electronics

MDPI

Article

An Efficient Indexing Scheme for Network Traffic Collection and Retrieval System

Chao Jiang [1,2], Jinlin Wang [1,2] and Yang Li [1,*]

[1] National Network New Media Engineering Research Center, Institute of Acoustics, Chinese Academy of Sciences, Beijing 100190, China; jiangc0110@163.com (C.J.); wangjl@dsp.ac.cn (J.W.)
[2] School of Electronic, Electrical and Communication Engineering, University of Chinese Academy of Sciences, Beijing 100049, China
* Correspondence: liyang@dsp.ac.cn

Abstract: Historical network traffic retrieval, both at the packet and flow level, has been applied in many fields of network security, such as network traffic analysis and network forensics. To retrieve specific packets from a vast number of packet traces, it is an effective solution to build indexes for the query attributes. However, it brings challenges of storage consumption and construction time overhead for packet indexing. To address these challenges, we propose an efficient indexing scheme called IndexWM based on the wavelet matrix data structure for packet indexing. Moreover, we design a packet storage format based on the PcapNG format for our network traffic collection and retrieval system, which can speed up the extraction of index data from packet traces. Offline experiments on randomly generated network traffic and actual network traffic are performed to evaluate the performance of the proposed indexing scheme. We choose an open-source and widely used bitmap indexing scheme, FastBit, for comparison. Apart from the native bitmap compression method Word-Aligned Hybrid (WAH), we implement an efficient bitmap compression method Scope-Extended COMPAX (SECOMPAX) in FastBit for performance evaluation. The comparison results show that our scheme outperforms the selected bitmap indexing schemes in terms of time consumption, storage consumption and retrieval efficiency.

Keywords: network traffic retrieval; packet storage format; packet indexing; wavelet matrix

Citation: Jiang, C.; Wang, J.; Li, Y. An Efficient Indexing Scheme for Network Traffic Collection and Retrieval System. *Electronics* **2021**, *10*, 191. https://doi.org/10.3390/electronics10020191

Received: 17 December 2020
Accepted: 12 January 2021
Published: 15 January 2021

1. Introduction

With the rapid development of the Internet in recent years, network traffic has increased dramatically, which also brings more challenges to network security due to the openness of the Internet. The monitoring and analysis of network traffic have become one of the critical approaches to ensure network security. Most network monitoring applications, such as intrusion detection, firewall and so on, obtain threat information through real-time analysis of the network traffic and perform corresponding actions without storing the network traffic. However, in some cases, it is very convenient to record all the content of network traffic, including packet or flow entries, network sessions, application-layer logs, etc. In this way, the network managers are able to conduct retrospective analyses of the network behaviors and application data that have occurred. The archiving and retrieval of network traffic is becoming increasingly essential in anomaly detection, network forensics and many other research fields.

The design of packet storage format has practical significance for network traffic retrieval. A comparative analysis of the packet storage formats used for network forensics can be found in Reference [1]. Libpcap [2] is a standard packet file format. The Pcap file format is straightforward but not suitable for an extension. Pcap Next-Generation (PcapNG) [3], which has high scalability and the ability to append data, is a new file format evolved from the Pcap format for packet dumping. In other words, we can append some non-standard custom blocks to the original file without affecting the compatibility and

integrity of the dump file. Applications such as Wireshark and Tcpdump will still process packets in the dump file correctly.

Traditional network traffic analysis tools, such as Tcpdump and Wireshark [4], use a linear search algorithm to find the matched packets. It is necessary to scan the entire file for matching when performing a query operation. Thus, the complexity of the query increases linearly with the size of the original file. To improve the retrieval efficiency, indexing for the query attributes has become a widely used scheme.

Several methods for the archiving and retrieving of network traffic have been proposed, such as TelegraphCQ [5], Hyperion [6], Time Machine [7], TIFAflow [8] and IndexTrie [9]. These methods record packet or flow entries and build indexes for various query fields. Meanwhile, the offsets of the packet or flow entries in the trace files are recorded. When executing a query operation, the index files are retrieved according to the query attribute to get the number of the matched entries. Then the address offsets can be obtained through the mapping relationship between the entry number and the address offset. It is easy to extract packet or flow entries from the original trace file with these offsets. However, building indexes based on multiple fields in the packet header will not only consume a large amount of storage space but also slow down the speed of retrieval. Moreover, disk I/O is easy to become a bottleneck. In order to reduce the storage consumption of the index data and improve retrieval efficiency as much as possible, several data structures have been applied to index and retrieve the network traffic. For example, B+ Tree was used in References [10,11], and bitmap was used in References [12–15].

The main challenges of network traffic collection and retrieval systems include fast indexing, efficient archiving and fast retrieval. To achieve fast indexing and efficient archiving, there is a need to choose an efficient data structure that can quickly build indexes with low storage consumption. Fast retrieval requires that the system can retrieve specific packets from a large number of packet traces at a high speed. In this paper, we implement a network traffic collection and retrieval system. First, we present the architecture of our system and design a packet storage format based on the PcapNG. Then, an indexing scheme based on the wavelet matrix data structure is proposed to solve the challenges faced by the network traffic collection and retrieval system. The rest of the paper is organized as follows: Section 2 introduces the related work. Section 3 gives a brief overview of the wavelet matrix data structure used for packet indexing. Section 4 presents our implementation, which includes the architecture of the proposed network traffic collection and retrieval system, the storage format of packet trace files and the indexing scheme. Section 5 shows the performance evaluation results. Section 6 concludes this paper.

2. Related Works

With the increasing demand for collecting large amounts of network traffic, the retrieval of historical network traffic has become a major challenge. Many indexing schemes have been proposed for the retrieval of network traffic. The main data structures used in these schemes include hash, B+ tree and bitmap.

- Hash: Hash index uses a certain hash algorithm to convert the key value into a hash value. According to the hash value, the location for an entry to be inserted or queried can be found immediately. Therefore, it has a very fast speed. Linked lists are the most commonly used method to resolve hash conflicts. Inevitably, as the number of entries increases, hash collisions will become more frequent. This will result in a significant reduction in retrieval efficiency. Moreover, a hash index cannot support range queries and union queries. In Reference [7], S. Kornexl and V. Paxson et al. proposed a typical hash-based indexing scheme Time Machine for the archiving and retrieval of the network traffic stream. It could greatly reduce the recorded volume based on the heavy-tailed characteristics of network traffic. Due to the characteristics of hash, Time Machine was faced with the problem that the retrieval efficiency will be significantly reduced with the increase of network traffic.

- B+ Tree: B+ tree has been widely used in relational database management systems, such as MySQL and SQL Server. Compared with the hash index, the B+ tree index has good support for range queries. However, the B+ tree is not suitable for indexing columns with many duplicate values. For network traffic retrieval, some commonly used indexed attributes, such as the destination port, typically contain a large number of duplicate values. In References [10,11], the B+ tree was used to build indexes for network traffic. According to the evaluation results in Reference [10], the performance of indexing for network traffic using B+ trees is up to 40 K records per second. Hence, the indexing efficiency of the B+ tree cannot meet the requirements of the current 10G or even higher network links.
- Bitmap: Indexing for packets with bitmaps is a popular way to solve the problem of network traffic retrieval. Many indexing schemes based on bitmaps, such as pcapIndex [12], FastBit [13] and NET-FLi [14], have been studied. Bitmap index maps a key value to a bit vector which contains 0 or 1. Table 1 illustrates a bitmap example. Bitmap index supports efficient queries and can easily handle complex queries. However, the disadvantage of the bitmap index is that it takes up much storage space when handling a large number of records. To address this challenge, many bitmap compression algorithms have been proposed. In Reference [16], Wu et al. proposed the WAH algorithm. For further compression on the basis of WAH, many variants of WAH have been proposed, including Position List WAH (PLWAH) [17], Compressed 'n' Composable Integer Set (CONCISE) [18], Compressed Adaptive Index (COMPAX) [14], SECOMPAX [19], Byte Aligned Hybrid (BAH) [20] and Compressing Dirty Snippet (CODIS) [21]. Some of them are already applied in network traffic archival and retrieval. In Reference [22], Chen et al. presented a survey of different traffic archival and retrieval systems and various bitmap index compression algorithms.

Table 1. Bitmap index example.

RowID	Value	Bitmap Index			
		=1	=2	=3	=4
1	4	0	0	0	1
2	2	0	1	0	0
3	3	0	0	1	0
4	1	1	0	0	0
5	3	0	0	1	0

FastBit is an open-source data processing library, which offers a set of searching functions supported by compressed bitmap indexes. FastBit uses the basic WAH for bitmap compression, and it is the most widely used bitmap index technique. For example, it has been used in several network traffic archiving and retrieval systems proposed in References [8,23,24]. In Reference [25], Fusco et al. presented the real-time bitmap index WAH for network traffic. In Reference [14], COMPAX was applied for the indexing of streaming network traffic. Although bitmaps have been compressed by various methods, the usage of storage space and retrieval efficiency still cannot meet the requirements of high-speed network links.

3. Overview of Wavelet Matrix

In this paper, the proposed indexing scheme is implemented based on the wavelet matrix data structure introduced by Claude et al. in Reference [26]. This section provides a concise description of the wavelet matrix. The wavelet matrix is an alternative representation for large alphabets that retains all the properties of wavelet trees but is significantly faster [26]. Therefore, we first give a brief overview of the wavelet tree [27,28] before illustrating the usage of the wavelet matrix.

3.1. Wavelet Tree

The wavelet tree is a compact and efficient data structure that converts a string into a balanced binary tree composed of bit vectors. It was introduced by Grossi, Gupta and Vitter in Reference [27]. According to the Reference [28], the wavelet tree data structure has been adopted in various applications, such as basic and weighted point grids, sets of rectangles, strings, inverted indexes, document retrieval indexes, full-text indexes and so on.

Let $S = \{a_1, a_2, \cdots, a_n\}$ represents a sequence of symbols $a_i \in \Sigma$, where Σ is an alphabet with the size σ. Three basic operations, namely *rank*, *select* and *access*, are defined for wavelet tree. For sequence S, $rank_a(S, i)$ the operation returns the number of times that the symbol a appears in the sequence S from the start position to position i, $select_a(S, j)$ the operation returns the position of the j-th occurrence of symbol a in S, and $access(S, i)$ operation retrieves the symbol at position i in S.

Figure 1 shows an example of the balanced wavelet tree constructed from the sequence $S = \{1521863875843287\}$. At the root node, the alphabet $\Sigma = \{1, 2, 3, 4, 5, 6, 7, 8\}$ of S can be divided into two parts, $\Sigma_l = \{1, 2, 3, 4\}$ and $\Sigma_r = \{5, 6, 7, 8\}$. The symbols belonging to the alphabet Σ_l in S are marked with "0", and the symbols belonging to the alphabet Σ_r are marked with "1". Thus, the sequence S can be encoded as the bit sequence $\{0100110111100011\}$ at the root node. In the bitmap of each node, a 0 bit means the corresponding symbol belongs to the left subtree of this node, and a 1 bit means that it belongs to the right subtree. This procedure is executed recursively at each node until reaching the leaf node.

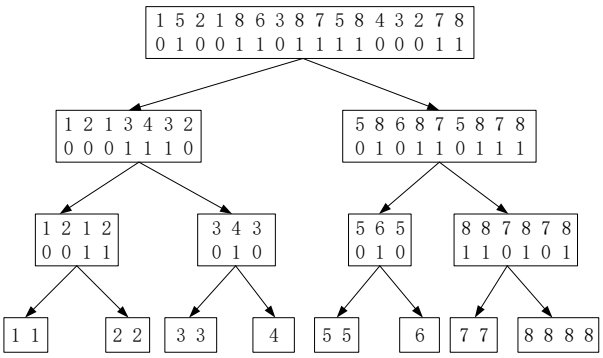
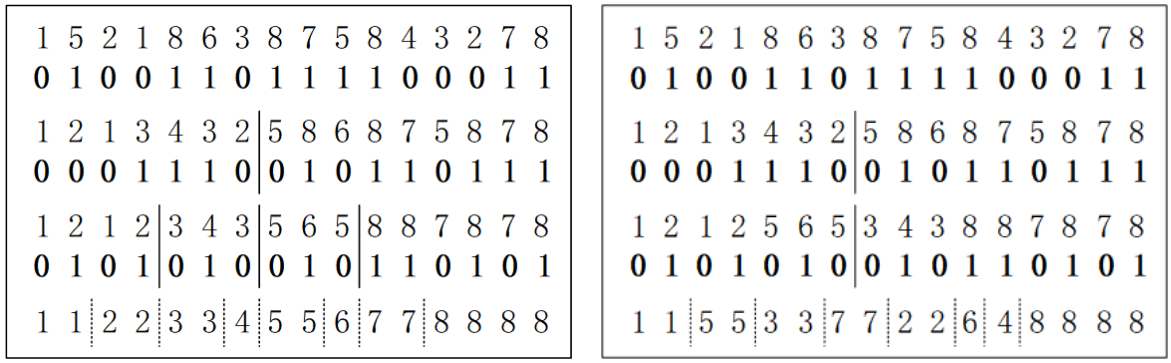


Figure 1. The wavelet tree for sequence $S = \{1521863875843278\}$.

A balanced wavelet tree constructed from sequence S has σ leaves and has a height $lg\ \sigma$. At each level, n bits are required to represent the bit sequence. In this basic form, the space required by wavelet tree is $n\ lg\ \sigma + o(n\ lg\ \sigma) + O(\sigma\ lg\ n)$ bits. Using the bitmap implementation in Reference [29], these three operations described above are performed in time $O(lg\ \sigma)$. It can be seen that the execution time has nothing to do with the length of sequence S. It is only related to the alphabet size σ. However, the space consumption will increase significantly for large alphabets because extra $O(\sigma\ lg\ n)$ bits are required to store the topology of the tree. It is significant to remove the $O(\sigma\ lg\ n)$ bits used for pointers for large alphabets. The wavelet matrix proposed is a good solution for this purpose.

3.2. Wavelet Matrix

Compared with the wavelet tree, the wavelet matrix employs a simpler mapping mechanism from one level to the next. Specifically, all the symbols that are marked with "0" at each level go left, and the rest goes right. For each level, a single value z_l is stored to mark the number of zeros in level l. Figure 2a,b show the wavelet tree without pointers and the wavelet matrix for the sequence in Figure 1, respectively.



(**a**) (**b**)

Figure 2. (**a**) The wavelet tree without pointers for sequence $S = \{1521863875843278\}$; (**b**) the wavelet matrix for the same sequence $S = \{1521863875843278\}$.

In order to further illustrate the usage of the wavelet matrix, we give a description of the three basic operations for sequence S in detail. Let B_l represents the bitmap at level l and $i = i_0$. To perform the $access(S, i)$ the operation, the first step is to check the value of $B_0(i_0)$. If $B_0(i) = 0$, then the corresponding position i_1 at the next level is set to $rank_0(B_0, i_0)$. Otherwise, i_1 is set to $z_0 + rank_1(B_0, i_0)$. Next, a similar operation can be performed at the next level. This procedure is repeated until reaching the leaf node. The value at the leaf node is the result of $access(S, i)$.

For the computation of $rank_a(S, i)$, the scope of position from 0 to position i should be tracked step by step. Initially, $p_0 = 0$ and $i = i_0$. At each level l, p_l is mapped to p_{l+1} and i_l is mapped to i_{l+1}. If the symbol a is marked as "0", then p_{l+1} is set to $rank_0(B_l, p_l)$ and i_{l+1} is set to $rank_0(B_l, i_l)$. Otherwise, $p_{l+1} = z_l + rank_1(B_l, p_l)$ and $i_{l+1} = z_l + rank_1(B_l, i_l)$. The procedure is repeated until arriving at the leaf level. Thus, the final result of $rank_a(S, i)$ is $i_l - p_l$. Compared with the standard wavelet tree, the *rank* operation needs an extra binary rank operation for the wavelet matrix. This is because the position p_l needs to be tracked. It can be accelerated by storing an array C, in which $C[a]$ points to the start position of the symbol a in the array of leaves. Thus, the final return value of $rank_a(S, i)$ is $i_l - C[a]$.

The *select* is the inverse operation of *rank*. For the operation $select_a(S, j)$, starting from the leaf node corresponding to the symbol a, an upward tracking of the position of a needs to be done. If the position j_l of symbol a at level l is mapped from a "0" in level B_{l-1}, the corresponding position j_{l-1} is $select_0(B_l, j_l)$. Otherwise, the position j_{l-1} is $select_1(B_l, j_l - z_l)$. The position j_0 at the root, a bitmap is the final result.

Algorithm 1 gives the pseudocode for the three basic operations on the wavelet matrix. $label(v)$ denotes the symbol at the leaf node v and $mark_l(a)$ denotes the corresponding bit of symbol a at level l.

Just like the wavelet tree, the wavelet matrix is also an excellent index data structure. It is very suitable for the compressed representation of data, which is achieved by using specific coding on the bitmap or changing the shape of the tree. At the same time, it supports three fast basic operations: *rank*, *select* and *access*. These properties of the wavelet matrix make it very useful for packet indexing in the case of high-volume network traffic. Our work is dedicated to extracting packet fields from network packet traces and then indexing these fields for the fast retrieval of network traffic. We will give a detailed description of our index scheme based on the wavelet matrix in Section 4.

Algorithm 1 Wavelet matrix algorithms: on the wavelet matrix of sequence S, $access(0, i)$ returns $S[i]$; $rank(0, a, i)$ returns $rank_a(S, i)$; and $select(a, j)$ returns $select_a(S, j)$.

```
access(l, i)
if reach a leaf node v then
  return label(v)
end if
if B_l(i) = 0 then
  i ← rank_0(B_l, i)
else
  i ← z_l + rank_1(B_l, i)
end if
return access(l + 1, i)
```

```
rank(l, a, i)
if reach a leaf node v then
  return i − C[a]
end if
if mark_l(a) = 0 then
  i ← rank_0(B_l, i)
else
  i ← z_l + rank_1(B_l, i)
end if
return rank(l + 1, a, i)
```

```
select(a, j)
l ← lg(σ)
j ← j + C[a]
while l ≥ 0 do
  if mark_l(a) = 0 then
    j ← select_0(B_l, j)
  else
    j ← select_1(B_l, j − z_l)
  end if
  l ← l − 1
end while
return j
```

4. Implementation

In this section, we first introduce the architecture of our network traffic collection and retrieval system and the function of each module in the system. Then we show the design of the packet storage format based on the PcapNG format. Finally, we give the implementation details of the proposed indexing scheme called IndexWM.

4.1. System Architecture

In this part, we present the architecture of the proposed network traffic collection and retrieval system in detail. As illustrated in Figure 3, the system consists of four function modules: Collection Module, Storage Module, Indexing Module and Retrieval Module.

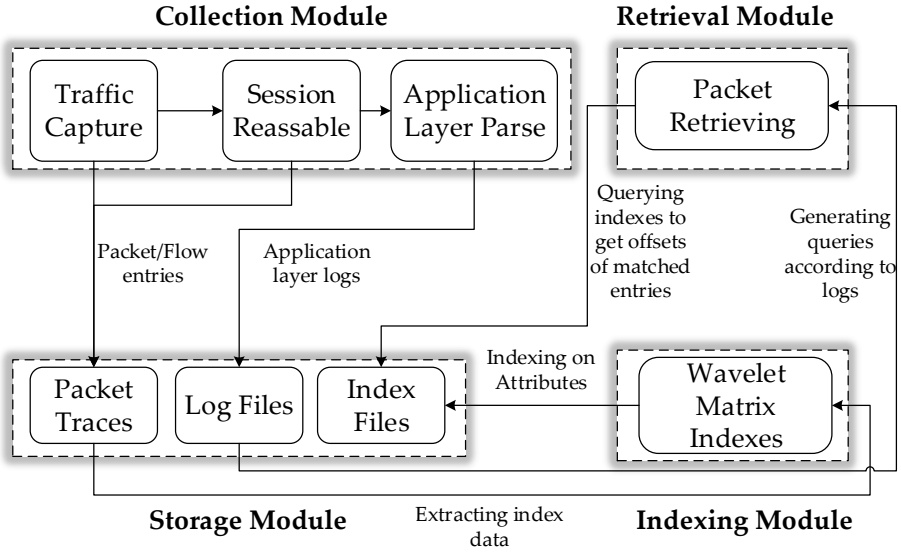


Figure 3. The architecture of the network traffic collection and retrieval system.

The collection module obtains real-time network traffic from the network interface or obtains network packets from the packet trace files stored on the disk. In addition, this module performs session reassemble to eliminate the influences of packet disorder, retransmission and fragmentation on subsequent analysis, analyzes application layer protocols and generates application logs. The storage module is used for recording raw packets, flow information and application layer logs. The raw packets are stored in a custom format based on PcapNG. Application layer logs contain application data, which can be used for the analysis of historical network behaviors. The indexing module extracts index data from the packet trace files and builds indexes for packets. The retrieval module

receives queries from users and retrieves the indexes associated with the query attributes to obtain the numbers and offsets of the matched packets. With these offsets, the system can easily extract the matched packets from the packet trace files. Usually, queries are generated based on the application logs.

4.2. Packet Storage Format

Here we first give a concise introduction of the Pcap Next-Generation (PcapNG) File Format [3]. A PcapNG format file must start with a Section Header Block (SHB). Generally, the whole file contains only one section header block, but there can be more than one Header Block in a dump file. A typical PcapNG file structure is composed of a Section Header Block (SHB), a single Interface Description Block (IDB) and several Enhanced Packet Blocks (EPBs). The PcapNG file format possesses good extensibility and portability. Meanwhile, it supports appending data to the end of the given file without affecting the readability of the file.

As described in Section 4.1, the collection module carries out a deep analysis of the network traffic to get some metadata information of packets, such as timestamp, payload length and application layer protocol. By attaching some specific metadata information to the packet trace file, the difficulty of extracting index data will be significantly reduced for the indexing module. Therefore, we introduce a custom packet storage format based on the PcapNG format. In our design, we add options that include some metadata information to the SHB and EPBs. At the same time, a Timestamp Index Block (TIB) is defined and appended to the end of the trace file. The packet storage format we designed is shown in Figure 4. Next, we will give a detailed description of this format.

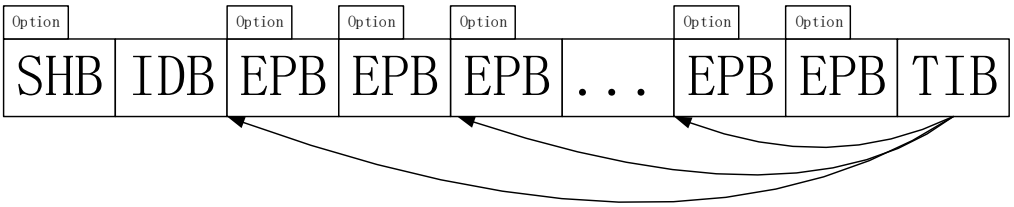


Figure 4. The packet storage format.

The option added to an SHB is some file description information, including, but not limited to, magic number, file generation time and packet capture device information. For each EPB, we embed an option that contains the description information of the packet or flow entry. The description information includes, but is not limited to VLAN Tag, layer 3 protocol (L3_Proto), layer 3 offset (L3_Offset), layer 4 protocol (L4_Proto), layer 4 offset (L4_Offset), Payload Length (PayloadLen), Application Layer Protocol (App_Proto), Session ID, etc. The specific format of the EPB option can be flexibly defined according to the practical system requirements. Thus, there is no need to parse packets again when we build indexes for packets. The retrieval module is facing the problem of how to retrieve packets quickly in a huge number of packets and how to limit the scope of retrieval, thereby enhancing the efficiency of packet retrieval. Generally, the packet retrieval conditions contain the start and end timestamp information, but it is not a good idea to build indexes for the timestamp of every packet. To solve this problem, we design a Timestamp Index Block (TIB) that contains the packet number and the corresponding timestamp information. The TIB is appended to the end of the file. Figure 5 shows the format of TIB.

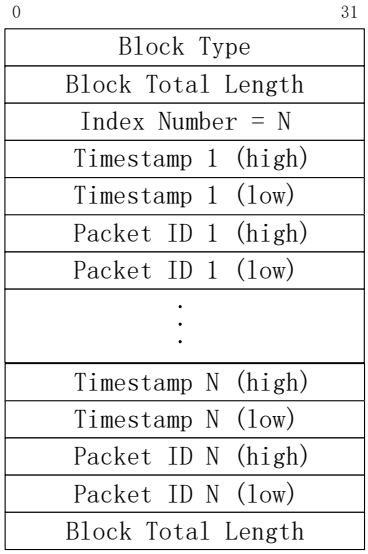


Figure 5. The Timestamp Index Block (TIB) format.

Now we explain the steps of recording packets with the proposed packet storage format in detail. We first create an SHB with a file description option and a single IDB. Then, for each received packet, we create an EPB to record the content of the packet. Meanwhile, an option that contains packet description information is added to the EPB. The above process of writing EPB will be repeated until no packets arrive or the file size reaches the set threshold. Starting from the first packet, we record the current packet number and timestamp every N packets and add them to the TIB. The value of N can be adjusted according to the file size threshold. Anyway, the number and timestamp of the last packet must be written to the TIB. Finally, we append the TIB to the end of the file.

4.3. Indexing Scheme

4.3.1. Indexing

Based on the wavelet matrix data structure, we propose an indexing scheme called IndexWM. Let us first introduce the attributes used for indexing. Although any field in a packet can be used as an index attribute, there are some common index attributes including timestamp, source IP address (SRCIP), destination IP address (DSTIP), source port (SPORT), destination port (DPORT), layer 4 protocol (L4_Proto), payload length (PayloadLen), application layer protocol (App_Proto) and so on. In this paper, we select the classical 5-tuples: SRCIP, DSTIP, SPORT, DPORT and L4_Proto as index attributes to simplify the implementation and evaluation. As shown in Figure 6, the first step of indexing is to extract the values of the index attributes from the original packet trace file. In our implementation, the packet trace file is achieved in the format described in Section 4.2. As a result, the process of extracting 5-tuples information is significantly simplified. For example, if we want to get the IP address of a packet, we only need to obtain the L3 offset and L3 protocol from the EPB option and then jump to the corresponding position to get the IP address value according to the L3_Proto format. It is similar to obtain the values of other fields in the packet. Moreover, we should maintain an array that stores the offsets of packets. The size of the array is equal to the number of packets. Using the offset in this array, we can quickly extract the matched packets from the packet trace file. After all the index data have been extracted from the trace file, we construct a wavelet matrix for each attribute in 5-tuples. In our implementation, we divide the SRCIP and DSTIP into four parts to reduce the size of the alphabet, thereby improving the retrieval efficiency of network traffic. At the end of the trace file, the timestamp and packet number information in TIB is utilized to index the timestamp.

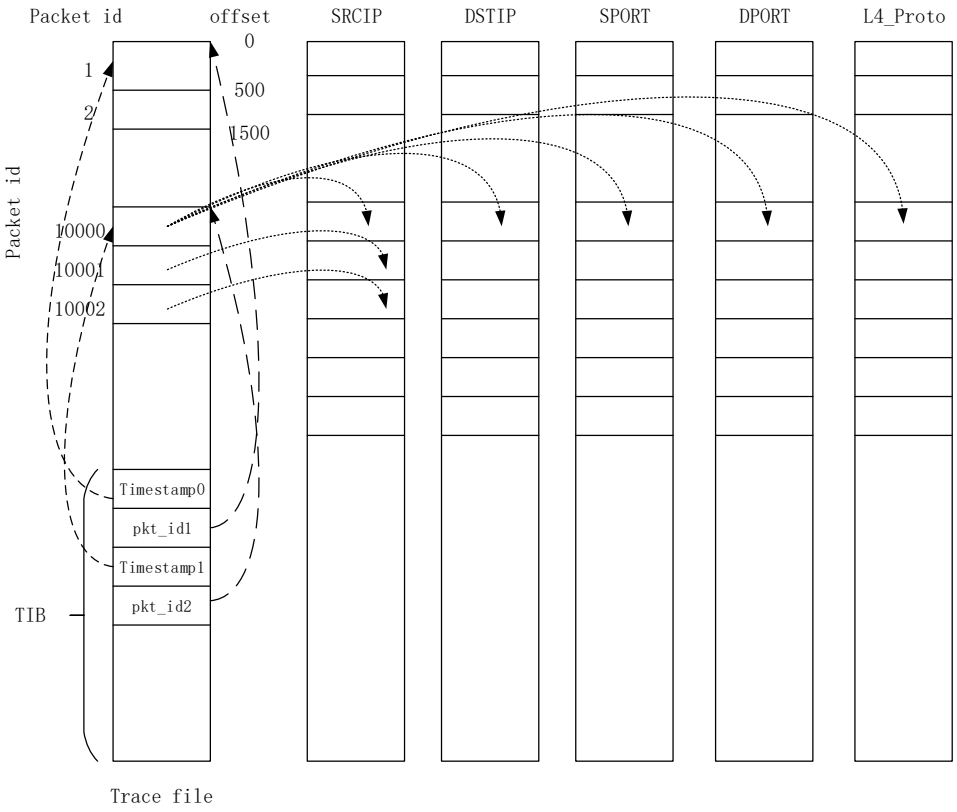


Figure 6. Extracting index data from a packet trace file.

In Reference [30], Raman, Raman and Rao proposed the RRR implementation for bitmap, which could answer rank queries in $O(1)$ time and provide implicit compression for bit sequences. With the RRR structure, a bitmap can be divided into blocks. It is assumed that each block is represented by b bits, and f blocks are grouped into a superblock. Each block can be replaced with a pair of values, namely a class value c and offset value o. The class value represents the number of 1 s in a block, and the offset value indicates the index of this block in the RRR table. Figure 7 shows an example of the RRR table. Each superblock boundary stores the sum of rank queries of previous blocks, which can avoid iterating the entire RRR sequence to answer the rank queries. For a rank query, we only need to jump to the corresponding superblock and then iterate over the blocks in this superblock. Despite the fact that the RRR implementation incurs some performance penalty in practice, they can significantly reduce the index size. In our indexing scheme, the RRR structure is employed to represent the bit sequences in the wavelet matrix. The value of b is set to 32, and f is set to 4. Rank operation on a bit sequence is the key step to perform the three operations described in Section 3.2. The rank operation $rank(i)$ is defined as the number of set bits (1 s) in the range $[0, i]$. To calculate $rank(i)$ under the RRR representation, we first need to calculate the block and superblock corresponding to position i. They are calculated as $i_b = \frac{i}{b}$ and $i_s = \frac{i_b}{f}$, respectively. Once the superblock i_s is obtained, the result is set to the sum of the rank values before i_s boundary, which is precalculated and stored at the boundary. Then we turn to the superblock i_s for iterative calculation. For each block in this superblock, we calculate the rank value through the class-offset pair (c, o) and add it to the result. This procedure is repeated until reaching i_b. The final result can be obtained by adding $rank_{i_b}(j, c)$ to the previous result, where $j = i \ mod \ b$.

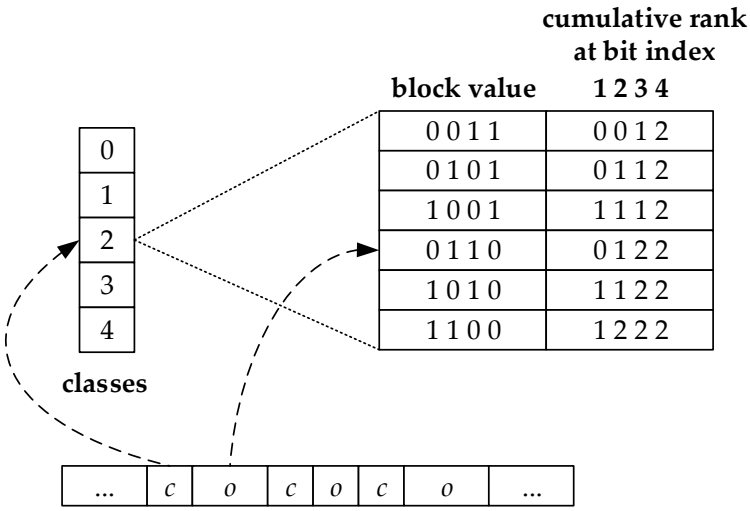


Figure 7. An example of an RRR table, with a lookup for class $c = 2$ and offset $o = 3$.

It is worth noting that the proposed indexing method supports indexing more packet fields, such as the flag fields in IP or TCP, payload length, and application layer protocol. The alphabet size of these fields is small, so indexing them does not take up much storage space. In a word, we can flexibly select fields to build indexes based on the wavelet matrix.

4.3.2. Index Query

In this part, we give the procedure of packet retrieval in detail through an example. To retrieve specific packets, we first need to query the index files on the disk. Here we consider two types of queries: single attribute query and multi-attribute query.

A single attribute query refers to the query on a single index attribute. The value of the query attribute can be a certain value (e.g., SRCIP $-$ "10.10.0.80") or a range represented by a wildcard (e.g., SRCIP $-$ "10.10.8.$*$"). A multi-attribute query can be regarded as the intersection or union of several single attribute queries. It means that the multi-attribute query can be decomposed into several independent subqueries, and the final query result can be obtained by merging the results of these subqueries. Figure 8 presents the procedure of packet lookup and extraction in terms of the query: $SRCIP =$ "10.10.$*$.$*$" $AND\ DPORT = 80$.

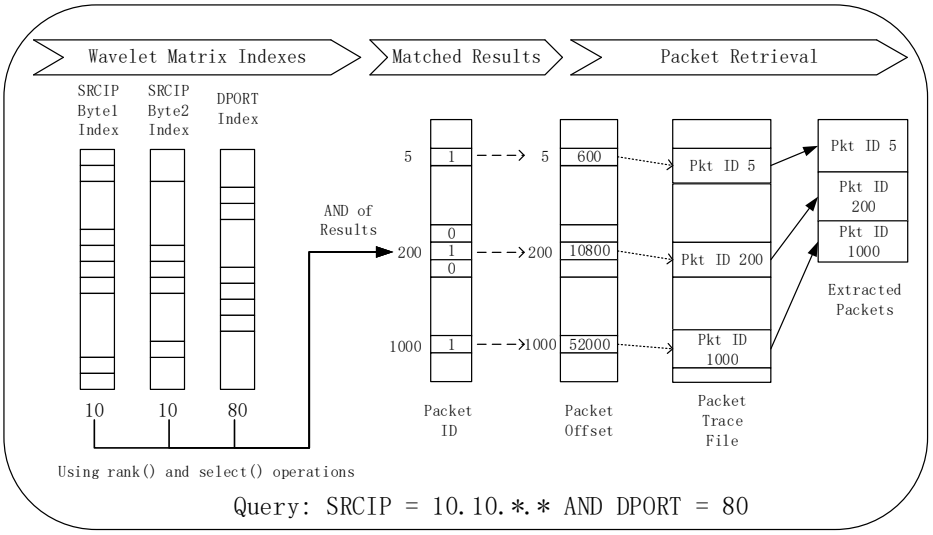


Figure 8. The procedure of packet lookup and extraction.

The query in Figure 8 is the intersection of a query on SRCIP and a query on DPORT, where the value of SRCIP is in the form of wildcards. Obviously, if a certain query attribute is represented by a wildcard, we do not need to query the corresponding index file. Therefore, for the example in Figure 8, we only need to pay attention to three index attributes: the first byte of SRCIP, the second byte of SRCIP and the DPORT. The first step is to retrieve the corresponding index file to obtain a list of matched packets for each attribute in the query. Given the wavelet matrix structure described in Section 3.2, the number of packets that satisfy the query attribute can be obtained by performing the *rank*() operation. Then we execute the *select*() operation iteratively to figure out the packet number of each matched packet. The list of matched packets for each attribute is represented by a binary array whose size is the same as the number of packets. Then, the results are merged into a single consistent list of packet numbers by performing the *and* operation. In the indexing phase, we maintain the mapping relationship between the packet number and offset. Thus, we can easily get the offsets of matched packets, which can be used to extract packets from the trace file.

Generally speaking, timestamp information is one of the essential attributes of a query. Without the limitation of the timestamp range, it is difficult to guarantee the efficiency of retrieval for high-volume network traffic. For the queries related to timestamps, we deal with them in a different way because the timestamp information is not indexed by the wavelet matrix. As the packet storage format described in Section 4.2, we create a timestamp index every N packets and add a custom timestamp index block (TIB) used to hold these indexes at the end of the trace file. The value of N is set to 10,000 empirically to reduce the timestamp index data size. This means that we can quickly find the packet number range matching a given timestamp query through a binary search in the TIB. In this way, the scope of retrieval is significantly reduced, thereby reducing the number of times to perform the *select*() operation for other query attributes.

4.4. Performance Optimization

The packet storage format designed in this paper is used to speed up the extraction of index data, and our indexing approach focuses on reducing the storage space and accelerating the packet retrieval. In this part, we will present some other techniques to improve the overall performance of the proposed system in Section 4.1.

With the rapid growth of network traffic, the traditional kernel stack is facing the problem of packet loss on high-speed network links. Thus, a large number of solutions, either software-based or hardware-based, such as Netmap [31], PF_RING [32], Data Plane Development Kit (DPDK) [33] and PacketShader [34], have been proposed to accelerate network packet processing. In the collection module of our system, DPDK is employed to capture high-speed network traffic. DPDK runs in user space and receives packets through polling mode, which eliminates the overhead of memory copy between user mode and kernel mode. The performance is optimized through CPU affinity, huge pages, memory pool management, lock-free ring queues, prefetching and some other techniques. To reduce the overhead of dynamic memory allocation, the memory pool in DPDK is used to pre-allocate memory for our application. Moreover, we focus on making full use of the multi-core hardware to parallelize different modules in our system. Based on the network card RSS technology, we distribute network traffic to multiple cores for parallel processing. The lock-free ring queue proposed in DPDK is exploited for inter-core communication. There are multiple threads for the indexing module in the system. Each indexing thread builds an index for an index attribute.

To save storage space, it is necessary to dump the packet trace files and index files to the disk in a compressed format. Due to the additional performance penalties caused by compression and decompression, we need to make a tradeoff between space and time. A comparison of several widely used compression algorithms, including gzip, bzip2, lzma, xz, lz4 and lzop, is given in Reference [35]. It can be seen from the comparison results that the lz4 compression algorithm is far superior to other compression algorithms in terms of time

consumption. For this reason, we choose the lz4 compression algorithm for compression, although it does not have an advantage in the compression ratio. Usually, the index file size is much smaller than the original trace file size, so the compression time of the index file is insignificant.

5. Performance Evaluation

In this section, we present a performance evaluation of the proposed indexing approach based on the following three metrics: time consumption for indexing, storage consumption for indexing and query response time. The platform used for our experiments is equipped with an Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40 GHz (Intel, Santa Clara, California, USA), 16 GB physical memory and 20 TB SAS HDD (TOSHIBA MG04SCA20EN, Tokyo, Japan). The operation system is Centos 7.4 (kernel version 3.10.0-693.el7.x86_64). As we know, the bitmap index is the most commonly used approach for packet indexing. WAH [16] is a representative bitmap compression algorithm. Several algorithms, such as PLWAH [17], COMPAX [14] and SECOMAX [19], are proposed on the basis of WAH to achieve further compression. From the results of Reference [19], it can be concluded that SECOMPAX outperforms other bitmap compression algorithms in terms of index size. Therefore, we choose WAH and SECOMPAX for performance comparison with the proposed IndexWM. FastBit [13] is a practical implementation of WAH, which supports efficient data processing and flexible query. The open-source code of FastBit can be found on the website (https://sdm.lbl.gov/fastbit/). We also implement SECOMPAX in FastBit. These two indexing methods are called FastBit-WAH and FastBit-SECOMPAX, respectively. Then we discuss the performance comparison between our method and pcapIndex [12], which uses COMPAX [14] for bitmap compression.

In our experiment, we build indexes on five attributes: SRCIP, DSTIP, SPORT, DPORT, L4_Proto. Six packet trace files, namely P1, P2, P3, P4, P5 and P6, are used for performance evaluation. P1 and P2 come from the real-time communication data on the egress route of a research institution, P3, P4 and P5 are randomly generated by Spirent traffic generator, and P6 is a public traffic data set from [36]. The capture length of P6 is 96 bytes. The details of the six packet datasets are shown in Table 2. In particular, since P5 and P6 contain more than 100 million packets, they are used to evaluate the indexing performance as a function of the number of packets.

Table 2. Description of six sample packet trace files.

Dataset	File Size (GB)	Number of Packets	Average Packet Size (Bytes)
P1	1.1	1,676,198	661.30
P2	2.2	3,036,636	773.43
P3	3.1	9,971,170	325.98
P4	10.0	14,952,785	712.0
P5	27	108,448,597	260.0
P6	8.4	112,414,148	1350.3

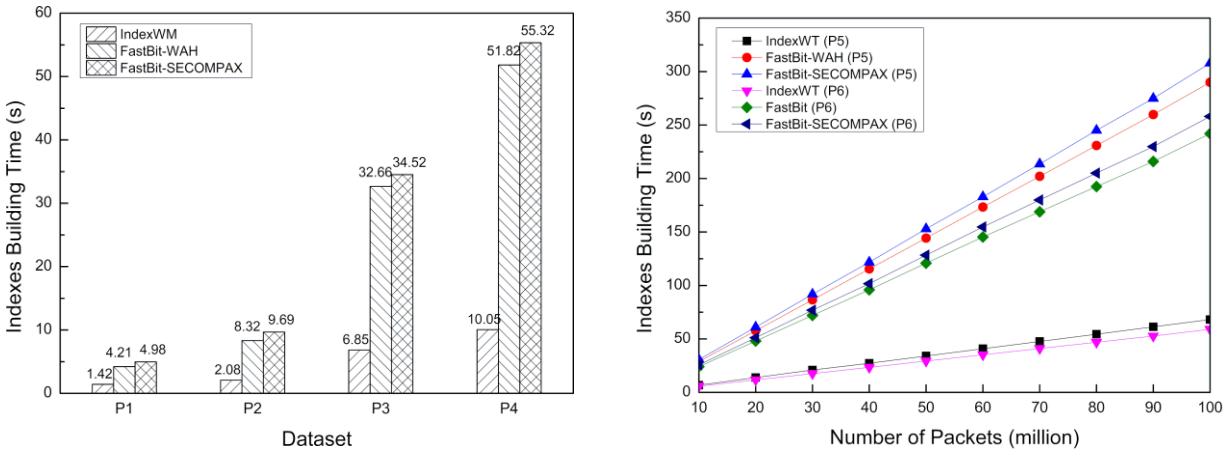
We mainly focus on the performance of the indexing and retrieval module in the proposed system, so we perform only an offline experiment for the evaluation. The collection module loads the packets from the packet trace files stored on the local disk instead of obtaining the real-time network traffic from the network interface.

5.1. Time Consumption for Indexing

First of all, we evaluate the time consumption for indexing of the proposed IndexWM, FastBit-WAH and FastBit-SECOMPAX. In order to eliminate the impact of disk I/O, we load the index data into memory in advance.

Figure 9a illustrates the comparison results of the indexes building time among IndexWM, FastBit-WAH and FastBit-SECOMPAX on four different datasets from P1 to P4. It can be seen that the time consumption of FastBit-WAH is slightly less than that of FastBit-

SECOMPAX. Next, we utilize the results of FastBit-WAH to specify the advantages of IndexWM in terms of time consumption for indexing. The time consumption of IndexWM for indexing on datasets P1, P2, P3 and P4 is 1.42, 2.08, 6.85 and 10.05 s, respectively, while the time consumption of FastBit-WAH is 4.21, 8.32, 32.66 and 51.82 s, respectively. Figure 9b shows the time consumption of these three methods on datasets P5 and P6 as a function of the number of packets. In our experiment, the number of packets changes from 10 million to 100 million with an increment of 10 million. For the simulated network traffic P5, the time consumption of IndexWM ranges from 6.81 s to 68.15 s, while that of FastBit-WAH ranges from 28.85 s to 289.96 s. For the actual network traffic P6, the time consumed by IndexWM ranges from 5.86 s to 59.12 s, whereas the time consumed by FastBit-WAH ranges from 24.12 s to 242.10 s. We can find out that the time consumption of these methods almost increases linearly with the increasing number of packets. However, in the case of the same number of packets, it takes less time to build indexes on P6 than on P5 for these three methods. This is because the randomly generated network traffic P5 has a larger alphabet size than P6 for index attributes. It can be concluded that IndexWM takes less time to build indexes than FastBit-WAH by about 68% to 76% for the packet trace files in Table 2. All the results above demonstrate that the IndexWM consistently outperforms FastBit-WAH and FastBit-SECOMPAX in terms of the time consumption for indexing.



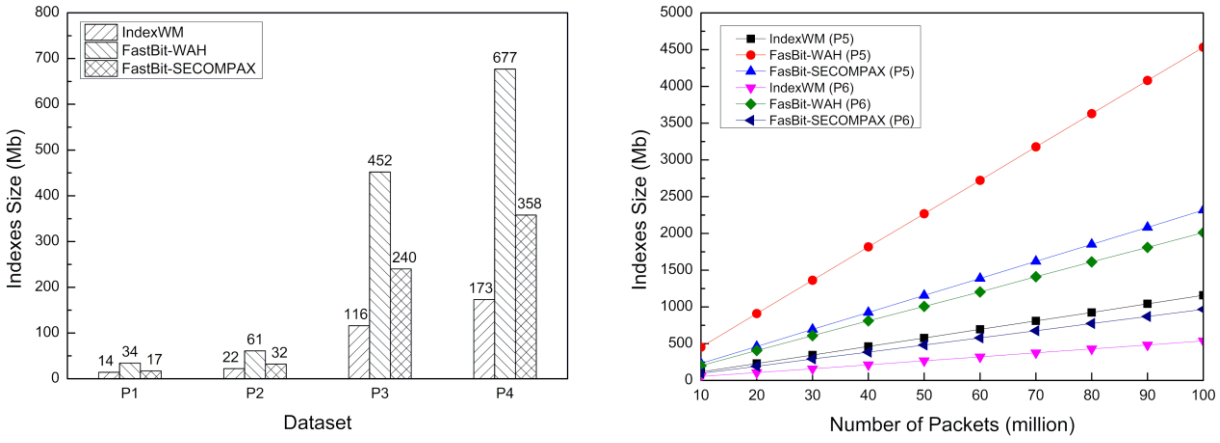
(**a**) (**b**)

Figure 9. A comparison of time consumption for indexing. (**a**) Time consumption on datasets P1, P2, P3 and P4; (**b**) time consumption on datasets P5 and P6 as a function of the number of packets.

5.2. Storage Consumption for Indexing

Indexing can improve the efficiency of packet retrieval, but at the same time, it also brings additional storage space consumption, which refers to the cost of dumping index files to disk. Figure 10a illustrates the storage consumption for indexing on the four packet datasets in Table 2. Compared with FastBit-WAH, FastBit-SECOMPAX reduces the size of the index for datasets P1, P2, P3 and P4 by about 50%. While compared with the FastBit-WAH, our IndexWM saves about 58.8%, 63.9%, 74.3% and 74.4% disk space on datasets P1, P2, P3 and P4, respectively. Figure 10b presents the index size for datasets P5 and P6 as a function of the number of packets. Next, we take FastBit-SECOMPAX and IndexWM as examples to explain in detail. For the simulated network traffic P5, the disk space consumption of IndexWM ranges from 116 MB for 10 million packets to 1159 MB for 100 million packets, while that of FastBit-SECOMPAX ranges from 232 MB to 2318 MB. As for the actual network traffic P6, the disk space consumed by IndexWM is from 54 MB for 10 million packets to 538 MB for 100 million packets, whereas the disk space consumed by FastBit-SECOMPAX is from 97 MB for 10 million packets to 967 MB for 100 million packets. Similar to the time consumption, the storage consumption for indexing also increases

almost linearly with the increased number of packets. Moreover, these three methods achieve better performance on P6. This can be explained by the fact that the alphabet size is smaller, and the compression ratio of bitmaps is higher for actual network traffic. Based on the above results, IndexWM performs better than FastBit-WAH and FastBit-SECOMPAX in terms of storage space consumption.



(**a**) (**b**)

Figure 10. A comparison of storage consumption for indexing. (**a**) Storage consumption on datasets P1, P2, P3 and P4; (**b**) storage consumption on datasets P5 and P6 as a function of the number of packets.

5.3. Query Response Time

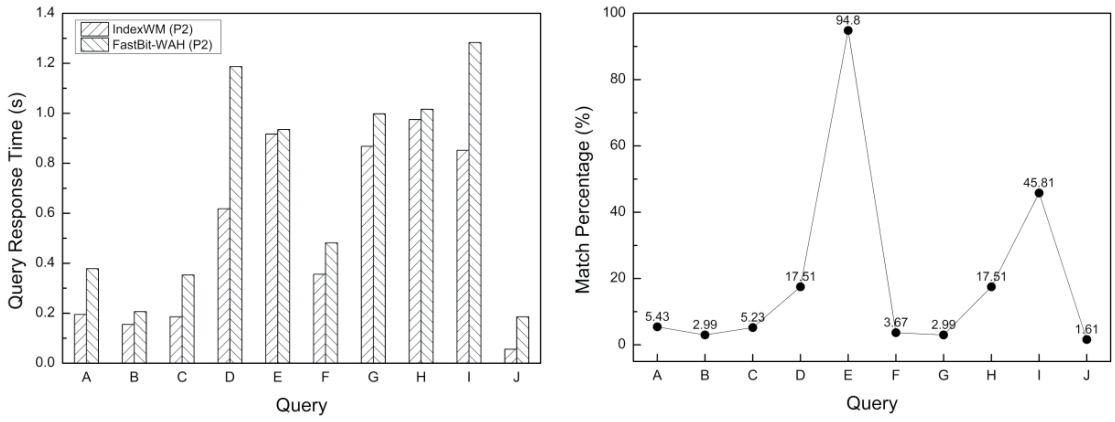
Query response time is one of the most important indicators of packet retrieval efficiency. In our experiment, the query response refers to the number and the corresponding offset of the packets that meet the query conditions. As shown in Table 3, three types of queries are considered for the evaluation of query response time. To illustrate the effectiveness and practicality of the proposed indexing scheme for actual network traffic, P2 and P6 are selected for query performance evaluation. For a single attribute query, we generate a query on each index attribute according to the values of the attribute in our dataset. The multi-attribute query can be obtained by the intersection or union of several single-attribute queries. As for the wildcard query, two queries are generated using the values of SRCIP and DSTIP, respectively. We generate a total of ten queries, including five single-attribute queries, three multi-attribute queries, and two wildcard queries. These queries are denoted by A to J, where $E = \{A \cap C\}$, $F = \{B \cap D\}$ and $G = \{D \cap E\}$.

Table 3. Example of three types of queries.

Types	Description	Examples
Single	Exact match on a single attribute	SRCIP = "10.10.0.80"
Multiple	Exact match on multiple attributes	SRCIP = "10.10.0.80" AND DPORT = 80
Wildcard	Wildcard match on a single attribute	SRCIP = "10.10.0.$*$"

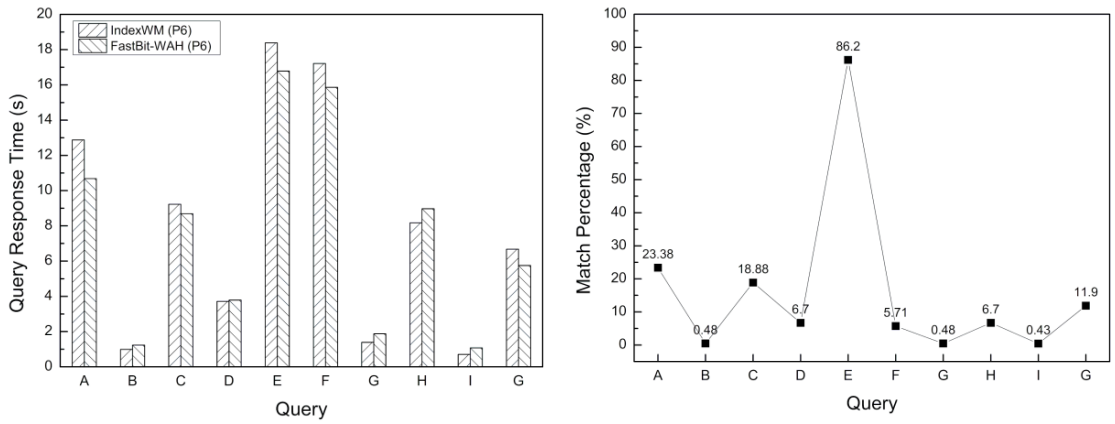
In our experiment, the query time depends on the time to retrieve the indexes and merge the results, which is usually closely related to the specific encoding method. The main purpose of various bitmap compression algorithms based on WAH, such as PLWAH, COMPAX and SECOMPAX, is to reduce the size of the index while ensuring comparable query efficiency. This can be proved in References [17,19]. Therefore, FastBit-WAH is selected as a representative comparison scheme to evaluate the query performance. The results of query response time for various queries on P2 are shown in Figure 11a. In

experiments using small dataset P2, queries A, B, C, D and E extract 164,982, 90,946, 158,777, 531,774 and 2,878,828 packets from P2, respectively. Query H, which is the intersection of query D and query E, returns the same number of matched packets as query D. Figure 11b shows the percentage of matched packets to the total number of packets. It can be seen from Figure 11a that IndexWM has a faster response speed than FastBit-WAH, especially for those single-attribute queries that only retrieve a small number of packets, such as queries A, B and C. The performance gap will become smaller when the query returns a large number of matched packets. This is because the more packets are extracted, the more select operations are required for IndexWM. This case is more significant in our experiments using a large dataset, P6. Figure 12a gives the evaluation results of query response time on P6. For dataset P6, the total number of packets exceeds 110 million. As shown in Figure 12b, the percentages of matched packets for single-attribute queries A, B, C, D and E are 23.38%, 0.48%, 18.88%, 6.70% and 86.2%, respectively. For queries A, C and E, more than 10 million packets are retrieved. As a result, the response speed of IndexWM is slower than FastBit-WAH. For queries B, G and I, IndexWM is faster than FastBit-WAH because the percentage of matched packets is very low.



(**a**) (**b**)

Figure 11. A comparison of query response time. (**a**) Query response time on dataset P2; (**b**) the percentage of matched packets on dataset P2.



(**a**) (**b**)

Figure 12. A comparison of query response time. (**a**) Query response time on dataset P6; (**b**) the percentage of matched packets on dataset P6.

According to the above results, we can conclude that IndexWM outperforms FastBit-WAH in terms of response time when the number of extracted packets is small. However, the result may be the opposite in the case of extracting a large number of packets. In our experiment, the timestamp attribute is not taken into consideration. However, in the actual network traffic collection and retrieval system, the timestamp is usually a necessary query condition, which can reduce the retrieval range. In most cases, the number within the timestamp range will not be as high as a million or even higher. Thus, in terms of query response time, IndexWM is still more efficient than FastBit-WAH in the practical system.

5.4. Performance Evaluation Discussion

According to Reference [26], the basic form of wavelet matrix requires $n\ lg\ \sigma + o(n\ lg\ \sigma)$ bits of space and can be built in $O(n\ lg\ \sigma)$ time, where n represents the number of packets and σ represents the alphabet size of the index attribute. As we can see from Figures 9b and 10b, these two performance metrics are approximatively linear with the number of packets for the same packet dataset. This is because the alphabet size changes very little when the number of packets varies from 10 million to 100 million for the same packet dataset. Besides, all these schemes have better performance on actual network traffic. This phenomenon is closely related to the statistical characteristics of index attributes in different packet traces. Taking the DPORT attribute as an example, the alphabet size of DPORT for 10 million packets in P6 is 32,871, while the size for 10 million packets in P5 increases to 65,536. This may also lead to a better compression ratio for bitmaps. It is similar to other index attributes. Figure 9a,b shows that the wavelet matrix can be built within 1 s for one million packets. Compared to the bitmap index schemes, this can meet the short construction time requirement of our network traffic collection and retrieval system. As illustrated in Figure 10a,b, the index data size of IndexWM corresponds to about 1% to 3% compared to the volume of the original packet traces in Table 2.

As described in Section 4.3, retrieving specific packets from the indexes requires performing a *rank* operation and several *select* operations on the wavelet matrix. These two operations can be performed in time $O(lg\ \sigma)$. It is obvious that the cost is only related to the alphabet size. Considering the query response time of a single-attribute query on a single field, it depends on the number of extracted packets, which determines the number of times that the select operation is performed. However, in practice, query results are usually obtained by combining the results of multiple subqueries. For example, if we query on SRCIP with a certain value, we need to retrieve the index files corresponding to the four bytes of SRCIP and merge the results. This will incur extra time overhead associated with the number of packets. Figure 11a indicates that IndexWM takes less retrieving time than FastBit-WAH in the case of extracting a small number of packets. However, as shown in Figure 12a, the performance of IndexWM decreases significantly if the number of extracted packets increases dramatically. In this case, FastBit-WAH has a better performance than IndexWM. To avoid this, the timestamp is one of the essential query attributes in our implementation for the purpose of reducing the retrieval scope. Thus, IndexWM will become a good solution for retrieving specific packets from a limited retrieval scope.

PcapIndex [12] is an efficient tool for packet indexing. Since the performance comparison results of WAH and COMPAX are provided in Reference [12], we can make an indirect comparison with pcapIndex. For storage consumption, pcapIndex is up to about 50% smaller than FastBit, and our IndexWM is about 60% to 75% smaller than FastBit. In terms of the time consumption for indexing, pcapIndex requires about 5% less time than FastBit, whereas IndexWM requires about 68% to 76% less time than FastBit. The query response time depends on the specific query, but IndexWM shows a remarkable performance improvement for retrieving a small number of packets compared to FastBit. Considering these three performance metrics, it is clear that our IndexWM also outperforms pcapIndex for packet indexing.

Compared to popular bitmap indexing algorithms, our indexing scheme IndexWM performs better, especially in terms of storage consumption and construction time. It is

well known that bitmap indexing has high retrieval efficiency because it supports efficient bitwise operations between different bitmap indexes. The proposed IndexWM has comparable or even better performance when extracting a small number of packets. Typically, our system only retrieves only a small number of packets within a small timestamp range. In this case, our system can achieve high retrieval efficiency. From what has been discussed above, we can conclude that IndexWM is perfectly viable for packet indexing in the network traffic collection and retrieval system.

6. Conclusions

Nowadays, network attacks are becoming increasingly sophisticated, which makes them often unable to be detected in real time. Therefore, it is becoming increasingly necessary to record all the content of network traffic for retrospective analysis. Historical network traffic retrieval is becoming increasingly essential in network monitoring, network forensics and many other research fields. In order to quickly retrieve specific packets from high-volume network traffic, indexing for packets has proven to be a highly effective solution, but it brings challenges of storage consumption and construction time for indexes. The wavelet matrix is a compact and efficient data structure to represent a sequence and answer some queries on it. In this paper, we propose an efficient indexing scheme IndexWM based on the wavelet matrix to achieve short construction time, low storage consumption and high retrieval efficiency. Moreover, we design a packet storage format based on PcapNG to reduce the difficulty of extracting index data from packet trace files and reduce the scope of retrieval by appending timestamp indexes to the end of the file. Based on the above optimization schemes, we implemented the network traffic collection and retrieval system. We use different types of network traffic, such as actual network traffic from a research institution and network traffic generated by Spirent, to evaluate the performance of the proposed IndexWM. The results indicate that IndexWM requires about 68% to 76% less time than FastBit-WAH and saves about 60% to 75% storage space compared to FastBit-WAH. Moreover, even for large datasets, the IndexWM has excellent retrieval efficiency when only extracting a small number of packets. We can conclude that the proposed IndexWM performs significantly better than FastBit-WAH and FastBit-SECOMPAX in terms of the indexing rate, storage consumption and retrieval efficiency. In the future, we will select more fields in packets for indexing, enabling fine-grained and flexible queries. The corresponding wavelet matrix is expected to be small in size and has a simple structure because each of these fields has a small alphabet size.

Author Contributions: Conceptualization, C.J. and J.W.; methodology, C.J. and J.W.; software, C.J.; writing—original draft preparation, C.J.; writing—review and editing, C.J., Y.L. and J.W.; supervision, J.W.; project administration, J.W.; funding acquisition, J.W. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by the Strategic Leadership Project of the Chinese Academy of Sciences: SEANET Technology Standardization Research System Development (Project No. XDC02070100).

Acknowledgments: We would like to express our gratitude to Jinlin Wang and Yang Li for their meaningful support for this work.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Joshi, R.C.; Pilli, E.S. Network Forensic Acquisition. In *Fundamentals of Network Forensics: A Research Perspective*; Springer London: London, UK, 2016; pp. 97–106.
2. Tcpdump/Libpcap Public Repository. Available online: https://www.tcpdump.org (accessed on 3 November 2020).
3. PCAP Next Generation Dump File Format. Available online: https://github.com/pcapng/pcapng (accessed on 3 November 2020).
4. Goyal, P.; Goyal, A. Comparative study of two most popular packet sniffing tools-Tcpdump and Wireshark. In Proceedings of the 2017 9th International Conference on Computational Intelligence and Communication Networks (CICN), Girne, Cyprus, 16–17 September 2017; pp. 77–81.

5. Chandrasekaran, S.; Cooper, O.; Deshpande, A.; Franklin, M.J.; Hellerstein, J.M.; Hong, W.; Krishnamurthy, S.; Madden, S.R.; Reiss, F.; Shah, M.A. TelegraphCQ: Continuous dataflow processing. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, CA, USA, 10–12 June 2003; p. 668.
6. Desnoyers, P.J.; Shenoy, P. Hyperion: High volume stream archival for retrospective querying. In Proceedings of the 2007 USENIX Annual Technical Conference, Santa Clara, CA, USA, 17–22 June 2007; pp. 1–14.
7. Kornexl, S.; Paxson, V.; Dreger, H.; Feldmann, A.; Sommer, R. Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic. In Proceedings of the 5th ACM SIGCOMM Conference on Internet measurement, Berkeley, CA, USA, 19–21 October 2005; pp. 267–272.
8. Chen, Z.; Ruan, L.; Cao, J.; Yu, Y.; Jiang, X. TIFAflow: Enhancing traffic archiving system with flow granularity for forensic analysis in network security. *Tsinghua Sci. Technol.* **2013**, *18*, 406–417. [CrossRef]
9. Xie, G.; Su, J.; Wang, X.; He, T.; Zhang, G.; Uhlig, S.; Salamatian, K. Index–Trie: Efficient archival and retrieval of network traffic. *Comput. Netw.* **2017**, *124*, 140–156. [CrossRef]
10. Geambasu, R.; Bragin, T.; Jung, J.; Balazinska, M. On-demand view materialization and indexing for network forensic analysis. In Proceedings of the 3rd USENIX International Workshop on Networking Meets Databases, Cambridge, MA, USA, 10 April 2007; pp. 1–7.
11. Pcap-Index. Available online: https://github.com/taterhead/PCAP-Index (accessed on 3 November 2020).
12. Fusco, F.; Dimitropoulos, X.; Vlachos, M.; Deri, L. pcapIndex: An index for network packet traces with legacy compatibility. *ACM Sigcomm Comput. Commun. Rev.* **2012**, *42*, 47–53. [CrossRef]
13. Wu, K.; Ahern, S.; Bethel, E.W.; Chen, J.; Childs, H.; Cormier-Michel, E.; Geddes, C.; Gu, J.; Hamann, B.; Koegler, W.; et al. FastBit: Interactively Searching Massive Data. *J. Phys. Conf. Ser.* **2009**, *180*, 012053. [CrossRef]
14. Fusco, F.; Stoecklin, M.P.; Vlachos, M. NET-FLi: On-the-fly compression, archiving and indexing of streaming network traffic. *Proc. VLDB Endow.* **2010**, *3*, 1382–1393. [CrossRef]
15. Xu, X.Y.; Li, M.S. Bitmap Index Design and Implementation. *J. Netw. New Media* **2006**, *27*, 188–191.
16. Wu, K.; Otoo, E.J.; Shoshani, A. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.* **2006**, *31*, 1–38. [CrossRef]
17. Deliège, F.; Pedersen, T.B. Position list word aligned hybrid: Optimizing space and performance for compressed bitmaps. In Proceedings of the 13th International Conference on Extending Database Technology, Lausanne, Switzerland, 22–26 March 2010; pp. 228–239.
18. Colantonio, A.; Pietro, R.D. Concise: Compressed 'n' Composable Integer Set. *Inf. Process. Lett.* **2010**, *110*, 644–650. [CrossRef]
19. Wen, Y.; Chen, Z.; Ma, G.; Cao, J.; Zheng, W.; Peng, G.; Li, S.; Huang, W. SECOMPAX: A bitmap index compression algorithm. In Proceedings of the 23rd International Conference on Computer Communication and Networks (ICCCN), Shanghai, China, 4–7 August 2014; pp. 1–7.
20. Li, C.; Chen, Z.; Zheng, W.; Wu, Y.; Cao, J. BAH: A Bitmap Index Compression Algorithm for Fast Data Retrieval. In Proceedings of the 2016 IEEE 41st Conference on Local Computer Networks (LCN), Dubai, UAE, 7–10 November 2016; pp. 697–705.
21. Zheng, W.; Liu, Y.; Chen, Z.; Cao, J. CODIS: A New Compression Scheme for Bitmap Indexes. In Proceedings of the 2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Beijing, China, 18–19 May 2017; pp. 103–104.
22. Chen, Z.; Wen, Y.; Cao, J.; Zheng, W.; Chang, J.; Yinjun, W.; Ma, G.; Hakmaoui, M.; Peng, G. A survey of bitmap index compression algorithms for Big Data. *Tsinghua Sci. Technol.* **2015**, *20*, 100–115. [CrossRef]
23. Deri, L.; Lorenzetti, V.; Mortimer, S. Collection and exploration of large data monitoring sets using bitmap databases. In Proceedings of the Second International Conference on Traffic Monitoring and Analysis, Zurich, Switzerland, 7 April 2010; pp. 73–86.
24. Li, J.; Ding, S.; Xu, M.; Han, F.; Guan, X.; Chen, Z. TIFA: Enabling Real-Time Querying and Storage of Massive Stream Data. In Proceedings of the Second International Conference on Networking and Distributed Computing, Beijing, China, 21–24 September 2011; pp. 61–64.
25. Fusco, F.; Vlachos, M.; Stoecklin, M.P. Real-time creation of bitmap indexes on streaming network data. *VLDB J.* **2012**, *21*, 287–307. [CrossRef]
26. Claude, F.; Navarro, G.; Ordóñez, A. The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Syst.* **2014**, *47*, 15–32. [CrossRef]
27. Grossi, R.; Gupta, A.; Vitter, J. High-order entropy-compressed text indexes. In Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Baltimore, MD, USA, 12–14 January 2003; pp. 841–850.
28. Navarro, G. Wavelet trees for all. *J. Discret. Algorithms* **2014**, *25*, 2–20. [CrossRef]
29. González, R.; Grabowski, S.; Mäkinen, V.; Navarro, G. Practical Implementation of Rank and Select Queries. In Proceedings of the 4th Workshop on Efficient and Experimental Algorithms (WEA), Santorini Island, Greece, 10–13 May 2005; pp. 27–38.
30. Raman, R.; Raman, V.; Rao, S.S. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, USA, 6–8 January 2002; pp. 233–242.
31. Rizzo, L. Netmap: A novel framework for fast packet I/O. In Proceedings of the 2012 USENIX Conference on Annual Technical Conference, Boston, MA, USA, 13–15 June 2012; p. 9.

32. PF_RING. Available online: http://www.ntop.org/products/packet-capture/pf_ring/ (accessed on 3 November 2020).
33. Data Plane Development Kit. Available online: http://www.dpdk.org (accessed on 3 November 2020).
34. Han, S.; Jang, K.; Park, K.; Moon, S. PacketShader: A GPU-accelerated software router. In Proceedings of the ACM SIGCOMM 2010 Conference, New Delhi, India, 30 August –3 September 2010; pp. 195–206.
35. Quick Benchmark. Available online: https://catchchallenger.first-world.info/wiki/Quick_Benchmark:_Gzip_vs_Bzip2_vs_LZMA_vs_XZ_vs_LZ4_vs_LZO (accessed on 3 November 2020).
36. 201904090815.Pcap. Available online: http://mawi.wide.ad.jp/mawi/ditl/ditl2019-G/201904090815.html (accessed on 3 November 2020).