LSH to implement an effective real-time approximate clustering. In particular, use LSH functions based on $p$-stable distributions, such as the ones proposed by Datar et al. [36]. Each LSH hash function $h_{\vec{a},b} : \mathbb{N}^d \to \mathbb{Z}$ maps a vector $\vec{r} \in \mathbb{N}^d$ into a single value (“bin”) and is defined as

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

where $\vec{a}$ is a $d$-dimensional random vector with each component chosen independently from a Gaussian distribution\(^4\), $W$ is the width of a bin, and $b$ is a

\(^4\)The Gaussian distribution is 2-stable; it can be shown that elements, which are close in the
real number chosen uniformly from the range $[0, W]$. The intuition behind this LSH method is as follows. If two points are close in a multi-dimensional space, then their projections into a sub-space will most likely remain close together. On the contrary, if two points are far apart, their projections will be far apart in most of the cases. The distance between two points can be indirectly estimated by looking at a set of random projections. If many projections are similar, then the original two points are close together with high probability. Therefore, at the core of this method is the scalar projection (dot product) between a random vector $\vec{a}$ and the input vector $\vec{r}$. A comprehensive and yet practical explanation of LSH functions based on $p$-stable distributions is provided by Slaney et. al. [112].

In our setting, we consider each flow record as a vector $\vec{r} \in \mathbb{N}^d$, where $d$ is the number of the attributes relevant to the sorting process. The purpose of the LSH functions is to direct “similar” flows to the same hash bucket, which will eventually contain a chain of “similar” records. In our scenario, we wish to pack together flow records based on the attributes most often queried. Therefore, we chose $\vec{r}$ to be an 11-dimensional vector ($d = 11$) consisting of the attributes source and destination IP addresses ($2 \times 4$ bytes, i.e., 8 dimensions) as well as source and destination ports and protocol numbers (3 dimensions).

By clustering flows according to this attributes, we also observe that attributes such as the number of byte exchanged and the duration, tend also to assume similar values. This happens because the 5-tuple consisting of IP addresses, source and destination port numbers and protocol that we use for clustering is highly correlated to the application.

We compute the hash table index value $H_1$ as a combination of $n$ LSH functions. In detail,

$$H_1(\vec{r}) = \left(\sum_{i=1}^{n} h_{\vec{a}_i, \vec{b}_i}(\vec{r})\right) \mod P$$

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:

---

Euclidean distance sense will be mapped to the same value with high probability, and to distinct values otherwise [36].
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, `MemoryMax` and `MemoryMin`, 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 `MemoryMax`, blocks are created by packing (and purging) the longest chains. The process stops when `cnt` reaches a value lower than `MemoryMin`. A pseudocode of the online reordering process is shown in Algorithm 2.

**Algorithm 2** online-Reorder(DataStream)

**Require:** a stream `DataStream` of multi-attributes records

**Ensure:** data blocks created from sets of similar records

1: \( P := \text{hash.length}() \);  \(< \text{size of hashtable} \)

2: \( \text{while } \overrightarrow{r} := \text{getNextRecord(DataStream)} \text{ do} \)

3: 

4: /* Buffer the current multi-attribute record */

5: \( h_1 := \sum_{i=1}^{n} \text{LSH}_{A_i}(\overrightarrow{r}) \mod P \triangleq \text{HashTable index} \)

6: \( h_2 := \sum_{i=1}^{n} \text{LSH}_{B_i}(\overrightarrow{r}) \mod Q \triangleq \text{for InsertionSort} \)

7: chain = hash\[h_1\].insertionSort(r, h_2)

8: 

9: /* Archive and index blocks of similar records */

10: \( \text{if } \text{chain.length()} > \text{maxBlockSize} \text{ then} \)

11: \( \text{emitDataBlocks(chain, archive, index)} \)

12: \( \text{end if} \)

13: 

14: /* Check if we are out of the memory budget */

15: maxCount := hash.totalNumBuckets()

16: \( \text{if } \text{maxCount} > \text{MemoryMax} \text{ then} \)

17: /* Force the block creation for purging the hash */

18: \( \text{while } \text{hash.totalNumBuckets()} > \text{MemoryMin} \text{ do} \)

19: chain := longest.chain(hash)

20: \( \text{emitDataBlocks(chain, archive, index)} \)

21: \( \text{end while} \)

22: \( \text{end if} \)

23: \( \text{end while} \)
4.3 Online-LSH (oLSH) Stream Reordering

Figure 4.8: *Comparison of positions of a query without (left) and with (right) oLSH reordering.*

Figure 4.8 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. The horizontal bars of Figure 4.8, which represent the sum of matching positions within fixed-size bins, show that matching positions are spread all over the column in the unsorted version, whereas in the version using the online-LSH matching records are concentrated in the first half.

Data retrieval from compressed columns benefits from this because fewer blocks from the archive must be accessed and decompressed. In addition, as the blocks store sets of homogeneous records, the average block compression ratios are substantially improved and, therefore, as we show in the evaluation section, faster decompression speeds can be achieved.

Finally, Figure 4.9 displays an instance of the actual bitmap index capturing the first byte of the source IP field (256 columns) with and without the reordering component. The color coding on the bitmap index indicates which COMPAX codeword is used (0-Fill is shown in white). At the bottom of each
Figure 4.9: Visualizing the bitmap index with and without the oLSH component. This bitmap index captures the first byte of the source IP field (256 columns). The different COMPAX codewords are indicated in diverse colors.

bitmap index, we show the number of codewords for each column. It is easy to see that the oLSH reordering results in a significantly lower number of codewords used for encoding the same amount of information.

4.4 Evaluation

In this section, we evaluate the implementation of our approach. We investigate critical performance metrics of the archival and data retrieval process. In addition, we compare with other prevalent bitmap indexing approaches.

We use two real data sets in the evaluation:

- Six days of NetFlow traces of access traffic from a large hosting environment (HE).
4.4 Evaluation

- Two months of NetFlow traces of internal and external traffic in an average-sized enterprise production network (PN).

Each dataset is stored as a set of flat files, each corresponding to an hour of traffic. Details of the two datasets are listed in Table 4.4. The traffic of the two networks 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 4.2.

### Table 4.1: Used data sets

<table>
<thead>
<tr>
<th>Data set</th>
<th># flows</th>
<th>Length</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hosting Environment (HE)</td>
<td>231.9 million</td>
<td>6 days</td>
<td>6.9 GB</td>
</tr>
<tr>
<td>Production Network (PN)</td>
<td>1.2 billion</td>
<td>62 days</td>
<td>37 GB</td>
</tr>
</tbody>
</table>

### Table 4.2: Flow attributes present in the index and archived data columns.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Size</th>
<th>Index</th>
<th>Archive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source IP address</td>
<td>4 bytes</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Destination IP address</td>
<td>4 bytes</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Layer 4 protocol</td>
<td>2 bytes</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TCP/UDP source port</td>
<td>2 bytes</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TCP/UDP destination port</td>
<td>2 bytes</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Number of packets</td>
<td>4 bytes</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Number of bytes</td>
<td>4 bytes</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>First time stamp</td>
<td>4 bytes</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Duration</td>
<td>4 bytes</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>TCP flags</td>
<td>1 bytes</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Source AS number</td>
<td>2 bytes</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Destination AS number</td>
<td>2 bytes</td>
<td>-</td>
<td>✓</td>
</tr>
</tbody>
</table>

We have implemented the indexing, storage, and querying techniques as a C++ library of 25 000 lines of code. The library comes with three similarly tuned implementations of WAH, PLWAH and COMPAX compressed bitmap indexing algorithms. In all cases, boolean operations do not require an explicit bitmap decompression. For portability reasons, we decide not to exploit specific instruction sets, such as the SSE4.2 [69], that provides the POPCNT instruction for counting the number of bits set to 1 in an unsinged word. Instead, we rely on precomputed lookup tables for implementing bit counting.

---

5 The GNU compiler (GCC) provides the built-in function `builtin_popcount` for exploiting such an instruction, if present.
The software does not require any external library except for lzo [92], which provides us the LZO1X-1 [93] algorithm implementation that we use for compressing columns. LZO1X-1 does not offer the best compression rate, but instead is designed for achieving high compression and decompression speed.

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 formatted with a single Ext3 partition.

### 4.4.1 Disk Utilization

We measure the storage space requirements of our methodology for both the archive and the compressed bitmap indexes.

**Archive Size:** We compare the disk space requirements when storing the flow data using different storage approaches:

- The flow data are stored as uncompressed flat files.
- The flow data files are compressed with popular compression utilities: gzip and lzop. This represents the typical scenario for flow archive systems [65].
- The flow data files are split attribute by attribute. Each attribute column is independently compressed using gzip and lzop.
- Our proposed archival method that relies on a compressed columnar approach with small compression blocks. We evaluate this approach with and without oLSH reordering.

In Table 4.3 we report the storage space footprint when flow data traces are compressed using the general purpose compression utilities gzip and lzop. For both utilities, we measure the storage footprint when compressing entire flow files (row-wise) and when columns corresponding to different attributes within a flow file are compressed independently (column-wise).

---

6The hard drive is a 7200 rpms Hitachi HDP725032GLA380 equipped with 8 MB of cache. The system is capable of performing cached reading at 2400 MB/s and unbuffered disk reads at 80 MB/s (measured with hdparm).
Table 4.3: Disk space requirements when compressing the two data sets with general purpose compressors (gzip and lzop). We report the disk consumption when attribute columns are compressed independently and when the compression is done on the entire flow trace.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Raw</th>
<th>GZIP</th>
<th>LZOP</th>
<th>GZIP</th>
<th>LZOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>HE</td>
<td>6.9 GB</td>
<td>2.5 GB</td>
<td>3.5 GB</td>
<td>2.5 GB</td>
<td>3.6 GB</td>
</tr>
<tr>
<td>PN</td>
<td>37 GB</td>
<td>8.1 GB</td>
<td>13.2 GB</td>
<td>8.8 GB</td>
<td>13.4 GB</td>
</tr>
</tbody>
</table>

When compressing the flow records, the disk space is lower when using gzip compression than when using lzop. This is because the lzo algorithm implemented by the lzop utility is optimized for speed rather than compression efficiency. However, this is exactly the reason for adapting lzo in a streaming setting, where flow records have to be compressed on the fly. When compressing each attribute column independently, the disk consumption slightly increases.

In Table 4.4 we report the disk consumption when NET-FLi stores the datasets with and without enabling oLSH reordering of flows.

Table 4.4: Disk space requirements when compressing flow data with NET-FLi with and without oLSH component enabled.

<table>
<thead>
<tr>
<th>Data set</th>
<th>Raw</th>
<th>NET-FLi archive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>LZO</td>
</tr>
<tr>
<td>HE</td>
<td>6.9 GB</td>
<td>3.7 GB</td>
</tr>
<tr>
<td>PN</td>
<td>37 GB</td>
<td>13.8 GB</td>
</tr>
</tbody>
</table>

We observe that the column-oriented NET-FLi archive, without the oLSH component enabled, is up to 6% bigger than the flat files compressed with lzo. This marginal space overhead is attributed to the fact that we compress data in small blocks (up to 4000 records each). This leads to a reduction of the compression rate. In addition, our approach requires some additional space for keeping the required header at the beginning of every block. However, the above additions in our approach allow the selective decompression of any block in the archive.

The most interesting results of these measurements are the disk savings introduced by the oLSH component. In fact, the on-the-fly reordering allows the disk consumption of our storage architecture to be reduced by as much
as 40%. Therefore, the combination of the online-LSH and \textit{lzo} leads to similar compression rates as \textit{gzip} when applied to the flat files (the LZO+oLSH column of Table 4.4 displays similar numbers to the GZIP columns of Table 4.3). Thus, our storage architecture consumes as much space as standard flow-based storage solutions, but can leverage the substantially higher decompression speed provided by \textit{lzo} during data retrieval operations. As we show later in the evaluation, the oLSH component further boosts the decompression speed.

**Index Size:** We now focus on the disk consumption of the indexes built using different compressed bitmap indexing technologies. We compare the disk requirements when indexing the seven (most queried) flow attributes reported in Table 4.2 using our encodings COMPAX and COMPAX2 and the existing WAH and PLWAH.

Table 4.5 reports the disk consumption used by the WAH, PLWAH, COMPAX and COMPAX2 indexes without oLSH-based sorting. Compared with WAH, COMPAX and COMPAX2 indexes are up to 40% smaller, whereas PLWAH offers comparable compression rates.

<table>
<thead>
<tr>
<th>Data set</th>
<th>WAH</th>
<th>PLWAH</th>
<th>COMPAX</th>
<th>COMPAX2</th>
</tr>
</thead>
<tbody>
<tr>
<td>HE</td>
<td>8.1 GB</td>
<td>5.1 GB</td>
<td>4.9 GB</td>
<td>4.9 GB</td>
</tr>
<tr>
<td>PN</td>
<td>26.3 GB</td>
<td>17.4 GB</td>
<td>18.6 GB</td>
<td>18.0 GB</td>
</tr>
</tbody>
</table>

As shown in Table 4.6, enabling the oLSH component allows the index size to be further reduced not only for COMPAX but also for other bitmap indexing techniques, making our contribution of independent interest. For example, WAH indexes are more than 42% smaller when oLSH is enabled. COMPAX2 is the bitmap indexing technology providing the lowest disk consumption (up to 30% lower than WAH), but the gain is only marginal compared with COMPAX and PLWAH, which provide similar compression rates when oLSH is enabled.

**Visualizing the oLSH Improvements:** One can also communicate the improvements attributed to the oLSH record reordering process visually. In Figure 4.10, we plot the bitmap index as encoded by COMPAX and COMPAX with oLSH by depicting an \textit{uncompressed} version of the bitmap. The various
Table 4.6: Comparison of index sizes built using different bitmap encodings with the oLSH component enabled.

<table>
<thead>
<tr>
<th>Data set</th>
<th>WAH oLSH</th>
<th>PLWAH oLSH</th>
<th>COMPAX oLSH</th>
<th>COMPAX2 oLSH</th>
</tr>
</thead>
<tbody>
<tr>
<td>HE</td>
<td>4.8 GB</td>
<td>3.4 GB</td>
<td>3.3 GB</td>
<td>3.2 GB</td>
</tr>
<tr>
<td>PN</td>
<td>15.1 GB</td>
<td>11.7 GB</td>
<td>12.8 GB</td>
<td>11.6 GB</td>
</tr>
</tbody>
</table>

codewords are indicated with different colors provided in the legend. We display the bitmap indexes for a number of attributes, namely, all four bytes of source IPs and destination IPs, source and destination ports (only partially), and protocol.

As an example, the bitmap index on the top-left-hand side of the figure, captures the first byte of the source IP field of the network flow records. There are 256 columns on this bitmap, one for each of the 256 values possible. Each of the bitmap indices displayed reflects the relevant field for a traffic duration of one hour. Next to the bitmap encoded using COMPAX, we position the resulting bitmap index when oLSH is enabled. On the far-right-hand side, we display the size ratio of the oLSH encoded bitmap index versus that of the unsorted traffic.

The visual comparisons cogently show the power of the oLSH process in producing compact bitmap indexes.

4.4.2 Stream Record Processing Rates: Archive and Index

NET-FLi has been designed to handle high-speed streams of flow records, so in this section we measure the average sustainable insertion rate, expressed in flows per second (f/s). We test our storage solution by feeding it with uncompressed flow traces stored on a mainstream solid-state drive\(^7\). The drive provides a sustained reading speed of 170 MB/s corresponding to more than 5 million f/s. The indexing software has been configured to fetch flows sequentially from the solid-state drive and to store indexes and compressed columns to the mechanical 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 we compare it with the insertion rates of WAH and PLWAH-based indexes. In Table 4.7, we report record processing rates for

\(^7\)Intel X-25M G1, 80 GB model
Figure 4.10: Visual representation of the savings induced by the oLSH component. We display the bitmap index with and without the online reordering.

Table 4.7: Record processing rates of our system when creating both the index and the compressed archive. We compare building the index using WAH, COMPAX and COMPAX+oLSH.

<table>
<thead>
<tr>
<th>Data set</th>
<th>WAH</th>
<th>PLWAH</th>
<th>COMPAX</th>
<th>COMPAX oLSH</th>
</tr>
</thead>
<tbody>
<tr>
<td>HE</td>
<td>768K</td>
<td>924K</td>
<td>936K</td>
<td>474K</td>
</tr>
<tr>
<td>PN</td>
<td>1150K</td>
<td>1220K</td>
<td>1255K</td>
<td>513K</td>
</tr>
</tbody>
</table>

building both the index and the archive. We compare three variants: 1) indexes encoded with WAH, 2) with PLWAH, 3) with COMPAX and 4) with COMPAX and additional oLSH reordering.

Without oLSH reordering enabled, COMPAX offers insertion rates that are higher than WAH and comparable to PLWAH. The advantages in terms of
insertion rate are not only related to the lower disk consumption. In fact, we can measure similar differences (in percentage) when index serialization is disabled. By using a profiler, we realized that COMPAX is much more cache-friendly than WAH. Indeed, COMPAX can perform its on-line compression step that can pack three WAH words into a single word (FLF or LFL) without requiring many additional L2 cache misses in average. The test is confirmed by the higher insertion rate on the data set HE, which, because of its entropy in many attribute fields, taxes the CPU cache more than the PN dataset. Similar results can be observed for PLWAH, which offers similar indexing rates to COMPAX.

Without enabling oLSH reordering, our storage solution can handle as many as 1.2 million flows/sec. To put these numbers into perspective, medium-large networks exhibit peak rates of 50,000 flows/sec or more. When the oLSH component is enabled the insertion rate is reduced to half. The drop is substantial, but this eventually results in a disk space reduction of as much as 55%, and in a significant improvement in response time (details in the following section). Nonetheless, it is worth considering that 500K flows/sec is still more than double the maximum insertion rate reported in [102], where no data compression was performed.

### 4.4.3 Extended Comparison with PLWAH

As discussed in Section 4.1, PLWAH and the COMPAX have been introduced for achieving better compression rates than WAH in the case of sparse bitmaps. On real traffic traces, the two techniques provide equivalent compression rates. In this section, we offer an extended comparison with PLWAH under the more challenging condition of uniformly distributed data. Even if such traffic patterns are quite uncommon during regular traffic conditions, anomalous events such as these caused by port scans and worms, may lead to distributions that are close to uniform. It is worth to note that achieving higher compression rates under such circumstances is desirable as it renders the storage architecture more resilient to attacks.

To evaluate the different encoding techniques under this scenario, starting from a real flow trace of 2 million records, we synthetically create a new flow trace in which IP addresses and ports are uniformly distributed. We index the synthetically generated trace using WAH, COMPAX and PLWAH. In Table 4.8, we report the disk consumption, expressed in KB, when indexing some of the most queried fields.
Table 4.8: Disk consumption (in KB) of WAH, PLWAH and COMPAX when indexing IP addresses and ports with a uniform distribution

<table>
<thead>
<tr>
<th>Field</th>
<th>WAH</th>
<th>COMPAX</th>
<th>PLWAH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source IP address</td>
<td>52296</td>
<td>21046</td>
<td>29145</td>
</tr>
<tr>
<td>Destination IP address</td>
<td>52285</td>
<td>21049</td>
<td>29144</td>
</tr>
<tr>
<td>Protocol</td>
<td>435</td>
<td>449</td>
<td>396</td>
</tr>
<tr>
<td>TCP/UDP source port</td>
<td>14941</td>
<td>13446</td>
<td>7602</td>
</tr>
<tr>
<td>TCP/UDP destination port</td>
<td>14941</td>
<td>13444</td>
<td>7602</td>
</tr>
<tr>
<td>First time stamp</td>
<td>2247</td>
<td>1958</td>
<td>1870</td>
</tr>
</tbody>
</table>

When indexing IP addresses, COMPAX offers significantly smaller bitmap index size than WAH and PLWAH. In particular, COMPAX-encoded indexes are just 40% of the size of WAH encoded indexes, whereas PLWAH are 56%.

On the other hand, when indexing uniformly distributed port numbers, PLWAH offers the best compression rate. This is the case, because the FLF and LFL codewords introduced by COMPAX cannot be frequently applied, as the length of the fill is often longer than 256.

To better illustrate why PLWAH provides better compression rates than COMPAX when indexing the port number field, we analyze the COMPAX and PLWAH-compressed bitmap indexes. In particular, we are interested in understanding for which bitmap index columns PLWAH compresses better than COMPAX. In Figure 4.11, we report the disk consumption (expressed in numbers of 32-bit words) for every column of the bitmap index. Columns have been ordered by number of matching results, i.e. the number of ones they carry, so that sparse columns are on the left and dense columns on the right. The figure shows a clear separation between the two encodings: PLWAH performs better when compressing sparse columns, whereas COMPAX offers better compression rates for denser columns, which usually require more space. However, the difference in space consumption between COMPAX and PLWAH is higher for dense columns and lower otherwise. This means that when performing queries for common ports (dense columns), the COMPAX index requires fewer words to be fetched from the disk.

Our comparisons of the two bitmap encoding when used to index port numbers and IP addresses, suggest that when one is interested in achieving the best possible compression ratios, different encodings can be used for compressing distinct data attributes, or even distinct bitmap index columns within the same bitmap index. However, the adoption of heterogeneous compressed
Figure 4.11: Comparison between COMPAX and PLWAH. The x-axis shows the number of results (number of ones) contained in a column. The y-axis shows the number of 32-bit words (codewords) that encode the particular bitmap index column. At the bottom of the figure we also juxtapose a ‘win-lose’ graph that illustrates when each technique is better. Evidently, COMPAX compresses data using fewer words, when the data contain large number of matches (dense columns). Sparse columns holding few matches are encoded more efficiently using PLWAH.

bitmaps technologies requires boolean operations to be implemented across different encodings. While this approach is feasible, it results, in practice, in a substantially more complex implementation that makes the software harder to test, debug and maintain.

There are two main reasons why COMPAX has to be preferred over PLWAH in a networking context. First, COMPAX can better compress IP addresses, which are i) more commonly queried than port numbers, and, ii) fields where a higher number of distinct values, and therefore a larger index size, has to be expected, especially in the case of large networks. Second, the disk footprint benefits achieved by PLWAH when indexing port numbers are significant mostly in the case of uncommon ports, which, individually, do not require substantial indexing space.
4.4.4 Index Performance

In this section, we evaluate the overall index performance with WAH, since it has already been extensively used by the research community, and because there is a well tested and freely available implementation of WAH in FastBit [124]. We compare the search performance of the bitmap index solely, when encoded with WAH, COMPAX, or COMPAX with oLSH enabled (COMPAX+oLSH). We pose queries using random IP addresses, since IP addresses are typically the most often queried attribute. In addition, by querying for exact IP addresses, we are evaluating the performance of boolean operations over compressed bitmaps as each IP lookup requires the bitwise AND-ing of four different bitmaps, each corresponding to one of the four bytes of an IPv4 address. We conduct the experiment over the more heterogeneous ‘Hosting Environment’ (HE) dataset.

![Index search response times](chart)

**Figure 4.12:** Comparison of index search time for 3000 random IP addresses.

We choose 3000 random distinct IP addresses and group them into 30 sets of 100 addresses each. For every set, we report the time for executing the 100 independent IP address lookups executed sequentially. To put into perspective the performance offered by our system, we also uses WAH indexes in addition to COMPAX based indexes. Because we are interested in measuring the performance of the index in terms of the CPU efficiency, we built an index over 3 days from the HE data set, which can be completely be cached by the system. The size of the index is 1.5 GB for WAH, 845 MB for COMPAX and 314 MB for COMPAX+oLSH. The measurements are reported in Figure 4.12.

The response times shown depend entirely on the performance on the
4.4 Evaluation

boolean operations among the IP address byte indexes. The index query on
the COMPAX-based index is on average 15% faster than WAH-based index.
This result shows that adopting a more complex bitmap encoding does not
necessary result in performance penalties when implementing boolean oper-
ations directly on the compressed bitmaps (i.e., without explicit decompress-
ion). In fact, for COMPAX bitmaps, the more complex decoding is compen-
sated by increased cache locality which is a direct consequence of the space
savings (smaller working set). The performance figures that we get by com-
paring COMPAX with WAH, are similar to the ones reported by the authors
of PLWAH in their original paper [39], where PLWAH has been shown to
achieve at most 20% better response times than WAH and comparable per-
formance in many cases.

However, by enabling the oLSH component, COMPAX+oLSH is four
times faster, on average, than WAH to complete the IP address lookups. In this
case, the improvement is not just due to increased cache locality, but rather
to the sorting itself. The sorting results in literal words that are more dense;
so, fill words can represent longer sequences. In such a way, the number of
bitwise instructions is substantially reduced [73].

4.4.5 Query Performance

Finally, we measure the flow record retrieval performance offered by the
created compressed repositories for two different use cases: high selectivity
queries that rely on indexes, and low selectivity queries.

COMPAX indexes and compressed data archive are created with and
without oLSH reordering. We execute queries over the repository correspond-
ing to the dataset PN, which is worth of 1.2 Billion flow records. It is worth
to note that: i) we query the created repository without concurrently using
the system to store flow record streams, and ii) we use sequential (i.e. single-
threaded) implementations of the retrieval processes for posing queries. In
this way, our measurements are less affected by system noise (e.g. scheduling
and thread synchronization), and still able to capture realistic use cases, as
long-term historical analyses tasks are more frequently performed on storage
nodes rather than on the collection node itself.

The overall disk space required to store both the indexes and the archive
corresponding to the dataset PN amounts to approximately 31 GB without
the oLSH option. By enabling the oLSH sorting, the resulting repository size
is 20 GB. Naturally, we expect that the oLSH variant results in better query
response times.

In what follows, we describe the experiments corresponding to the two use cases mentioned above and we report the measured performance results.

**High Selectivity**

We pose 10,000 random IP address queries with an increasing number of wild cards 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 that would have to retrieve an increasing number of results. We sort and ‘bin’ the number of results into a histogram format. Each bin represents the average number of results returned by all queries in the same bin.

In Figure 4.13, we report three graphs corresponding to the following performance measurements:

1. **The index time**: is the time required to load the relevant bitmap index columns and to join them.

2. **The archive time**: is the time spent in retrieving the data from the compressed archive.

3. **The total query time**: is the sum of the index and archive times.

Each graph illustrates the runtime with and without the oLSH option. One can observe that by using oLSH reordering the total query time improves by more than 2 times. This result is attributed to the smaller sizes of both index and archive after oLSH reordering. For the queries with the lowest selectivity, the response time can reach 3 sec, whereas for the queries with higher selectivity the results can be returned in less than a second. This outcome is very encouraging because we do not make the query be selective with respect to time (i.e. “find only matches during day A”), but queried for results through the whole flow repository.

On the same figure, in the bottom graph, we segregate the total query time into its components. The time to retrieve results from the index and perform a join on them (Figure 4.13 middle) is nearly constant and approximately on the
order of 500 msec. The majority of the search query is spent on the retrieval from the archive, which is depicted on the lower part of the Figure. One can easily observe the great performance boost attributed to oLSH reordering. The careful packaging of flow records in both the index and the archive eventually leads to a significant improvement in the query performance. In particular, the gains from flow reordering in the archive can reach the 400% range attributed
to the better packing of similar records. In fact, the reordering reduces the number of blocks that must be decompressed at the data archive level and, in addition, because of the reduction of the overall entropy within each block, it boosts the decompression performance.

**Low Selectivity**

In many cases, decompressing the entire data archive or entire data columns might be required; this is the case, for example, when one has to compute the set of distinct IP addresses. For such a scenario, the bitmap index does not help, as the entire archive on one or more attributes needs to be decompressed. To evaluate such cases, we measure the decompression speed when the *entire* data archive is unpacked. We measure both variants: with and without oLSH tuple reordering.

For each network flow attribute, we measure the CPU time spent on the execution of the decompression routines only. In this way, we do not take into account the time spent on I/O, which is higher for the non-ordered data because of the higher disk consumption. Therefore, the decompression speeds reported in Figure 4.14 (expressed in Megabytes/second) represent a lower bound for the decompression performance improvement.

We observe that the oLSH reordering significantly boosts the decompression speed of the lzo compressor by as much as 3.6 times. In both datasets, the decompression speed boost is substantial for IP addresses and ports, which are the most commonly queried attributes.

It is worth noting that, with the only exception of the timestamp attribute archive, the reordering has a positive impact on the decompression speed. This is indeed expected even if the system uses IP addresses, ports and protocol attributes (the so-called 5-tuple) when performing the oLSH reordering. In fact, there is a high correlation between the 5-tuple and the remaining fields with the only exception of the timestamp attribute. This can be intuitively explained as a combination of ports and protocol identifies in the majority of the cases the networked application (e.g. HTTP usually runs on port 80/TCP), which, in turn, tends to manifest specific behaviours in terms of session duration and number of packets and bytes transferred. Similarly, the source and destination Autonomous Systems are related to the IP addresses. The timestamp attribute, on the contrary, is not strongly related to any other 5-tuple field, and, in addition, it is the only attribute having partially ordered values.

---

8Flow meters export a flow record once it is expired or terminated. The timestamp of the first
For these reasons, timestamp is the only attribute where oLSH reordering has a negative impact on the compression ratio, and, consequently, on the decompression speed.

**Figure 4.14:** Decompression speed in MB/sec. Note that for the most commonly queried attributes, such as source/destination IP addresses and ports, the decompression speed is more than double when oLSH reordering has been applied.

packet belonging to the flow is not used by flow meters to pose an order on the exported flow records.
4.5 NET-FLi at work

In this section, we demonstrate the utility of NET-FLi for typical scenarios in network monitoring: 1) detection of worm propagation, and 2) interactive exploration of the network traffic history.

Worm Propagation

Consider the scenario of an outbreak of a computer worm (cf. Figure 4.15) that exploits a vulnerability of an application protocol. All network nodes being contacted by the infected host need to be discovered to a) quarantine any other possibly infected machine, and b) better understand the propagation pattern of the worm.

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 needed to identify possibly infected machines.

We perform an analysis on the 2-month PN data set consisting of 1.2 billion flows and focus on machine $M$ that is suspected to spread a worm using a vulnerability on service port 123. We query the system for all machines to whom the machine connected to on service port 123. Therefore the query has the form:

```
SELECT DstIP
WHERE SrcIP = M AND DstPort = 123
```

Figure 4.15 displays an example of a worm propagation pattern: the center node represents a suspicious host, whereas an edge signifies a network communication on a vulnerable port between the two hosts.

The connectivity graph to a node can be created with NET-FLi by recursively querying for flows emitted by suspected 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 (after mounting the disk) as well as with a ‘warm’ cache at the operating system level, when other queries have previously been posed. We repeat the experiment 100 times. We discover that the uncached query response time is 62.314 sec on average (with a standard deviation $\sigma$ of 0.798). When reissuing the same query with a previously ‘warm’ cache, the response time drops down to 2.345 sec on average ($\sigma = 0.051$). In
comparison, the same 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 than the identical NET-FLi query.

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

Interactive Exploration

On top of NET-FLi we built an interactive graphical interface that enables easy exploration of network flows. Given a specific IP (port, protocol, etc) and a time frame, the interface discovers all flows involved, which are then overlaid on a map of the globe. Results are also aggregated by country. An
example of this query interface is shown in Figure 4.16. For the IP address, date and time (which can be of a compromised node), we can quickly identify which other Internet addresses are contacted by this host. Contacted destinations are aggregated by country on the right-hand side. However, the user can also drill-down on the individual IP addresses for more information (top-right graph). In addition, basic flow filtering by country is also incorporated in the visual interface (middle-right graph).

**Figure 4.16: Application for interactive exploration of flows and anomalies built on top of our technology**

Finally, we provide the capability to visualize the actual bitmap index for any of the indexed attributes. This is a particularly useful functionality, because it facilitates the visual identification of anomalies. Consider, for example, the snapshot of the bitmap index shown on the bottom left-hand side...
of Figure 4.16. This captures a portion of the bitmap index for the attribute ‘Destination Port’ of the network flows. From the horizontal lines (columns represent the different ports), it is immediately apparent that some resource is initiating a port scanning process. If this belongs to an unscheduled activity, it is something worth investigating by the data administrator.

Such cases can easily be identified by visualizing the bitmap indexes constructed. In addition, recording an ‘average’ or expected bitmap index profile and monitoring the deviation of the current traffic from that profile, can provide an interesting avenue for discovering network anomalies. These possibilities can be potentially accommodated on top of the provided technology and visualization tools, but reside outside the scope of the current work.

4.6 Summary

In this chapter we introduced NET-FLi, a high-performance solution for high-speed data archiving and retrieval of network traffic flow information. 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 streaming contexts challenged by high data volumes and exceptionally high insertion rates. Our solution achieves the following:

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

- Compression ratios equivalent to that of gzip achieved by **boosting** a lightweight compressor, LZO, with an effective stream based flow record reordering approach based on locality sensitive hashing (LSH).

- Decompression speeds on par with LZO, which is among the fastest general purpose compressors. The proposed record reordering improves the LZO decompression speed as much as 3.6 times.

- Efficient drill-down functionalities with support for fast existence queries achieved by introducing support for partial decompression.

- Space efficient indexes based on a novel adaptive, compressed bitmap indexing technique, which we call COMPAX. The record reordering scheme substantially improve the compression ratio of COMPAX and other compressed bitmap indexing encodings.
Our solution achieves interactive response times when performing drill down operations by reducing the amount of data to be compressed at retrieval time. Fine grained decompression is achieved by exploiting the synergy of compressed bitmap indexes and small data compression units, or data blocks. In our implementation, we compress data blocks with one of the fastest general purpose compressor implementation, LZO, which is optimized for achieving high decompression speeds rather then high compression ratios. To boost its performance, we introduced a flow record sorting technique that improved both decompression speed and compression ratios.

Nevertheless, as we have shown in Figure 4.13, query response times are still dominated by the time spent in decompressing data blocks. In the next chapter, we describe a novel compressor, called RasterZip, designed to be a better replacement for LZO.
Chapter 5

RasterZip: a real-time compressor with partial decompression

In the previous chapter, we have described NET-FLi, a storage architecture that can be used to create highly compressed flow record repositories that are amenable to search. We have shown that queries can be efficiently answered by using indexes in order to reduce the amount of data to decompressed at retrieval time. In addition, we have shown that a general purpose compressor optimized for achieving high decompression speed rather than compression ratio, LZO, can compress the data as much as GZIP by applying the proposed approximate stream-sorting technique oLSH.

In this chapter, we describe a domain-optimized compressor, which we call RasterZip, that significantly outperforms LZO in terms of compression ratio and decompression speed without penalizing the maximum insertion rate provided by NET-FLi. RasterZip is optimized for network traffic data. It uses a carefully-engineered encoding to exploit data patterns such as bounded data-ranges, frequent value repetitions and shared prefixes that are typically present in many network traffic attributes (e.g. timestamps and IP addresses), and also introduced by the approximate sorting technique described in the previous chapter.

In addition, RasterZip has been designed to exploit the presence of
indexes and has built-in support for partial and selective decompression. RasterZip is both attribute- and index-aware as it provides fine-grained decompression granularity at the attribute level: it compresses blocks of homogeneous attribute values as multiple compressed sub-blocks allowing selective decompression at query time. More in detail, attribute values stored at specified positions can be retrieved from a RasterZip-compressed data block without requiring the entire block to be decompressed.

By introducing RasterZip we make the following contributions:

- **Carefully-Engineered Encoding.** We introduce a new RLE-based compression encoding that a) is designed for network traffic data, b) exploits features of modern CPU architectures, and c) can efficiently cope with RLE compressible and incompressible content.

- **Novel Decompression.** We propose new exciting avenues for data decompression. Our methodology provides fine-grained sub-block decompression; only portions of the archive block that contain query results are decompressed. In addition, we introduce an adaptive decompression strategy that intelligently selects at runtime between full or partial block decompression, according to which strategy is expected to be the fastest. This, essentially, encodes a very lightweight decision classifier for the offered decompression strategies.

- **Better Performance.** Compared to LZO, which is the fastest LZ based compressor, RasterZip provides better response times for high selectivity queries; creates 22% smaller archives; and achieves equivalent compression speeds.

This chapter is structured as follows. In Section 5.1 we provide an overview of our approach and we revisit the oLSH reordering mechanism to better understand the data patterns introduced. In Section 5.2 we describe RasterZip and in Section 5.3 we evaluate it. Finally, Section 5.4 concludes the chapter.

### 5.1 Overview of the Approach

In this section, for clarity of presentation, we briefly revisit the flow record compression methodology implemented by the architecture described in the previous chapter.
5.1 Overview of the Approach

Figure 5.1: An overview of the compression methodology for streaming records. In this chapter we focus on the data compression component.

The compression methodology is illustrated in Figure 5.1. The multi-attribute streaming records (5.1.A) are hashed into hash-buckets containing chains of similar records (5.1.B). Data blocks of homogeneous attribute types are created from each hash chain by separating flow records in columns (5.1.C) and then indexed (5.1.E) and archived (5.1.D). In this chapter, we treat indexing as a black-box that can be realized by high-speed bitmap indexes like the ones described in Chapter 4 and we focus, instead, on the archival component (5.1.D).
Each data block is archived in fine grained compression units, which we call sub-blocks. At query time, the index will return the row positions of interest in the archive. This information is used by the query processor to identify the compressed data blocks of the archive that contain the requested data. More importantly, the query processor decides at runtime whether it is more beneficial to decompress the whole archive block or only the specific sub-blocks that contain the answer to the query (whichever is faster).

### 5.1.1 Online Record Reordering

In the previous chapter, we introduced oLSH in order to boost the compression ratio of both bitmap indexes and the general purpose compressor, LZO, that we applied for compressing data columns. As we have shown in the previous chapter through a visual representation of the bitmap indexes (e.g. Figure 4.10), the oLSH reordering technique produces longer sequences of repeated values.

![Difference of consecutive values](image)

**Figure 5.2:** We depict the cumulative difference between consecutive values per block archive. **Top:** Using the online-LSH approach the entropy of the data is significantly reduced. **Bottom:** Storing the streaming records without reordering results a difference of values that is up to 14 times larger.

Figure 5.2 illustrates the benefits of online record reordering in a different
way. For one hour of network traffic we create the archive for the attribute of destination ports, with and without the oLSH reordering enabled. For each processed block of streaming records we compute the sum of differences between consecutive values. This procedure is repeated for every data block. We depict this because smaller consecutive differences of values give rise to better compression schemes. The unsorted stream shown at the bottom depicts significantly lower homogeneity: the cumulative difference between consecutive values is up to 14 times larger compared to the stream using oLSH-reordering. It is evident that the process is very effective in clustering values that belong to adjacent ranges and, thus, that share common prefixes. It is worth to note that the reordering is particularly effective for the attributes that are used as input of the reordering process (IP addresses, ports, and protocol). The values of the other attributes, e.g., flow duration, typically share a long common prefix anyway, which is the reason why they do not have to be taken into account for the oLSH process.

In the next section, we describe RasterZip, a novel compressor that exploits these compression opportunities.

### 5.2 Algorithm

The online stream reordering mechanism provides blocks of attributes that exhibit areas of high homogeneity. To exploit the common prefix structure created by the record rearrangement, we propose an effective compression mechanism that scans the data column-wise and takes advantage of the fact that in this manner adjacent bytes are often identical. The column-wise order is the basis of our compression based on run-length encoding (RLE) principles. Because we scan the data in a raster-like fashion, we call the new compression algorithm *RasterZip*. Our approach also supports partial data decompression, by decompressing only the RLE subblocks that contain results of interest. Finally, we show how to adaptively select between full and partial decompression at runtime. We introduce an adaptive decompression component that intelligently shifts between full- or partial-decompression, according to which approach is expected to result in lower decompression time. In this section, we discuss our compression and decompression techniques in detail.
5.2.1 Compression

The compression routine operates on a data block of a single attribute. Each block consists of $m$ values of size $n$ bytes. In our setting $n$ would correspond, for example, to two bytes for port numbers, or four bytes for IPv4 addresses. Each data block of every attribute is treated as an $m \times n$ matrix of bytes, stored in row-major order.

Traditional data archiving approaches process the data row by row for reasons of efficiency and simplicity. Because of the data reordering, consecutive row-records share similar/adjacent values and hence have common prefixes. RasterZip compresses data efficiently by processing them column by column, where each column represents one byte of the currently processed data attribute. This conceptual and practical distinction of our approach is shown in Figure 5.3. Note, that since rows share common prefixes, we expect very high compression for the first byte-columns. For the last byte-columns where higher variability is generally observed, lower compression ratios are to be expected.

![Figure 5.3: Left: Traditional approaches store and process data bytes in a row-wise fashion. Right: RasterZip scans the data bytes of an attribute in a columnar manner, in order to exploit the prefix structure of the input data.](image)

Based on the above discussion, compression consists of two logical steps. First, the matrix is (logically) transposed and then the resulting matrix (in column-major order) is traversed row-by-row to compress long sequences of repeated symbols. We provide an in-depth description of the two steps and the resulting encoding.

1) Transposition. The $m \times n$-matrix is transposed, i.e., the $i^{th}$ row becomes
the $i^{th}$ column for all $i \in \{1, \ldots, m\}$. Given that the first bytes of subsequent data items (i.e., subsequent entries in each of the first few columns in the original matrix) are often the same, transposing the matrix creates rows with many identical bytes in a row. As a small example, given an input of three IPv4 addresses ($3 \times 4$ byte matrix), stored in memory as

$$10.4.20.22|10.4.20.23|10.4.21.24,$$

the transposition yields the following representation:

$$10.10.10.4.4.4.20.20.21.22.23.24.$$

This transformation results in long sequences of identical bytes not only for IP addresses, but also for other fields such as timestamps, flow duration, and so on. We emphasize, that the above transposition is merely logical. One does not need to carry out the matrix transposition, but only traverse the data in a different order.

2) Repeated symbol compression: RasterZip efficiently compresses long sequences of repeated symbols (runs) using a technique based on run-length encoding (RLE) principles. RLE encodes the input stream as a list of pairs $(r, l)$ where $r$ is the symbol and $l$ is the length. For example, the sequence $9,9,9,4,4,4,3$ is by RLE encoded as $(9,3),(4,3),(3,1)$. RasterZip encodes runs in a more structured and efficient way that provides higher decompression performance compared to the RLE encoding, particularly when highly compressible and incompressible content are interleaved in the input stream. In addition, our encoding has been designed to enable random access on the compressed data. This is particularly useful for supporting partial data decompression.

![Figure 5.4: Flowgraph of RasterZip compression. Sequences of repeated symbols in the input stream are detected by the Runs Encoder and encoded as Bitmap (B) or Verbatim (V) blocks by the Block Encoder.](image)

Figure 5.4 depicts a flowgraph of the compressor. The input stream for the compressor is the matrix given in column-major order. This is the currently
processed data window, representing the output of the online-LSH component described in the previous chapter. The runs encoder scans the input stream and outputs to the block encoder pairs of the form \((r, l)\) representing sequences of repeated symbols. The block encoder groups together runs in batches of 32. In addition, it dynamically selects between two different subblock types: a Verbatim (V) block which stores RLE-incompressible content and a Bitmap (B) block which stores RLE-compressible content.

Specifically:

- **V-blocks** store a group of up to 32 runs whose length is exactly one. Runs of length 2 are stored as two runs of length 1 (i.e. the block encoder never receives runs of length 2 from the runs encoder). Therefore, such a block stores data that is hard to compress using RLE. A one-byte header is prepended. The header specifies the block type using 1 bit (i.e. that it is a V-block) and the number of runs which are stored in the block using five bits \((2^5 = 32\) runs). Because this block-type only stores runs of length 1, lengths need not be explicitly stored. Figure 5.5 depicts how an RLE incompressible input stream is encoded with a V-block.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{v-block-example.png}
\caption{V-blocks are used to store content that is incompressible with RLE. V blocks store value repetitions of length 1. Runs of length 2 are split in two runs of length 1.}
\end{figure}

- **B-blocks** encode up to 32 runs where at least one run has length greater than 1. B-blocks consist of:

  1. A header (1 byte) specifying the block type and the number of runs (similarly to V-blocks).

\[^1\text{A block of attribute values is encoded into multiple subblocks of type V or B. The terms V-block and B-block refer to subblocks. The proper interpretation of the term block should be clear from the context.}\]
5.2 Algorithm

2. A presence bitmap (32 bits) indicating which runs have length greater than 1.

3. A Runs part storing up to 32 runs.

4. Lengths part storing up to 32 lengths.

Therefore, unlike the original RLE, RasterZip does not store runs and lengths as pairs. Instead, runs are stored sequentially, followed by the sequence of the corresponding lengths. The reason for this choice becomes clear in Section 5.2.2 where we describe how the V- and B-blocks are created in an online fashion.

Note that a B-block stores mixed content of compressible and incompressible content. Therefore, the presence bitmap (32 bits) exactly indicates which of the runs have length greater than 1. Recall that runs of length 2 are always treated as 2 consecutive runs of length 1. So, length is only explicitly stored for runs with length greater than 3. This is indicated by setting the corresponding bit in the presence bitmap to one. We also emphasize that, having allocated one byte for each length, we could have stored $2^8 = 256$ consecutive identical symbols. However, we exploit the fact that we do not explicitly store lengths 1 and 2. Therefore, each length-byte in the B-block can store lengths in the range $[3, 258]$. This allows us to pack long sequences of the same symbol even more effectively.

An example of a B-block is given in Figure 5.6. For the initial sequences of 10’s and 9’s the lengths are explicitly stored (4 and 3, respectively), while for the singular values that follow, no explicit length is recorded. This is captured in the presence bitmap, by setting the appropriate bits to 1 or 0.

**Input Stream:**

10, 10, 10, 10, 9, 9, 9, 8, 7, 4, 3, 10, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 7, 7

![Binary representation of the B-block](image)

**Figure 5.6:** B-blocks store RLE-compressible content. B blocks encode up to 32 consecutive runs where at least one run has a length greater than 1. Only lengths greater than 1 are explicitly stored.
5.2.2 Encoding properties

The proposed encoding presents several desirable properties with an emphasis on exploiting features of modern CPU architectures for enhanced performance. In particular, RasterZip provides:

- **Online blocks creation**: The two types of blocks can be created in a streaming fashion. First, note that a V-block can be treated as a special case of a B-block, as it is a B-block whose presence bitmap does not have any bits set to 1. Therefore, the block creation always commences by assuming that the current block is a B-block. Whenever a run of length greater than 2 is observed, the presence bitmap is updated and the corresponding length is appended as well. After 32 runs are processed, we check whether the completed block contains at least one run of length greater than 2 by examining if at least one bit is set in the presence bitmap. In that case, the block is finalized and a new B-block is created. Otherwise, the presence bitmap is dropped and the block is changed into a V-block by simply flipping the first bit in the header byte. Pseudocode of this process is given in Algorithm 3.

- **Low memory footprint during compression**: Through the whole compression process, the algorithm is only required to allocate memory for a single RasterZip sub-block. A sub-block is very small: at most 69 bytes (header + bitmap + runs + lengths = 1 + 4 + 32 + 32). We also note that the average sub-block size is even smaller in practice. We observed an average of 40 bytes in our experiments. The cache-line size on modern processors is 64 bytes; therefore, at most 2 cache-lines are used during the compression. This leads to a low number of cache-misses and enhanced performance of the proposed compression algorithm. In contrast, popular compression algorithms, such as the Lempel-Ziv based compression schemes, use hash tables or dictionaries to store previously seen data [107], and as such, result in many cache misses in practice.

- **Efficient handling of incompressible content**: RasterZip encodes data that is hard to compress using RLE (i.e., the length of each run is less than 3) with a V-block. This block-type enables fast decompression of incompressible data, because decompressing a V-block merely requires copying up to 32 bytes to the output, which can be implemented using the `memcpy()` function. On the contrary, decompressing hard to compress data encoded with the original RLE encoding requires a conditional branch (if statement) for every symbol, even if the length is as short as 1 or 2.
Algorithm 3 Compress(input)

Require: Uncompressed input stream
Ensure: Compressed output stream

1: currRun := 0
2: currLen := 0
3: bitmap := emptyBitmap()
4:
5: /* Produce (run, length) pairs from the input stream */
6: while (r, l) := getNextRun(input) do
7:   /* Buffer the run, and the length if greater than 1 */
8:     runs[currRun++] := r
9:     if (l > 1) then
10:        setBit(bitmap, currRun)
11:        lengths[currLen++] := l
12:   end if
13:  end while
14:
15: /* Encode a V or a B block every 32 tuples */
16: if (currRun == 32) then
17:    if isEmpty(bitmap) then
18:       block := createVBlock(currRun)
19:       block.appendRuns(runs, currRun)
20:    else
21:       block := createBBlock(currRun)
22:       block.appendBitmap(bitmap)
23:       block.appendRuns(runs, currRun)
24:       block.appendLengths(lengths, currLen)
25:    end if
26:
27: currRun := 0
28: currLen := 0
29: clearBitmap(bitmap)
30: output.append(block)
31: end if
32:
33: end while

The next two items show how RasterZip exploits features offered by modern CPU architectures. These characteristics are useful primarily during the decompression process, as they allow the efficient traversal of RLE subblocks and the computation of the span of each subblock.

- **Fast traversing of the compressed data:** The simple structure of V and B blocks allows compressed data to be traversed quickly. A V-block storing \( n \) runs always consists of \( 1 + n \) bytes, and \( n \) can be read from the first byte after
masking the three most significant bits to 0. Similarly, the size of a B-block is $1 + 4 + n + b$ bytes, where $b$ corresponds to the number of bits set to one in the bitmap. The value of $b$ can be efficiently computed using specialized bit counting instructions offered on modern processors. For example, Intel introduced POPCNT to the SSE4.2 instruction sets, which offers a dedicated assembly instruction to perform the bit counting operation.\(^2\)

- **Efficient computation of the block span**: given a block, its span is the number of bytes of uncompressed data stored in it. For both block types, the span can be efficiently computed: the span of a V-block is the number of runs that it stores. Computing the span of a B-block requires summing up the lengths greater than 2 and adding the number of runs with length 1. More formally:

$$Span_B = (n - b) + \sum_{i=1}^{b} l[i]$$

where $n$ is the number of runs, $b$ denotes the number of bits set to 1 in the bitmap, and $l[i]$ is the length of the $i^{th}$ run whose length exceeds 2. Since $b$ is at most 32, and therefore the array of lengths cannot be longer than 32, it is possible to unroll the loop for this computation using a switch statement with 32 entries. This optimization enables the fast computation of the block span, which is an essential component of the decompression process.

### 5.2.3 Decompression

Recall that for a certain attribute, an archive contains multiple compressed blocks, which in turn contain multiple RLE-subblocks. During runtime, user queries are evaluated using an index, which returns the row positions that match. Therefore, the query processor receives as input a number of row positions from which it determines which blocks require decompression.

We examine two block decompression strategies: a full decompression and a partial decompression. The latter approach decompresses only the RLE-subblocks that contain parts of the answer set. Intuitively the first approach is better when a block contains many results; the second approach when only few results need to be retrieved.

\(^2\)Modern compilers, such as GCC, offer the intrinsic function `__builtin_bitcount(unsigned int)` for using POPCNT in C. The `-msse4` compilation flag instructs the GNU GCC compiler to use the POPCNT instruction.
5.2 Algorithm 89

We start by providing more details on the two decompression strategies. Later, we describe an adaptive strategy that intelligently chooses the approach that leads to faster decompression time.

**Full-Decompression (FD)**

Assume that a compressed block is marked for decompression because it contains results for a given query. The block contains \( m \) data items each with size of \( n \) bytes and the index shows the row positions \( r_1, \ldots, r_k \) \( (k \leq m) \) within the block. The Full-Decompression strategy is shown in Figure 5.7. It decompresses the entire block (i.e. all RLE-subblocks) which is then stored in column-major order. Block transposition back to row-major order does not need to be performed, because the required data rows can be retrieved directly, as we explain below.

![Figure 5.7: Full-Decompression (FD) decompresses the entire set of RLE subblocks and then uses the row positions to compute the indices of the bytes that need to be retrieved in the uncompressed matrix stored in column-major order.](image)

Consider each decompressed archive block as an \( m \times n \) matrix. Each required row position \( r \) can be retrieved by accessing the bytes at the offsets \( r, r + m, \ldots, r + (n - 1)m \), since the offset between the bytes that belong to a certain row is exactly \( m \).
An example of the retrieval procedure is given in Figure 5.8, which on the left illustrates a block with three IPv4 addresses and 4 bytes per row. Because the decompressed matrix is in column-major order, as shown in the right of Figure 5.8, to access a row \( r \) at position \( x \), the decompressor traverses the data every 4 bytes at offsets: \( x, x+4, x+8, x+12 \).

**Figure 5.8: Byte traversal technique: the row at position \( x \) from a \( 3 \times 4 \) matrix stored in column-major order is retrieved by accessing the four bytes at the offsets \( x, x+m, x+2m, x+3m \).**

### Partial-Decompression (PD)

The Full-Decompression decompresses all RLE-subblocks, even if they contain no results. Partial-Decompression decompresses only subblocks that contain query results. An illustration of this approach is given in Figure 5.9.

The partial decompression algorithm first calculates the set of offsets \( O \) in the matrix (stored in column-major order) that should be accessed to retrieve the requested rows. The offsets \( O \) are used to determine whether an RLE-subblock (\( V \)- or \( B \)-block) must be decompressed: a subblock \( b \) that compresses the matrix byte offset \( i \) to \( j \) needs to be decompressed if there is an offset \( k \in O \), where \( i \leq k \leq j \). The decompressor can efficiently find which subblocks need to be decompressed using the procedure we outline below. After decompressing the required subblocks, the requested rows are extracted using the same byte-traversal technique described for the Full-Decompression. The process described above is summarized in Algorithm 4.

**Practical implementation of Partial-Decompression:** A naive approach to determine which RLE-subblock needs to be decompressed is to test, for each subblock, whether it contains at least one byte of the rows in the query (line 6 of Algorithm 4). For this purpose, we need to compute the *subblock range,*
5.2 Algorithm

**Compressed block:** 13 subblocks

```
1 2 3 4 5 6 7 8 9 10 11 12 13
```

**Un-Compressed block (n=4):**

```
1 3 5 8
2 4 6 9
10
```

**Requested rows:**

```
192 168 0 10
10 155 0 5
```

**Figure 5.9:** Partial Decompression (PD) decompress only RLE subblocks containing at least a byte that is required for retrieving the requested rows.

**Algorithm 4**  PartialDecompress($\text{Rows}, B, n$)

Require: A list of row positions $\text{Rows}$

- A compressed block $B$
- The size $n$ of each attribute value

Ensure: The attribute values stored in the requested row positions

1: $O := \emptyset$  // start with an empty set of offsets
2:
3: /* Retrieve the offsets from the row positions */
4: for all $r \in \text{Rows}$ do
5:   $O := O \cup \text{getOffsets}(r, n)$
6: end for
7:
8: /* Decompress only the subblocks containing at least a byte of interest */
9: for all $b \in \text{SubBlocks}(B)$ do
10:   $(i, j) := \text{subBlockRange}(b)$
11:   if $\exists k \in O : k \leq j \land k \geq i$ then
12:     decompressSubBlock($b, i, \text{output}$)
13:   end if
14: end for
15:
16: /* Byte-traversal technique as in Figure 9 */
17: return retrieveRowsFromFileMajor($\text{Rows}, \text{output}, n$)
which is the range of byte offsets stored in an RLE subblock. If the span of a subblock is $\sigma$ and the sum of the spans of the preceding subblocks is $S$, then the range of a subblock is $[S+1, S+\sigma]$. Recall that the RasterZip encoding allows to efficiently compute the span of a subblock. In order to test whether a subblock range contains a byte of a queried row, the naive approach would require to map the offsets of the queried row bytes into the transposed matrix and then to test each subblock. If at least one of the new offsets lies in the range between $S+1$ and $S+\sigma$, then the corresponding subblock should be decompressed. This naive strategy is very simple, but requires many comparisons. In practice, it leads to very slow performance. Even if only a small fraction of rows have to be extracted, the decompression speed is considerably worse than performing a full-decompression.

We next describe a practical and efficient implementation of Algorithm 4 that uses a fixed amount of memory to store the set of offsets and substantially reduces the number of needed comparisons. The main idea of the implementation is to represent the set of offsets as a bitmap instead of a list of integers. Each bitmap entry corresponds to a certain range of bytes. A bitmap entry is set to 1 if and only if at least one byte in the corresponding range belongs to one of the queried rows. In the next paragraph, we describe the technique in more detail.

The transposed $m \times n$ matrix (in column-major order) is partitioned into $s$ fixed size cells, where $s$ is a small constant. Intuitively, a grid of $s$ slots is logically superimposed on the matrix. A bitmap of length $s$, called grid bitmap, is used to mark each cell that contains at least one byte that is part of the query result (see Figure 5.10 where $s = 16$). Given the set of rows to be retrieved, the bits that need to be set to 1 can be determined very efficiently. First, the offsets of the desired row bytes are mapped into the transposed matrix. The mapped offsets are (still) in the range $[1, m \cdot n]$. Then, by linearly rescaling the offsets in the range $[1, s]$ we find the positions of the grid bitmap that need to be set.

Given the grid bitmap, it can be determined efficiently whether a subblock needs to be decompressed: its subblock range simply needs to be linearly rescaled as well by a factor of $s/(mr)$. If any bit is set in the grid bitmap in this rescaled range, the block needs to be decompressed. This process can exploit the bit counting assembly instruction and does not need any comparisons. The number $s$ of cells is a tunable parameter that allows to trade off precision for performance. More detailed grids (i.e. with larger values of $s$) are more precise in estimating the subblocks to be decompressed, but require more
5.2 Algorithm

Figure 5.10: A Grid Bitmap is used to partition the uncompressed \( m \times n \) matrix into 16 distinct cells \((s = 4 \times 4)\). Given a query, the bit corresponding to each cell is set to 1 if there is at least a byte in the cell that has to be retrieved from the compressed data. A subblock has to be decompressed if and only if it overlaps with a cell that has been marked with a 1 in the corresponding Grid Bitmap.

space to be stored, and consequently, are slower to query in average due to cache misses.

5.2.4 Adaptive Block Decompression

We have so far described two algorithms for block decompression. In this section we examine how to adaptively choose between the two algorithms at runtime. Partial decompression is preferred when a small fraction of RLE-subblocks need to be decompressed. Otherwise, the cost for computing the span of multiple RLE-subblocks makes full decompression a better strategy.

We introduce an adaptive decompression component that uses two runtime parameters, namely the selectivity of a query (within a block) and the compression ratio of a block, to predict which decompression strategy will
result in faster decompression. Below, we describe how the two parameters relate to the decompression strategy:

- **Selectivity**: If the selectivity within a block is low (i.e., a large fraction of rows have to be retrieved), then many RLE subblocks need to be decompressed. In this case, full-decompression is preferable. In contrast, for high selectivity queries, a small fraction of subblocks is decompressed. In this case partial-decompression is faster.

- **Compression ratio**: The compression ratio is defined as the ratio between the size of the compressed data and the size of data before compression [107]. When the compression is better (lower compression ratio), then a block is compressed into fewer RLE-subblocks, and a larger fraction of the subblocks needs to be decompressed. Therefore, better compressed blocks benefit more from full decomposition. Conversely, when RasterZip achieves worse compression ratios, a small fraction of RLE-subblocks have to be decompressed.

The selectivity of a query and the compression ratio of a block largely determine the fraction of RLE-subblocks that need to be decompressed and therefore can be used to predict which of the two strategies will perform better. In addition, the selectivity \( s \) and compression ratio \( c \) can be **efficiently computed at runtime before accessing the compressed subblocks**. We exploit these two parameters to build an adaptive decompression component shown in Figure 5.11. The adaptive decompression component accepts as input a compressed block \( b \) and a set \( R \) of row positions and, then, it uses a model to decide if partial or full decompression will provide a lower response time.

Given a block \( b \), the compression ratio can be efficiently computed from the block header, which stores the size of the block before and after compression (\( Unc_{SIZE} \) and \( Comp_{SIZE} \) in Figure 5.11). The selectivity is computed by dividing the number of rows to be decompressed by the total number of attribute values packed in \( b \). The adaptive decompression component, chooses between the two strategies using a simple and yet effective model built from profiling RasterZip. The model is a \( l \times l \) matrix, where the two dimensions are selectivity and compression ratio. The value in a cell indicates the decompression strategy that is expected to provide faster decompression for the corresponding range of selectivity and compression ratio values.

A profiled model is data- and attribute-agnostic: once built using real or synthetic data, it can be applied for different datasets and queries. To profile RasterZip we executed queries of different selectivity on compressed traffic flow records and measured the time required to decompress the queried rows.
5.3 Evaluation

NET-FLi, the architecture described in the previous chapter, represents the ideal environment for evaluating RasterZip, as it provides built-in bitmap indexing functionalities and offers a stream based record reordering technique that i) significantly boosts the LZO performance both in terms of compression ratios and decompression speed, ii) creates the data patterns that RasterZip is able to exploit. We plug RasterZip into NET-FLi to compare it against the fastest Lempel-Ziv compressor (LZO). RasterZip operates at equivalent com-
Figure 5.12: A 10 × 10 grid model built using a profiling dataset for deciding when to decompress using Partial-Decompression (PD) or Full-Decompression (FD). We indicate with N/A, cells that were not filled using the profiling dataset. Partial-Decompression is preferable for blocks that exhibit low compression ratios and/or high block selectivities.

pression speeds, produces more than 22% smaller archives, and allows our architecture to provide up to 20% better response times in the large majority of the queries.

All experiments have been conducted 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 the compressed archives on a 320 GB desktop hard drive.3

As input to the system we provided uncompressed flow traces stored on a commodity solid state drive.4 The drive provides a sustained reading speed of 170 MB/s, which corresponds to more than 5 Million flows/second (f/s). The system has been configured to fetch flows sequentially from the solid

3 The hard drive is a 7200 rpm Hitachi HDP725032GLA380 equipped with 8 MB of cache. The system is capable of performing cached reading at 2400 MB/s and unbuffered disk reads at 80 MB/s (measured with hdparm).
4 Intel X-25M G1, 80 GB model
state drive and to store both indexes and compressed columns to the mechanical desktop hard drive. This simple setup allows us to reproduce flow rates that can only be observed in large ISP networks. We use two data sets 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).

Each flow record include the following attributes: source and destination IP addresses and L3 ports, the L3 protocol, TCP flags, source and destination Autonomous System (AS) number, number of packets, number of bytes, the time of the first packet and the duration of the flow.

Information about the datasets are summarized in Table 5.1. The GZIP column reports the storage footprint when compressing the raw data with the gzip utility.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># flows</th>
<th>Length</th>
<th>Raw Size</th>
<th>GZIP Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>HE</td>
<td>231.9 Million</td>
<td>6 days</td>
<td>6.9 GB</td>
<td>2.5 GB</td>
</tr>
<tr>
<td>PN</td>
<td>1.2 Billion</td>
<td>62 days</td>
<td>37 GB</td>
<td>8.1 GB</td>
</tr>
</tbody>
</table>

**5.3.1 Compression Ratio**

In Table 5.2, we report the size of the archives compressed with RasterZip and LZO. First, we note that due to the approximate flow reordering scheme, LZO, which is optimized for compression speed rather than for achieving high compression ratios, almost matches the compression ratio of GZIP (shown in Table 5.1). By properly exploiting the prefix structure of partially reordered traffic records, RasterZip can further reduce the storage space footprint for both datasets. In fact, we observe that RasterZip uses 22% and 24% less space than LZO for the HE and PN datasets, respectively.

**5.3.2 Insertion Rate**

Our system has been designed for compressing and indexing high-speed streams of flow records in real-time. RasterZip aims at reducing the disk usage
Table 5.2: Storage space requirements for different compression algorithms.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>LZO</th>
<th>RasterZip</th>
</tr>
</thead>
<tbody>
<tr>
<td>HE</td>
<td>2.6 GB</td>
<td>2.0 GB</td>
</tr>
<tr>
<td>PN</td>
<td>8.2 GB</td>
<td>6.2 GB</td>
</tr>
</tbody>
</table>

of compressed archives and at supporting very high insertion rates. Having showed that RasterZip reduces the disk usage of two reference compressors, we evaluate the insertion rate it can support. For this purpose, we used the two datasets and compared the insertion rate our archive achieves when data columns are compressed on-the-fly using LZO and RasterZip.

Table 5.3: Record processing rates when using RasterZip or LZO for compressing data columns.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>LZO</th>
<th>RasterZip</th>
</tr>
</thead>
<tbody>
<tr>
<td>HE</td>
<td>474K f/s</td>
<td>471K f/s</td>
</tr>
<tr>
<td>PN</td>
<td>513K f/s</td>
<td>512K f/s</td>
</tr>
</tbody>
</table>

As shown in Table 5.3, RasterZip realizes almost identical insertion rates as LZO, which is the fastest known compressor of the reference Lempel-Ziv family. The average insertion rate is close to 500 thousand flows/second (f/s) for both datasets regardless of the compression algorithm used. To put this number into perspective, medium-large networks exhibit peak rates of 50 thousand flows/second. Therefore, we learn that RasterZip has substantially lower storage space requirements than LZO (and GZIP) and supports very high insertion rates.

5.3.3 Adaptive Decompression Selection

Next, we evaluate the adaptive decompression component to see if it can properly select between the full and partial decompression strategies at run-time.

Profiling stage: We constructed the model for the adaptive decompression component by profiling the decompression speed of the two decompression strategies using real network data and queries of different selectivity. Specifically, we created a set $P$ containing the distinct IP addresses present in one hour of traffic from the production network dataset (PN). Then, we performed a query for every element in $P$ and for every data block that was decompressed to answer the query we computed the following four metrics: the compression
ratio of the block (percentage), the selectivity of the query (percentage), and the time required to decompress the block using full and partial decompression. We used these metrics to profile a very compact grid model of size $8 \times 8$ bits as described in Section 5.2.4.

**Evaluation stage:** We evaluate the adaptive decompression component by comparing the block decompression time it provides with the decompression time offered by the full and partial decompression strategies. For this purpose, we issue the IP queries described above over a different time frame. In Figure 5.13 we illustrate the query response time using full, partial, and adaptive decompression. The figure on the bottom shows how the response time varies with selectivity for a fixed compression ratio (20%), while the one on the top shows how the response time varies with the compression ratio when the selectivity is fixed to 20%. The line corresponding to adaptive decompression is at the bottom for the vast majority of the queries. This illustrates that the adaptive decompression component effectively selects the decompression strategy that is faster. This is feasible simply by relying on a very compact model stored in a $8 \times 8$ bitmap. We use this model for the rest of our evaluation experiments.

### 5.3.4 Overall Performance

Finally, we measure the overall query response time of the archives compressed using RasterZip with the adaptive decompression component enabled. We compare the response time of this setup with the response times that the system offers when LZO is used for data compression.

To evaluate the response times, we query for flow records using ports corresponding to well-known applications.\(^5\) For this purpose, we extracted the 311 ports listed in the file `etc/services` of the test machine and executed destination port queries over 7 days of each dataset (PN and HE). This corresponds to 2177 different queries. We ordered queries by the number of rows retrieved (the selectivity) and, for every query, we plot in Figures 5.14 and 5.15 the ratio between the response time when the query is executed over a RasterZip compressed archive and the response time for the same query executed over an LZO compressed archive. Values lower than one correspond to queries where RasterZip offers better response times than LZO.

RasterZip provides better query response time than LZO for more than

\(^5\)The Internet Assigned Numbers Authority (IANA) assigns port numbers to applications.
Figure 5.13: The Adaptive strategy chosen on-the-fly intelligently shifts between Full-Decompression and Partial-Decompression. Shown for a fixed compression ratio (bottom) and selectivity (top).

87% – 96% of the queries depending on the dataset (see Figure 5.14 and 5.15 for the PN and HE dataset, respectively). For high selectivity queries, the query response time is in the order of milliseconds. The few cases where RasterZip offers worse response time than LZO, correspond to: a) few fast running queries and b) extremely low selectivity queries, which are rarely posed by network administrators. Examples of the second type of queries
5.4 Summary

In this work we introduced RasterZip, the first high-performance compressor specifically designed for network traffic archives that exploits the prefix structure of partially reordered data and recent advances in stream indexing for providing fine-grained decompression granularity. RasterZip offers the following key features:

- A new RLE-based online compression encoding that carefully packs compressible and incompressible content, exploits features of modern com-

---

**Figure 5.14:** RasterZip over LZO query response time for the Production Network dataset (PN).

---

...can be observed on the right side of Figure 5.15. These points correspond to queries on just port 80 (HTTP protocol), which covers the majority of the network traffic in a hosting environment. We are not interested in optimizing our system for such scenarios, because network analysts, who are more interested in searching for specific network events, combine multiple attributes (e.g. port and IP addresses) to refine their searches.
puter architectures, and achieves compression speeds of more than half a million flows per second.

- A novel adaptive decompression strategy that intelligently shifts between full- and partial-decompression.

- 22% more compact archives and faster query response times, particularly for the challenging class of selective queries, than competitive approaches, while offering equivalent compression speed.

Even if our design is focused on traffic flow data, there are several domains and applications that demonstrate similar traits (strong data repetitions, bounded data-range, and shared prefixes), and, therefore, can also benefit from our compressor. For example, to ensure consistent service levels, data centers and cloud computing environments are equipped with hardware and software sensors constantly reporting environmental (system temperature) and performance measurements (CPU load) [53, 81]. Therefore, monitoring infrastructure observe monitoring streams that inherently contain numerous repetitions and shared prefixes.
Chapter 6

pcapIndex: indexing packets with legacy compatibility

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. For example, network-forensics operators require 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, for accelerating the process of reverse engineering of network protocols, 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.

In this chapter, we describe an index-based packet filtering architecture that enables fast packet searches within packet-trace repositories. Our filtering architecture has been designed with legacy compatibility in mind, and does not require existing packet-trace repositories to be re-encoded with a new packet-trace format. In addition, 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. We experience impressive speedups on packet-trace search opera-
tions: our experiments suggest that the index-enabled libpcap may reduce the packet retrieval time by more than 1100 times.

Even if our main focus is on libpcap, the design of our packet trace indexing architecture is of general applicability, and, in fact, it can be ported with limited efforts to other packet trace formats, like the ERF format introduced by Endace [48].

The contributions of this work are as follows:

- We highlight that BPF filters are ill-used for searching within packet-traces. In contrast, compressed bitmap indexes are ideally suited for this purpose because they can retrieve from disk desired packets without having to linearly scan traces.

- We design and implement pcapIndex: the first indexing scheme for packet-traces based on compressed bitmap indexes. pcapIndex extends the widely-used libpcap without modifying the existing API, so that every libpcap-based application can profit from indexing without need for reimplementation. We show that our indexing scheme reduces the response time by up to 3 orders of magnitude. The additional storage space footprint is on average less than 7 MBytes per indexed field and per GByte of trace.

- We compare three state-of-the-art bitmap index compression encodings, i.e., WAH, PLWAH, and COMPAX, in the context of packet indexing. Based on our experiments, we show that COMPAX, the proposed encoding, exhibits the best promise for indexing packet traces.

This chapter is structured as follows. In Section 6.1 we highlight the benefits of using indexing for accelerating the exploration of packet trace repositories and we explain the reasons why bitmap indexes are ideally suited for this task. Section 6.2 describes the proposed packet indexing architecture, which we evaluate in Section 6.4. We conclude the chapter with Section 6.5.

6.1 From Packet Filters to Indexes

The exploration of large-scale packet repositories is in many cases an iterative task that consists of posing searches and refining them in several steps until the desired information is found. This might involve accessing many packet traces and accessing the same trace multiple times.
Current packet-trace analysis solutions indistinctly rely on packet filtering mechanisms, for selecting packets from a packet trace or for filtering packets from a live packet stream. The de-facto reference packet filtering mechanism is the Berkeley Packet Filter (BPF), which we have described in Chapter 2. The packet filtering operations are costly to perform because they require the linear scan of the entire packet-trace with the subsequent filtering of packets that satisfy the search criteria. Filters are evaluated against each packet. This has the following implications: i) the entire packet trace has to be read even when only a few packets qualify the filtering criteria, and ii) all the packets have to be implicitly parsed by the BPF engine. In a packet trace analysis context, where a trace is usually accessed many times, this process is extremely inefficient. Surprisingly, current packet-trace analysis solutions do not provide support for external indexing schemes that can substantially speed up the exploration process.

This work presents the first indexing scheme for network packet traces based on compressed bitmap indexing principles. The concept of compressed Bitmap Indexes (BI’s) is ideally suited for speeding up packet filtering operations within packet traces. This is for the following reasons:

- BI’s are tailored for read-only data. When dealing with packet-trace data, information is only appended but never modified.
- BI’s are best suited for indexing numerical attributes and the vast majority of network protocol fields are numerical.
- BI’s do not require rebalancing (something that tree-based indexes require) and can perform fast bitwise operations even in the compressed domain.
- 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 [21], but using only the index structure. Cardinality queries allow frequency distribution plots of desired attributes to be computed without accessing the packet trace.
- Compressed BI’s are very compact in size [125] and can be memory efficient even for large packet-traces.
- Finally, compressed BI’s support very fast insertion rates, which is desirable for indexing packets from high-speed links.
6.2 Architecture

In this section we describe the architecture of pcapIndex, an indexing infrastructure that transparently provides indexing functionalities to libpcap-based applications. Existing applications written on top of the well known library can benefit from indexing by simply linking with the index-augmented libpcap variant we developed. pcapIndex can leverage both filtering technologies, index and BPF based, that are combined for providing a flexible two-stage filtering mechanism.

Before describing the architecture itself, we briefly overview the pcap trace format. 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.

As shown in Figure 6.1, pcapIndex is composed of two components: a packet indexer and a query processor. 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 \( \text{pkt-id} \). 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 \( \text{pkt-id} \) 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 6.1. 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-ids matching a query. The indirection array is then used to map pkt-ids 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.

The packet indexer and the query processor are described in detail in Section 6.2.1 and 6.2.1.

6.2.1 Packet Indexer

Figure 6.2 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-ids to offsets. The offset of a packet is equal to the cumulative size of the trace header,
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 6.2: The packet indexer is composed of a parser and a set of indexing plugins. It decodes packets, indexes attributes, and updates the indirection array.

the `pcap` headers, and the packets. For example, the 3rd packet in Figure 6.2 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 VLAN 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., Skype, retrieved by running Deep Packet Inspection (DPI) [5, 7] engines over the packet payload. 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.*.*).

6.2.2 Query Processor

In Figure 6.3 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 6.3 matches HTTP GET requests. Chaining it with the index filter of Figure 6.3 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.
Figure 6.3: PcapIndex query workflow. The query processor parses the Index filter, retrieves the right BI’s, joins BI’s, maps pkt-ids to packet offsets, reads matching packets, and finally applies the BPF filter.
6.3 Implementation

Each plugin computes a compressed bitmap of the positions of matching packets. For example, the \textit{(SrcIp)} plugin 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 \textit{pkt-ids}.

6.3 Implementation

PcapIndex extends the \textit{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 \texttt{+} character. For example, the call:

\begin{verbatim}
pcap_t = pcap_open_offline("test.pcap+SrcIp=10.10.10.10")
\end{verbatim}

opens the pcap trace \textit{test.pcap} and creates an array of \textit{pkt-ids} of the packets matching the filter \textit{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 \texttt{".off"} suffix (in this example \textit{test.pcap.off}). Once the array of matching \textit{pkt-ids} is created, e.g., the packet identifiers 200, 10010, and 10500 in Figure 6.3, the trace traversal process can start. This is performed by calling \textit{pcap\_next\_ex}, which is the standard libpcap function for traversing a trace:

\begin{verbatim}
pcap\_next\_ex(pcap_t *, struct pcap\_pkthdr **, u\_char **)
\end{verbatim}

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 \textit{pkt-ids} 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 \texttt{+} character.

In Algorithm 5 we show the pseudo-code of our implementation for the \textit{pcap\_next\_ex} function. We use a counter, which is initialized to zero when a trace is opened, as a cursor in the \textit{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.

\begin{algorithm}
\caption{pcap\_next\_ex(pcap\_h)}
\begin{algorithmic}[1]
\Require an open pcap handle \texttt{pcap\_h}
\While{\texttt{has\_results}}
\State /* Seek to the packet */
\State \texttt{pktId} = \texttt{pcap\_h.PacketID[pcap\_h.cursor]};
\State \texttt{pktOffset} = \texttt{pcap\_h.PacketOffsets[pktId]};
\State \texttt{seek(pcap\_h.tracefile, pktOffset)};
\State /* Read the pcap header and the packet itself */
\State \texttt{hdr} = \texttt{read\_pcap\_header(pcap\_h.tracefile)};
\State \texttt{pkt} = \texttt{read\_packet(pcap\_h.tracefile, hdr.caplen)};
\State \texttt{pcap\_h.cursor +=1};
\State /* Execute the BPF filter, if configured */
\If{(!\texttt{has\_bpf(pcap\_h)})}
\State \Return \langle \texttt{hdr, pkt} \rangle;
\ElsIf{\texttt{(bpf\_exec(pkt, pcap\_h.bpf)}}}
\State \Return \langle \texttt{hdr, pkt} \rangle;
\EndIf
\EndWhile
\State \Return \texttt{NULL};
\end{algorithmic}
\end{algorithm}

\section{Evaluation}

We evaluate critical performance metrics of the index-enhanced \texttt{libpcap} library. In particular, we evaluate: 1) the storage 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 \texttt{pcapIndex}.

In the evaluation, we use two pcap packet traces captured in two distinct locations. The trace \texttt{trace1.pcap} has been captured at the border gateway of a university network, whereas \texttt{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 6.1 reports the total size of the indexes when using the different encodings.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>WAH</th>
<th>PLWAH</th>
<th>COMPAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace1</td>
<td>99 MBytes</td>
<td>65 MBytes</td>
<td>55 MBytes</td>
</tr>
<tr>
<td>trace2</td>
<td>89 MBytes</td>
<td>61 MBytes</td>
<td>51 MBytes</td>
</tr>
</tbody>
</table>

COMPAX indexes are the one having the lowest storage space footprint. 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 6.2 reports the measured time using a desktop hard drive and a solid-state drive for storing the packet traces and the corresponding indexes.

For the regular hard drive the indexing time is almost the same as the reading time: *I/O 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 I/O wait percentage high. Even with the faster solid-state drive, the I/O is the main bottleneck. Based on this observation, we highlight that because of the I/O 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 occurrences. 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-ids` 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 6.3 and 6.4 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.

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.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>read</th>
<th>WAH</th>
<th>PLWAH</th>
<th>COMPAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace1</td>
<td>7141</td>
<td>13424</td>
<td>13331</td>
<td>10583</td>
</tr>
<tr>
<td>trace2</td>
<td>5708</td>
<td>11522</td>
<td>11534</td>
<td>8768</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>read</th>
<th>WAH</th>
<th>PLWAH</th>
<th>COMPAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace1</td>
<td>21406</td>
<td>23796</td>
<td>23069</td>
<td>22380</td>
</tr>
<tr>
<td>trace2</td>
<td>17839</td>
<td>19648</td>
<td>19350</td>
<td>18761</td>
</tr>
</tbody>
</table>
Table 6.3: *Query response time (in milliseconds) with pcapIndex and BPF on a solid-state drive. The grey cells mark the speedup of pcapIndex.*

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Source ports</th>
<th>Top 200</th>
<th>Bottom 200</th>
<th>Source IP addresses</th>
<th>Top 200</th>
<th>Bottom 200</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>BPF</td>
<td>index</td>
<td>Speedup</td>
<td>BPF</td>
<td>index</td>
</tr>
<tr>
<td>trace1</td>
<td></td>
<td>7076</td>
<td>654</td>
<td>10.8×</td>
<td>7101</td>
<td>6</td>
</tr>
<tr>
<td>trace2</td>
<td></td>
<td>5714</td>
<td>983</td>
<td>5.8×</td>
<td>5746</td>
<td>47</td>
</tr>
</tbody>
</table>

Table 6.4: *Query response time (in milliseconds) with pcapIndex and BPF on a regular hard drive. The grey cells mark the speedup of pcapIndex.*

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Source ports</th>
<th>Top 200</th>
<th>Bottom 200</th>
<th>Source IP addresses</th>
<th>Top 200</th>
<th>Bottom 200</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>BPF</td>
<td>index</td>
<td>Speedup</td>
<td>BPF</td>
<td>index</td>
</tr>
<tr>
<td>trace1</td>
<td></td>
<td>21555</td>
<td>8925</td>
<td>2.4×</td>
<td>21532</td>
<td>28</td>
</tr>
<tr>
<td>trace2</td>
<td></td>
<td>17886</td>
<td>9142</td>
<td>1.9×</td>
<td>17870</td>
<td>128</td>
</tr>
</tbody>
</table>

for each query. In fact, the average execution time is very close to the time reported in Table 6.2 for reading all packets. *Disk I/O is therefore the main factor that determines the BPF response time.*

For high 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 low 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 low 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.

### 6.5 Summary

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 of existing pcap-based repositories. Our evaluation suggests that indexing packet traces imposes minimal disk overhead and provides impressive speedups, particularly when few packets need to be retrieved from large packet-traces, i.e., that is when searching for needles in a haystack. In addition, we compare the performance of three state-of-the-art compressed bitmap encodings on packet-traces and find that COMPAX has the best performance. Finally, we show 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 higher speedups.
Chapter 7

Conclusions

The Internet has become a global IT infrastructure providing ubiquitously accessible, interactive, and secure services used on a daily basis by a large fraction of the global population. To meet users’ expectations, network administrators require sophisticated monitoring infrastructures for detecting misconfiguration and faults, for measuring the performance, and for enabling timely reactions to security threats.

To gain insights into the actual status of production networks, current monitoring infrastructures rely on network probes that constantly monitor important network aspects. Therefore, large volumes of network measurement data are produced by monitoring infrastructures.

Stream processing is the data processing paradigm usually applied to analyze high-speed streams of network measurement data, as it allows a static set of stream properties to be captured from the data stream as it passes by, without requiring the entire stream to be stored on disk. However, there are situations where the exact storage of incoming data streams is required to enable long-term historical analyses of network measurement data. The collection of entire data streams is required in general for performing any analysis task that cannot be performed over a live stream, because the information to be captured is not known at the time the stream passes by. Advanced data processing systems are required for collecting high-speed streams of network measurement data.

In this research work, we have focused on fundamental technologies, namely indexing and compression, to enable the creation of advanced large-
scale network measurement data repositories capable of substantially reducing the data volumes while providing efficient retrieval mechanisms for extracting the records of interest from compressed data. We have shown that by introducing novel encodings carefully optimized for the data patterns usually present in network monitoring contexts, substantial performance gains can be achieved.

We have designed a new compressed bitmap index variant, called COM-PAX, and we have shown that it outperforms the current state of art technologies both in terms of compression ratios and lookup times when used for indexing attributes present in network flow records and packets.

We have explored new exciting avenues in the field of stream compression by proposing a stream based approximate sorting technique, called oLSH, that rearranges the order of multi-attribute numerical streaming records to substantially boost the performance of general purpose compressors and compressed bitmap indexes.

To exploit the pattern introduced by oLSH, we have introduced RasterZip, a novel lossless real-time compressor optimized for the network monitoring domain that provides fine-grained decompression granularity by leveraging indexes, and achieves higher compression ratios than the fastest general purpose compressor, LZO, while providing comparable compression rates and higher decompression performance.

The aforementioned indexing, approximate sorting and compression algorithms represent the foundation of NET-FLi, a software architecture designed for compressing and indexing high-speed streams of network flow records in real-time. Our architecture can cope with extremely high-speed flow records streams while i) compressing the data more than the widely used gzip and ii) producing indexes for the most commonly queried data attributes.

Additionally, we have shown that our architecture can be applied for augmenting the de-facto packet capture and trace analysis library, libpcap, with compressed bitmap indexes without breaking the legacy compatibility. We have evaluated pcapIndex, our index enhanced libpcap variant, when used to perform packet searches over large packet traces and we have shown that impressive speedups can be obtained, especially when just a few packets have to be retrieved.

Even if in our research we have focused on the networking domain, we believe that our contributions are of general interest and can be applied to a broad spectrum of applications challenged by high-speed streams of multi-attribute numerical records.
7.1 Future Work

Our work contributed to the creation of advanced compression and indexing schemes for network monitoring data. There are many directions for future work, including performance optimizations, the design of more advanced re-ordering techniques, and the exploitation of the information encoded in compressed bitmap indexes for assisting tasks such as visualization and anomaly detection. In what follows, we list promising directions for future work.

Support for custom flow records

While evaluating NET-FLi, 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.

Scale-out distributed architectures

NET-FLi is inherently parallel by design and, therefore, can be used as a back-end to develop scale-out distributed architectures based on clusters of commodity servers. In particular, we believe that the integration of our work into the Hadoop infrastructure is a promising research path.

Introduce self-tuning functionalities

NET-FLi can exploit modern multi-core processors to compress and index the data, in parallel. Architecture-specific tuning has been done to achieve high-performance. For example, the current NET-FLi implementation manipulates the CPU affinity settings [55] to bind specific threads to CPU cores in order to better exploit the CPU cache hierarchy, and, therefore, to improve the scalability. System-specific parameters, such as the NET-FLi data block size and the oLSH window size, can be tuned to achieve different trades-offs in terms of compression ratios, insertion rates, and response times. Further work is required to make NET-FLi adaptive and self-tuning. In particular, system-level profiling has be done to build cost models that take into account architectures-specific information and user’s preferences.
**Compressed bitmap indexes creation**

Creating compressed bitmap indexes for attributes present in high-speed streams of network traffic in real-time is feasible, but requires substantial CPU resources. We believe that further optimizations of the indexing performance are possible. In particular, heavy tailed distributions commonly found in network data attributes, such as IP addresses and port numbers, could be exploited to achieve higher cache locality, and therefore performance.

**Exploit compressed bitmap indexes for anomaly detection**

Our experiments show that the compressed indexes exhibit specific patterns depending on the traffic behavior. A promising research track is the utilization of the patterns visible in the compressed domain for proposing novel visualization and anomaly detection techniques.

**Attribute agnostic flow record reordering**

Currently, the flow record reordering scheme hashes flow records using the most frequently queried attributes, such as IP addresses and ports, to improve the performance. Further research is required for proposing more advanced reordering approaches that can adjust their behaviour dynamically, according to the processed data streams, for providing higher compression ratios or resilience to attacks (i.e., intelligent load shedding functionalities).

**Improve RasterZip encoding**

RasterZip represents both symbols and lengths with a single byte. However, the symbol and length values are often much smaller than 255 and, therefore, in many cases fewer bits can be used for the encoding. By allowing byte misalignments (i.e. by reducing the number of bits used to represent runs and lengths) the compression ratio can be improved. More advanced compressed data layouts could be proposed for encoding runs using techniques inspired by modern high-speed integer compression algorithms, such as Simple9 [18] and PFor [131], that have shown that high-decompression speeds can be achieved even without preserving the byte alignment.


**Introduce indexes for similarity searches**

Both NET-FLi and pcapIndex use indexes for accelerating exact queries (e.g., search packets having a specific IP address). Introducing the support of *approximate* searches between network measurement records would bring additional value for assisting the exploration of large-scale network data repositories. For example, the ability to search *similar* payloads in packet repositories would be particularly useful to reverse engineer proprietary or undocumented network protocols.

**Integrate packet and index filters**

In Chapter 6, we have shown that compressed bitmap indexes can substantially accelerate packet searches over packet traces. By chaining the standard Berkeley Packet Filter (BPF) with our index based filters, users can leverage both the performance of indexing and the flexibility of BPF. Indexing and BPF filters have to be explicitly specified by the user. Further work is required to create a unified filtering infrastructure that can transparently leverage indexes, if present, and BPF filters.

**Embed indexing information into packet traces**

The goal of pcapIndex, described in Chapter 6, is to provide indexed searches over unmodified packet traces encoded in *pcap* format. Therefore, the indexes are stored externally as files. Novel packet trace formats, or format extensions, could be proposed for embedding indexing information in the packet trace itself. In particular, we believe that the next generation of pcap encoding, *pcap-ng* [14], which is still an experimental and not fully defined standard, should accommodate for indexing as well.

### 7.2 Publications

The work presented in this thesis is based on the following publications:


*P02:* F. Fusco, X. Dimitropoulos, M. Vlachos, L. Deri, **pcapIndex: An Index**


In addition, these publications were coauthored during and before this thesis:


7.3 Patents

The research described in this dissertation led to three patents applications: two have been filed at the United States Patent and Trademark Office (US201200054160 and US201200054161), and the third is filed at the European Patent Office.
Bibliography


This dissertation is based on the research work that I have carried out as a Pre-doctoral researcher at the IBM Zurich Research Laboratory, Switzerland. However, this work started long before I joined IBM Research and it would not have been done without the help of many bright people I had the pleasure to discuss and work with in my entire research career. Nevertheless, I would like to express my gratitude to some people in particular.

First of all, I would like to express my sincerest gratitude with my thesis supervisor, Prof. Berhard Plattner, who believed in me and gave me the opportunity to join the Communication Systems Group at ETH Zurich as an external PhD student. I would like to thank all the group members and Dr. Xenofontas Dimitropoulos in particular, for the time spent in teaching me how to present scientific results and how to judge research contributions.

I am also grateful to many colleagues at IBM Zurich Research Laboratory for the discussions on so many and so diverse research topics. Among them, I would like to thank my manager Dr. Douglas Dykeman who trusted me and my hard-working attitude, and with whom I spent many many hours in the lab. Endless gratitude goes to Dr. Michail Vlachos for guiding me in this research journey and for spending his precious time discussing about database technologies.

Endless gratitude goes also to Dr. Luca Deri, the one who introduced me into the world of network monitoring research. I sincerely thank him for the exchanges we had, on a regular basis, over the last decade.

I also would like to thank my former colleagues at Endace Limited. Among them I would like to thank Stuart Wilson for showing me the difference between science and religion.

Finally, I would like to thank the most important person, who has tolerated my long absences while I was thinking of bits and bytes.
Curriculum Vitae

Francesco Fusco was born in Grosseto, Italy on May 19, 1982.

Educational Curriculum Vitae

2010-2012  Doctor of Sciences
Swiss Federal Institute of Technology (ETH), Zurich

2004-2008  Dottore Magistrale in Technologie Informatiche
University of Pisa, Italy

2001-2004  Dottore in Informatica
University of Pisa, Italy

Work Experience

2009-2012  Pre-Doctoral Researcher
IBM Research, Zurich, Switzerland

2008-2009  Software Engineer, Network Specialist
Endace Ltd., Hamilton, New Zealand

2004-2008  Software Engineer, Network Specialist
Frelancing projects with:
     NEC Europe Ltd. Network laboratories
     Agilent Labs
     Mutina Technology (now Empirix)