CompactFlow: A Hybrid Binary Format for Network Flow Data

Michal Piskozub, Riccardo Spolaor, Ivan Martinovic

CompactFlow: A Hybrid Binary Format for Network Flow Data

Michal Piskozub, Riccardo Spolaor, and Ivan Martinovic

University of Oxford, UK
{michal.piskozub,riccardo.spolaor,ivan.martinovic}@cs.ox.ac.uk

Abstract. Network traffic monitoring has become fundamental to obtaining insights about a network and its activities. This knowledge allows network administrators to detect anomalies, identify faulty hardware, and make informed decisions. The increase of the number of connected devices and the consequent volume of traffic poses a serious challenge to carrying out the task of network monitoring. Such a task requires techniques that process traffic in an efficient and timely manner. Moreover, it is crucial to be able to store network traffic for forensic purposes for as long a period of time as possible.
In this paper, we propose CompactFlow, a hybrid binary format for efficient storage and processing of network flow data. Our solution offers a trade-off between the space required and query performance via an optimized binary representation of flow records and optional indexing. We experimentally assess the efficiency of CompactFlow by comparing it to a wide range of binary flow storage formats. We show that CompactFlow format improves the state of the art by reducing the size required to store network flows by more than 24%.

Keywords: Networks · Binary Format · Cybersecurity Data Processing

1 Introduction

In recent years, we have witnessed an astonishing evolution of networks in terms of complexity, variety, and versatility. An increasing number of devices have started to embed networking capabilities and to require Internet connection to provide their full functionalities. Hence, guaranteeing the connectivity of such devices has become fundamental to the operation of the entire networking infrastructure. In order to carry out this task, network administrators have to be provided with reliable tools to monitor traffic flowing through a network. In addition to that, administrators have to be able to investigate past events by retrospectively analyzing the state of a network at any given point in time. For this reason, it is necessary to archive network traffic in a fast and space-efficient way.

Monitoring networks at the granularity of packets offers perfect visibility of their state but also requires overwhelming computing resources and storage space to be devoted. While packet-level approach may have been possible in

the early days of networking, it is infeasible in modern networks due to the increasing number of interconnected devices and the volume of data produced by them. Moreover, the ubiquitous adoption of encryption in network communication to protect user privacy has made packet-level traffic capturing obsolete since encrypted payloads do not provide any meaningful information. Due to these limitations, network monitoring has shifted toward a network flow as a more coarse-grained representation of traffic data. A network flow comprises information of a communication from a temporal perspective as a five-tuple: protocol, source and destination IP addresses and ports. Differently from packet-level data, flows capture only metadata, such as the overall number and size of exchanged packets.

Flow exporters are devices in the flow creation process that capture and assign network packets to flows based on their five-tuple and within a temporal interval. Once flows are created, they are sent to a flow collector using a given export protocol. A flow collector is a device in charge of storing flow data for future use. The most popular export protocol is Cisco's NetFlow [9], which inspired the creation of the open standard IPFIX [8].

Over the years, a number of flow collectors have been proposed by networking companies and researchers. The main goal of such devices is to rapidly collect and store flows in such a way that avoids blocking the next oncoming flows. More importantly, they have to adopt a storage format that is efficient in terms of the size required and indexing to process future queries. Network administrators are constrained by the space available to store network traffic, thus older traffic has to be periodically deleted. For this reason, a space-optimized format saves storage space, which allows for keeping network traffic of longer periods for retrospective analysis. Unfortunately, our investigation of open-source flow collectors showed that they use an inefficient flow representation in their formats, even among the ones that favor storage efficiency over processing speed.

In this paper, we present CompactFlow - a binary format to represent network flows that favors storage and processing performance while supporting indexing. In particular, the CompactFlow format relies on dynamic field sizes and is based on a linked list to store the contents of flows. This accounts for a significant reduction of storage size. In fact, experimental results show that CompactFlow files are on average almost 3 times smaller than the ones using binary formats of other flow collectors, and 24% smaller than the ones using the binary format of the state-of-the-art System for Internet-Level Knowledge (SiLK) collector [29].

CompactFlow can be considered a hybrid binary format since it allows for customization according to administrators' analysis purposes: (i) it supports additional indexing methods to increase the speed of repetitive queries, and (ii) it is possible to choose which flow fields or which specific values of a flow field to index. Unlike database-based approaches, our solution allows for high-speed saving of flows without the risk of dropping them or resorting to sampling since the indexing can be done after successful storage. The design principles of CompactFlow join two best practices of storage (binary files) and querying

(indexing), to have a robust system for network monitoring and processing of cybersecurity events.

Contributions The contribution of this paper is twofold:

- We present a binary file format to store network flows using less space than state-of-the-art approaches. Our format supports popular indexing methods to allow faster data processing in the security context.
- We perform a thorough analysis of all open-source network flow collectors and a popular data serialization library by analyzing their binary formats.

Organization The remainder of the paper is organized as follows. In Section 2, we survey the state-of-the-art techniques for network traffic monitoring. We present our CompactFlow format in Section 3 while we evaluate and compare its performance to other formats in Section 4. In Section 5, we discuss the results. Finally, we present conclusions in Section 6.

2 Related Work

In the last two decades, many approaches have been proposed to monitor network traffic. This effort has been necessary to carry out management and security analyses on networks, such as identification of anomalies or failures, and detection of attacks. Most of these analyses cannot be done in real-time, hence network traffic has to be stored in persistent memory in order to make it available when needed. For this reason, it is necessary to store and query network traffic efficiently. A first important distinction between storage approaches is related to the granularity of traffic collection: packet- and flow-level.

2.1 Packet-level Traffic Collection

Collecting network data at packet-level provides fine-grained information about traffic but it requires fast dedicated equipment. Desnoyers et al. in [12] propose Hyperion, a system that relies on a log-structured file system that is optimized for writing data streams to store packet-level network traffic. This system indexes data stream segments via distributed multi-level Signature indexes. The authors claim to be able to write and index up to 1M and 200K packets per second, respectively.

Maier et al. in [25] propose to focus only on the part of the packet stream that may be interesting for a network intrusion detection system (NIDS). Hence, they present the TimeMachine system which applies a cut-off heuristic (i.e. it only considers the first N bytes) to reduce the size of the data stream to store.

Fusco et al. in [15] present PcapIndex which extends *Libpcap* by supporting rapid packet filtering via COMPAX compressed bitmap index [14]. Doing this, PcapIndex reduces the disk overhead and the response time of queries.

Unfortunately, the aforementioned methods are not suitable for large-scale networks since they do not scale on the number of devices connected. Moreover, such fine-grained information would require an overwhelming storage capacity.

2.2 Flow-level Traffic Collectors

In order to cope with the shortcomings of packet-level traffic collection, the networking community has moved toward the collection of traffic information by aggregating packets into flows. Compared to packet-level one, flow-level network traffic is more privacy-preserving (i.e. packets are aggregated), and more scalable over the amount of traffic and number of connected devices in modern networks. The first standard for exporting network flow information was NetFlow [9]. Initially, Netflow version 5 was released by Cisco in 1996 and then extended to version 9 in 2004. Subsequently, the Internet Engineering Task Force (IETF) in 2013 released the IP Flow Information eXport (IPFIX) Internet Standard [8] which is a further enrichment of NetFlow v9. In what follows, we present various solutions available to collect, store and access network flows.

Storage Formats Network flow collectors adopt several solutions to store flow-level network traffic in persistent memory. Such solutions can be divided according to the way they structure and index the data [20]. A popular data structure to store flows is a database. The advantage of databases is that such data structures automatically handle the information storage and indexing via a DataBase Management System (DBMS). Traditional DataBase Management Systems, such as MySQL and PostgreSQL, store the information by rows (*row-based databases*). In our case, a row represents an entire flow (i.e. all its fields). Two examples of flow collectors that use a row-based database to store network traffic are Vermont and pmacct. Regarding the queries, row-based databases offer good flexibility but they have poor performance in terms of data retrieval and new flow insertion time. Moreover, a row-based database is not storage-efficient since it requires considerable indexing.

For this reason, column-based databases have been proposed for network flow storage. Rather than to consecutively store entire flows, column-based databases store them in columns by flow fields. Examples of column-based databases are MariaDB ColumnStore [3] and bitmap indexing methods (e.g. FastBit [31], and COMPAX [14]). Indexing by columns decreases data retrieval time for queries while maintaining good flexibility and moderate insertion time. In particular, FastBit is an order of magnitude faster than MySQL [11]. IPFIXcol is a collector that relies on FastBit. It supports IPFIX, bidirectional flows, and variable length fields. Unfortunately, the main shortcoming of column-based databases is poor performance in retrieving flows in their entirety. Moreover, such databases still have to maintain a reference to a specific flow (i.e. index) for each flow field resulting in overhead in storage size.

Another solution that aims to reduce storage space is to rely on *flat files*. A flat file typically stores data sequentially and does not embed any hierarchy nor indexing by default. For this reason, flat files do not offer query flexibility but they occupy much less space than a database [19]. Data in flat files can be represented in a text or binary format. Despite the portability of a flat file in text format, representing data in a binary format further reduces the storage

	Storage Formats				
	Database		Flat Files		
Collectors	Row-based	Column-based	Binary format	Text format	Bidirectional
Argus [4]	✓		✓	✓	✓
flowd [1]			✓		
IPFIXcol [30]		✓			✓
nfdump [17]			✓		
pmacct [24]	✓			✓	✓
SiLK [29]			✓		
Vermont [21]	✓				✓

Table 1. Comparison of open-source flow collectors.

size and the query response time. Examples of flow collectors that save network traffic in a binary format are Argus, flowd, nfdump, and SiLK.

In Table 1, we report several open-source network flow collectors and we compare them according to storage formats supported and whether they can represent bidirectional flows. It is worth noting that some collectors can use more than one storage format (e.g. Argus, and pmacct) and that the majority use flat file formats.

To perform a thorough comparison between our proposal and the state of the art, we analyze the binary formats used by collectors that allow storing network flows in flat files (i.e. Argus, flowd, nfdump, and SiLK). In Section 4, we show that our compact format outperforms all of them in terms of space efficiency.

Indexing Methods Flat file formats are optimal for network flow storage because they save space and have a negligible computational overhead in inserting new flows. However, the limitation of this format is that it does not provide integrated indexing of the flows, thus it lacks in query performance and flexibility. To cope with this shortcoming, researchers propose solutions to build indexes which offer a low retrieval time and require little storage. Typically, storing network flows from a flow exporter consists of two aspects: writing the flows to a flat file and building indexes of those flows. For this reason, most solutions rely on the multi-processing capabilities of modern computers [13, 22].

In the literature, researchers use different data structures to organize flow (or query) indexes [7]. For example, TelegraphCQ [5] stores indexes and results of queries via a modified version of PostgreSQL. Three other examples, GigaScope [10], MIND [23], and FloSiS [22] arrange indexes into trees, multi-level hashing tables, and Bloom filters, respectively. However, the most popular and best-performing approach leverages bitmap indexing. As an example, Reiss et al. in [28] and Chen et al. in [6] applied the concept of bitmap indexing (i.e. FastBit [31]) to improve the performance of TelegraphCQ [5] and TimeMachine [25] (applied on network flows), respectively. More recently, Xie et al. in [32] present Index-trie, a novel data structure to index flows that combines trees and bitmaps.

Our CompactFlow format is designed to be hybrid. This means that it is able to support a variety of flow indexing methods.

Several approaches also propose fast compression/decompression algorithms to be applied to both stored flow data and indexing data structures to further reduce the storage size. Fusco et al. propose NET-Fli [14] and RasterZip [16] systems that compress on-the-fly flow data streams via compression algorithms, e.g. Lempel-Ziv-Oberhumer (LZO) [26], and Run-Length Encoding (RLE) [18]. Unfortunately, even the fastest compression algorithms generate computational overhead which translates to increased processing time.

3 CompactFlow Format Design

Network flow data is comprised of information relating to communication between two hosts on a network. Every flow consists of core fields (traditionally called a five-tuple) and additional fields that contain volumetric and temporal information pertaining to communications. A five-tuple includes protocol, source IP address, source port, destination IP address and destination port. The basic additional fields of a flow are the timestamp of the first packet, the overall duration, the number of packets and the total size of packets.

In this section, we introduce CompactFlow, a new format specifically designed to provide more efficient storage and fast processing of network flow data. Our proposal relies on a new binary file format, which supports both unidirectional and bidirectional flows. In Figure 1, we show the general structure of a CompactFlow file. In what follows, we describe all of its components and discuss our design choices.

3.1 CompactFlow File Header

A CompactFlow file is structurally divided into the header and body. The header stores information about the format (i.e. binary file marker, format version, and byte order) and contained flows (i.e. type, IP version, number of flows, and timestamp) that are later encoded in the body of the file. The header of a CompactFlow file contains a *Marker* as a first field by which the format can be recognized. We designed its value to be 0x00434600, where the inner bytes represent characters 'C' and 'F' and the outer bytes are non-character values to avoid being misinterpreted by applications for text files. The *Ver* value represents a version of the CompactFlow specification with the first byte being major and the second minor versions (e.g. 0.3). The *T* value stands for the type of flow in terms of its direction (unidirectional or bidirectional). Since CompactFlow supports both IPv4 and IPv6, the version of IP is given by the *IP* parameter. By design, IPv4 and IPv6 flows are stored in separate files. The *Flow Count* value stores the total number of flows contained in the file. Instead of storing complete timestamps in each flow, we only save the *Timestamp* (down to a precision of an hour) in the header since each file represents up to one hour of traffic. This allows for each flow record to store only the added time to that timestamp to

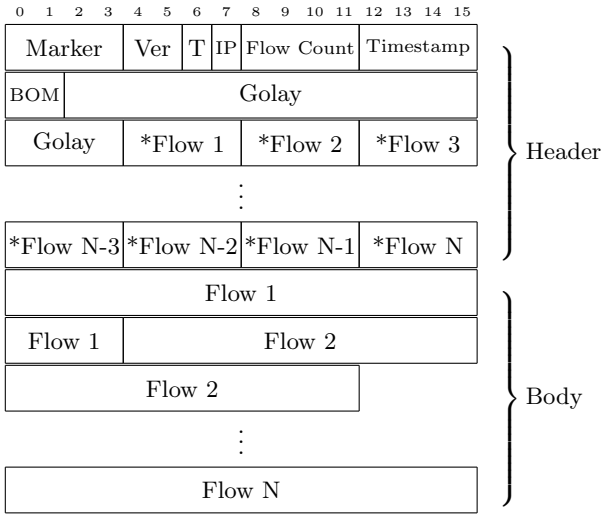


Fig. 1. CompactFlow file format (in bytes).

reconstruct it fully. The *Byte Order Mark* (BOM) is used to clarify that the CompactFlow file uses big-endian encoding. This decision is justified by the fact that big-endian order is used by default in network communications, in fact to such an extent that it is often referred to as the network order.

All fields described above constitute mandatory data in the header. Since those fields carry high importance to the remainder of the file, we use the *extended binary Golay code* (G_{24}) to detect and correct errors in them in the case of corruption. Such code allows us to recover 3 bits for each 12-bit word at a cost of doubling the size of data. Fortunately, we can afford it since the header constitutes a minimal, almost negligible, percentage of the size of the whole file.

Our proposal uses dynamic field sizes to store flows, thus flow records can vary in size. This means that given a current flow it is not possible to know in advance where the next one starts. Typically this is not a problem in the context of network security analysis since the way to process flows is to sequentially traverse each one to get to the ones of interest [29], or to build more complex network behavior profiles [27]. However, if CompactFlow is used in a network administration context, the types of workflows could require running the same queries to extract flows with a fixed set of parameters (e.g. given IP addresses). One could speed up such queries by indexing data of interest and then accessing it. This is described as *random access* and to enable this, CompactFlow precomputes an array of 4-byte pointers (offsets from the beginning of a file) to each flow record. Additionally, one can opt to use one of the indexing methods reported in Section 2.2. It is worth noting that this step is optional and such an array is not contained in the format by default. Overall, the header without an array of pointers takes 36 bytes of space.

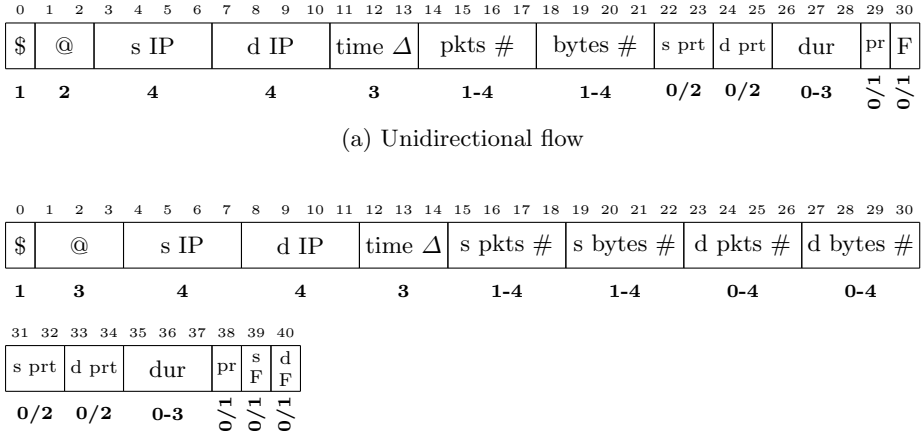


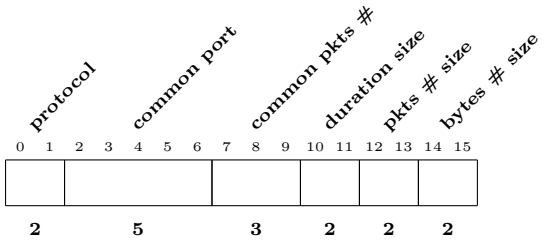
Fig. 2. Binary schema of CompactFlow records (in bytes).

3.2 Flow Binary Representation

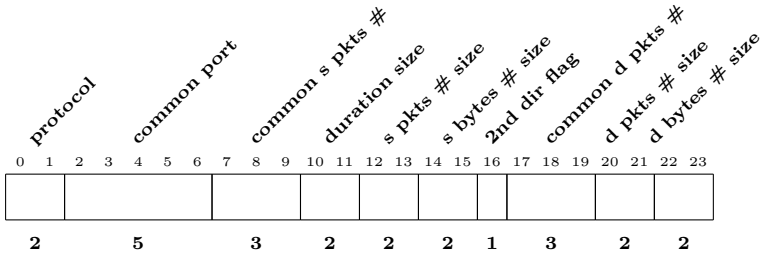
Flow records feature dynamic field sizes that are adjusted to the size of data that needs to be accommodated. The use of dynamic fields makes the flow more compact in cases where field values to be stored are small. As depicted in Figure 2a, every flow record contains the following fields: flow size (in bytes), control, source IP address, destination IP address, start time of the flow in terms of added milliseconds to the timestamp in the header, total number of packets, and total number of bytes. Optionally a flow can include a protocol, source and destination ports in the case of TCP or UDP protocols, duration and TCP flags. The length and position of variable-size fields is given by interpreting bits in the control field, described in Figure 3.

We noticed that some values are repeatedly used in flows. Saving the full values of such fields each time would require additional bytes per flow, which quickly build up if the number of flows is in the order of billions per day. For example, according to our observations the majority of traffic uses ICMP, TCP or UDP protocols. In order to save space, we use 2 bits (bits 0 to 1) of the control field, that allow for the storage of 4 values, to encode them with the fourth value meaning that another protocol is used which implies the existence of the protocol field in the binary data.

Port values (non-ephemeral) are encoded in a similar fashion using 5 bits (bits 2 to 6). They are only consulted if the protocol is either TCP or UDP (in other cases the port fields do not exist in the binary format). Encoded port values are specific to a given production network, hence they should be determined beforehand. Using those 5 bits, we can encode 32 values. The value of 0 means



(a) Unidirectional control field



(b) Bidirectional control field

Fig. 3. CompactFlow control fields (in bits).

that neither source port nor destination port belong to the list of most frequently used ports. The value of 1 is not used. The remaining 15 and 15 values mean that the source or destination port respectively is in the list of common ports. Each common port list hit saves 2 bytes of space. Since we observed that a significant number of flows constitute requests without a response which translate to a small number of packets, we use the next 3 bits (bits 7 to 9) to store small packet numbers. The value of 0 has a special meaning - the number of packets if different from a list of common small packet numbers. The remaining values from 1 to 7 are used to represent packet numbers, which results in 1 byte of saved space per flow. The next 2 bits (bits 10 to 11) in the control field store the length of the duration field. The value of 0 denotes that the duration is 0 and the respective duration field does not exist in the binary representation. The duration field can support non-standard values of up to 4.5 hours (with millisecond precision), even though the default active timeout value in the NetFlow 9 export protocol is only 30 minutes. This makes the format more robust towards changes of default values in flow exporters. The sizes of packets and bytes fields take 2 bits each (bits 12 to 13 and 14 to 15) to denote values up to 4 bytes. Summarizing, the size of a unidirectional flow can vary from 16 to 30 bytes. Each unidirectional CompactFlow file can store up to 143,165,576 flow records (unsigned integer using 4 bytes divided by a maximum size of a flow - 30), if one chooses to compute the array of pointers to each flow record.

CompactFlow also supports bidirectional flows (Figure 2b). The bidirectional format differs by the addition of packet and byte counters as well as TCP flags for the other side of the communication. It is not always the case that those counters

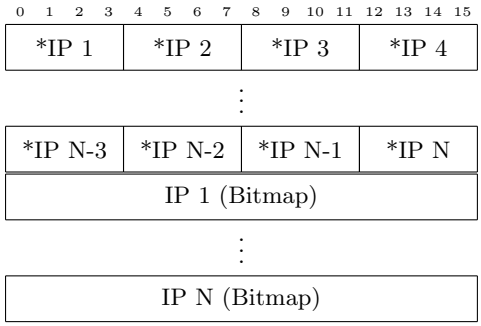


Fig. 4. Bitmap index of IP addresses.

exist, e.g. the communication might comprise only a request with no reply. In such scenarios, the destination counter values are not captured, hence the size of those counters can be equal to 0. For that reason, the size of a bidirectional flow is larger than its unidirectional counterpart and can take from 17 to 40 bytes.

To increase data processing performance, CompactFlow automatically places dynamic fields that could have the size of 0 at the end of the flow. If those fields were placed throughout the flow record, then one would need to query their size by calculating the corresponding flags in the control field in order to get to fields positioned after them. By using this design, we tried to minimize this behavior. Additionally, IPv6 flow records are supported and saved into separate files. The format for such files differs in the number of bytes allocated for each address - 4 bytes for IPv4 and 16 bytes for IPv6.

The second part of the CompactFlow framework considers flow processing techniques. Our proposal supports a variety of indexing methods. We present how to create the bitmap index, which is considered a state-of-the-art approach. Bitmap indexing is efficient as it uses only 1 bit per flow record to denote whether a given value of a flow field is present in a flow. Hence, each bitmap index is an array of bits with the size equal to the number of all flow records. Figure 4 reports the structure of such a bitmap index that is stored in a separate file as part of the CompactFlow format. Similarly, to the header of the unidirectional CompactFlow file, it stores the pointers to the locations of bitmap arrays for each field. In the case of this figure, these are IP addresses. In order to obtain selected flows, one needs to take the positions of bits with the value of 1 and jump to the respective flows by using the array of pointers in a binary CompactFlow file. Since the size of pointers is fixed-size (4 bytes), it is easy to jump to the correct ones with negligible overhead.

4 Evaluation

We compare our format to most popular open-source flow collectors that support binary file storage, i.e. Argus, flowd, nfdump, and SiLK. To carry out this comparison, we analyze the binary formats of such collectors to have full understanding of their flow representation. In Figure 5, we show the same uni-

directional flow represented using the aforementioned binary formats and our proposed CompactFlow format. The flow is shown in the plaintext format (Figure 5a) with color-coded field names (explained in Figure 5b). In our comparison we configure flow collectors to store only the fields that we consider, whenever possible. Most binary formats use fixed-length representation of each flow record, which makes the file format more straightforward to read from. Indeed, it is possible to jump by a constant number of bytes to get to the same field in the next flow. However, this feature also makes it extremely inefficient space-wise. CompactFlow is designed to achieve a trade-off between file size and processing speed. In fact, our proposed format applies a hybrid approach, in which the always-present fields are of constant length and the fields whose values can change are of variable length. This results in a compact representation that is the smallest of all presented binary formats, as shown in Figure 5h. It is worth noting that the protocol, destination port and number of packets are included in the control field as an optimization by our binary format (see Section 3.2). The sample flow in CompactFlow binary format is only 20 bytes. In what follows, we discuss and compare different flow collectors one by one.

Audit Record Generation and Utilization System (Argus) [4] is a popular, open-source flow monitoring framework. The Argus collector provides a binary file format to store flow records, which assigns a fixed-length space for fields constituting a flow. It is worth noting that such length remains fixed even when not all fields are captured. This approach leads to wasted space, which is a fundamental factor when dealing with large data sets. In total a flow record takes 116 bytes.

Another flow collector, flowd [1], offers binary storage at a reduced size of 48 bytes per flow record. It uses big-endian encoding and provides no file header. It offers an option to save protocol, TCP flags, and Type of Service without being able to selectively pick each one. Additionally, there is no option to save the timestamp from when the flow started, only the timestamp of receiving the flow by the collector. It also does not provide an option to store the duration of a flow. Both of these shortcomings limit its use in real-world settings.

A slight improvement in those regards is offered by nfdump [17]. It uses the nfcapd tool to collect flows from the exporter and to save them to binary files. It also allows for fine-grained specification of which fields to store. However, it keeps start and end timestamp from which duration field is calculated, which is not an efficient approach. It also uses little-endian encoding, which might seem strange since network order is big-endian. Moreover, the conversion between encodings might add unnecessary overhead to the collection process. The binary format of nfdump stores the header in 344 bytes, each flow record in 56 bytes and the footer in 44 bytes.

SiLK [29] is the most optimized open-source flow collector with a state-of-the-art binary format and a set of processing tools. It provides an option to specify endianness of files and provides optimizations such as storing flow duration instead of end time or storing average amount of bytes transferred per packet. This results in the smallest binary format of all open-source tools

02:59:40 8.96 TCP 192.168.1.43 58769 72.163.4.161 443 3 152

(a) Sample flow

timestamp duration proto srcIP srcPort destIP destPort packets bytes TCP flags

(b) Color legend

```
01:  3320 001d 0101 0102  c0a8 0101 0201 4105
02:  c0a8 012b 48a3 04a1  0600 e591 01bb 2020
03:  031a 1805 5c55 079c  0007 62a0 5c55 07a5
04:  0006 c660 1001 0602  0003 0098 3004 0003
05:  0000 0000 0000 0000  4800 0102 0000 0000
06:  4200 0005 0000 0000  0000 0000 0000 0000
07:  0000 0000 3200 0004  0000 0000 0000 0000
08:  0000 0000
```

(c) Argus

```
01:  600a 0000 0000 38a6  5c55 07ab 0002 d4a1
02:  1806 0000 c0a8 012b  48a3 04a1 e591 01bb
03:  0000 0000 0000 0003  0000 0000 0000 0098
```

(d) flowd

```
01:  0a00 3800 0600 0000  e401 bc01 9c07 555c
02:  a507 555c 0018 0600  91e5 bb01 0100 0000
03:  2b01 a8c0 a104 a348  0300 0000 0000 0000
04:  9800 0000 0000 0000
```

(e) nfdump

```
01:  da89 1003 2a80 2300  1840 0003 0000 0000
02:  e591 01bb c0a8 012b  48a3 04a1
```

(f) SiLK

```
01:  0000 1600 2c00 2b00  2400 2000 1400 1000
02:  0c00 0a00 0800 0400  1600 0000 0023 0000
03:  bb01 91e5 9800 0000  0300 0000 44bb 25ac
04:  6801 0000 0000 0000  a104 a348 2b01 a8c0
```

(g) FlatBuffers

```
01:  1862 e1c0 a801 2b48  a304 a136 a244 9891
02:  e523 0018
```

(h) CompactFlow

Fig. 5. Comparison of binary file formats.

Compression Method	CPU Cores	File Size (MB)	Size Gain (%)	RAM Load Time (s)	Processing Time (s)	Total Time (s)	Time Loss (%)
uncompressed		1,425	-	3.267	0.548	3.815	-
gzip	1	791	44	1.887	12.065	13.952	366
gzip	32	792	44	1.881	7.408	9.289	243
bzip2	1	697	51	1.609	61.638	63.247	1,658
bzip2	32	696	51	1.603	3.155	4.758	125
lzo	1	1,023	28	2.325	4.818	7.143	187

Table 2. Comparison of compression methods used with a CompactFlow file of 67,257,407 unidirectional IPv4 flow records.

with 24-88 bytes for the header and 28 bytes per flow record. However, it does not support bidirectional flows (even though it provides tools to match flows after they are captured) and it does not provide functionality to index flows. Moreover, it relies on the file hierarchy, which divides traffic into internal to internal, external to external, incoming and outgoing Web and ICMP traffic. While the hierarchy can speed up select queries regarding one of those types, it also makes the binary format more complex and adds overhead for more analytic queries, such as the selection of all flows within a time period to train machine learning models.

Additionally to flow collectors, we examine a popular, fast data serialization library, FlatBuffers [2]. This library supports a large variety of extensions in different programming languages to interact with its binary data format. One of its advantages is the flexibility of what information to store. This is done by writing a schema with a structure of data to be stored. In the case of storing flow records, FlatBuffers does not perform better than most of flow collectors. It uses 64 bytes per flow record. This is due to the support of only regular types (e.g. uint8, uint16, uint32, uint64) and no custom types (such as uint24) which leads to wasted space. Additionally, in order to store a collection of records, one needs to specify another type that will serve as a container for those records.

Even though the comparison favors the CompactFlow format, a single flow is not representative of a larger variety of flows on networks. In fact, the unidirectional record takes from 16 to 30 bytes, which in some cases can exceed SiLK's 28 bytes per flow record. For this reason, we evaluate the average size of a CompactFlow record by considering a variety of flows from a production network. Experimental results based on the analysis of 1,802,377,030 flows show that the average size of a flow in our proposed unidirectional binary format is 21.4 bytes - a value 24% smaller than SiLK's format.

To get a clearer view of what different binary formats mean in the real world, we used each of them to store over five months of flow data from the University of Oxford. The results are shown in Figure 6. In total 181,315,995,252 flows are stored that come from three networks with over 64 thousand hosts. The most inefficient format takes over 21 TB to store the entirety of this data. While the

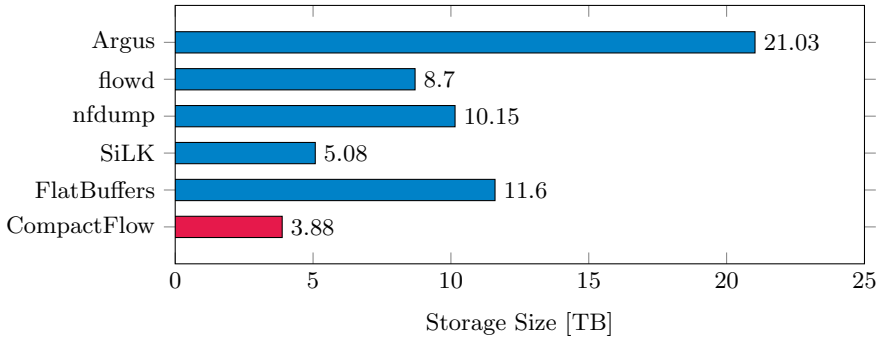


Fig. 6. 181 million flows from the University of Oxford using different flow collectors.

state-of-the-art binary format of SiLK uses about 5 TB, our proposed format uses only 3.88 TB.

The final aspect of the evaluation is a comparison of an uncompressed CompactFlow file against three common compression algorithms (i.e. gzip, bzip2, and lzo). Such an analysis is meant to show which approach is the fastest in terms of loading the file into memory (RAM load time) and reading contained flow records (processing time). We observed that those two metrics are a trade-off between disk speed and a chosen compression algorithm. As it was shown in 2014 in the evaluation of SiLK [29], reading flow records was faster from a compressed binary file. This was due to the limited speed of then widely used hard disk drives (HDDs). However, we assess that this is no longer true with the rising popularity and decreasing prices of solid-state drives (SSDs). In Table 2, we show that the increase of disk speeds (from 100 MB/s in [29] to 436MB/s in our analysis) results in faster processing of raw, uncompressed binary files. In order to reverse this trend, one can assign more CPU cores to speed up the decompression. We used 32 cores of a dual Intel Xeon CPU to determine that it takes 25% longer in total while reducing the file size by half. However, the prices and power requirements of such CPUs are high, which means that often they are not available to network administrators. As a result, compression is not suitable in the case of commodity hardware, which puts more emphasis on a small binary representation of flow data.

5 Discussion

Storing data in a binary format is more efficient than database-based methods in terms of size. In fact, the database-based methods need more space for indexing purposes, which may even take double the space required for the data [20]. They also do not allow a fine-grained control of what and when is indexed which accounts for their poor per-flow storage times. Our format provides a quicker and hybrid solution. It is also robust in case of errors. We use Golay code in the header to preserve fundamental information regarding the flows in the file, such as an hour-based timestamp, the IP version supported, or the type of flow.

Secondly, the header can be optionally enriched with an array of flow pointers. In this way, it would be easy to isolate the faulty flows in case errors occur within the file body. Faulty flows can be easily detected by relying on the first two flow fields, namely *flow size* and *flow control*. As a first check, we have to verify the following condition on the *flow size* value: $flow\_size \geq \sum_{i \in C} size(i) + 2$, where C is a set of flow fields of constant size and $size(i)$ is the size of field i. The additional 2 is related to the two variable fields with a minimum size of 1 byte (i.e. number of packets and bytes). The condition does not comprise the other variable fields since their minimum size is 0. A second check on flow consistency could be made on the packet and byte flow fields. Indeed, it is known that not only is the former smaller than the latter, but that the following condition is verified: $bytes\# \geq min\_packet\_size * packets\#$, where min_packet_size is the minimum allowed packet size by the considered protocol.

Our evaluation shows that even though flow collectors use the most efficient type of data storage, binary files, they usually do so in an inefficient manner. In fact, a string representation of each flow record would take less space than in most evaluated binary formats.

We do not evaluate compressed sizes of different flow collectors' binary formats. Even if compressed sizes were similar, in order to process the data, one needs to decompress it - which brings us to the initial problem since the file sizes start to matter again. We show in Section 4 that compression slows down the processing of flows. Moreover, memory prices show no signs of decreasing, hence it is important for a format to have a minimal memory footprint.

6 Conclusion

In this paper, we presented a hybrid binary file format to store network flow data. It not only is compact in its representation, but also supports well-known indexing approaches to speed up flow queries. To assess the performance of the CompactFlow format, we compared it to the most popular open-source flow collectors with an in-depth analysis of their binary formats. Then, we carried out an extensive comparison in terms of storage size on a real-world traffic dataset from the University of Oxford. Finally, we evaluated the impact of compression on our format in terms of file size and processing time.

References

1. flowd. https://code.google.com/archive/p/flowd/
2. Flatbuffers. https://google.github.io/flatbuffers/ (2015)
3. MariaDB ColumnStore. https://mariadb.com/kb/en/library/mariadb-columnstore/ (2017)
4. Argus: https://qosient.com/argus/ (1985)
5. Chandrasekaran, S., et al.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In: Proc. of ACM SIGMOD (2003)
6. Chen, Z., et al.: TIFAflow: Enhancing traffic archiving system with flow granularity. Tsinghua Science and Technology (2013)

7. Chen, Z., et al.: A survey of bitmap index compression algorithms for big data. Tsinghua Science and Technology **20** (2015)
8. Claise, B., et al.: IPFIX protocol specifications. RFC7011 (2004)
9. Claise, B.: Cisco Systems NetFlow Services Export Version 9. Tech. rep., The Internet Society (2004)
10. Cranor, C., et al.: Gigascope: A Stream Database for Network Applications. In: Proc. of ACM SIGMOD (2003)
11. Deri, L., et al.: Collection and exploration of large data monitoring sets using bitmap databases. Proc. of TMA (2010)
12. Desnoyers, P.J., et al.: Hyperion: High Volume Stream Archival for Retrospective Querying. In: Proc. of USENIX ATC (2007)
13. Fusco, F., et al.: High speed network traffic analysis with commodity multi-core systems. In: Proc. of IMC (2010)
14. Fusco, F., et al.: NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic. In: Proc. of the VLDB Endowment (2010)
15. Fusco, F., et al.: pcapIndex: An Index for Network Packet Traces with Legacy Compatibility. ACM Comput. Commun. Rev. **42** (2012)
16. Fusco, F., et al.: RasterZip: Compressing Network Monitoring Data with Support for Partial Decompression. In: Proc. of IMC (2012)
17. Haag, P.: Watch your Flows with NfSen and NFDUMP (2005)
18. Held, G., Marshall, T.: Data compression; techniques and applications: Hardware and software considerations. Wiley (1991)
19. Hofstede, R., et al.: The network data handling war: MySQL vs. NfDump. EUNICE (2010)
20. Hofstede, R., et al.: Flow monitoring explained: From packet capture to data analysis with NetFlow and IPFIX. IEEE COMST **16** (2014)
21. Lampertand, R.T., et al.: Vermont a versatile monitoring toolkit for ipfix and psamp. In: IEEE/IST Workshop MonAM (2006)
22. Lee, J., et al.: FloSIS: a highly scalable network flow capture system for fast retrieval and storage efficiency. In: Proc. of USENIX ATC (2015)
23. Li, X., et al.: Advanced Indexing Techniques for Wide-Area Network Monitoring. In: Proc. of ICDE (2008)
24. Lucente, P.: pmacct: steps forward interface counters. http://www.pmacct.net/pmacct-stepsforward.pdf (2008)
25. Maier, G., et al.: Enriching Network Security Analysis with Time Travel. ACM Comput. Commun. Rev. (2008)
26. Oberhumer, M.: Lempel-Ziv-Oberhumer data compression (2013)
27. Piskozub, M., et al.: MalAlert: Detecting Malware in Large-Scale Network Traffic Using Statistical Features. SIGMETRICS Perform. Eval. Rev. **46**(3), 151–154 (2019)
28. Reiss, F., et al.: Enabling Real-Time Querying of Live and Historical Stream Data. In: Proc. of SSBD (2007)
29. Thomas, M., et al.: SiLK: A tool suite for unsampled network flow analysis at scale. Proc. of IEEE BigData (2014)
30. Velan, P., et al.: Flow Information Storage Assessment Using IPFIXcol. In: Dependable Networks and Services. Springer (2012)
31. Wu, K., et al.: FastBit: interactively searching massive data. Journal of Physics (2009)
32. Xie, G., et al.: IndexTrie: Efficient archival and retrieval of network traffic. Computer Networks **124** (2017)