Toward Efficient Querying of Compressed Network Payloads

Teryl Taylor
UNC Chapel Hill
tptaylor@cs.unc.edu

Scott E. Coull
RedJack
scott.coull@redjack.com

Fabian Monrose
UNC Chapel Hill
fabian@cs.unc.edu

John McHugh
RedJack
john.mchugh@redjack.com

Abstract

Forensic analysts typically require access to application-layer information gathered over long periods of time to completely investigate network security incidents. Unfortunately, storing longitudinal network data is often at odds with maintaining detailed payload information due to the overhead associated with storing and querying such data. Thus, the analyst is left to choose between coarse information about long-term network activities or brief glimpses of detailed attack activity. In this paper, we take the first steps toward a storage framework for network payload information that provides a better balance between these two extremes. We take advantage of the redundancy found in network data to aggregate payload information into flexible and efficiently compressible data objects that are associated with network flows. To enable interactive querying, we introduce a hierarchical indexing structure for both the flow and payload information, which allows us to quickly prune irrelevant data and answer queries directly from the indexing information. Our empirical results on data collected from a campus network show that our approach can significantly reduce the volume of the stored data, while simultaneously preserving the ability to perform detailed queries with response times on the order of seconds.

1 Introduction

A complete incident response strategy for network attacks includes short-term detection and mitigation of the threat, as well as a broad forensic process. These forensic investigations attempt to discover the root cause of the attack, its impact on other resources, and how future attacks may be prevented. To perform these forensic tasks, however, a security analyst needs long-term, detailed information on the activities of monitored resources, which often includes the application-layer communications found within packet payloads, such as DNS queries and HTTP responses.

These requirements are perhaps best illustrated by the recent tide of attacks by so-called advanced persistent threats on major corporations and government contractors (eg., Google, RSA and Oakridge National Laboratory [30]). In each case, a single security breach (e.g., phishing or browser exploit) was used to gain a foothold within an otherwise secure network, which remained undetected for weeks while computing resources were accessed and proprietary information was exfiltrated. The only way to grasp the full impact of these types of attacks is to trace through each step and examine the associated communications information, including DNS names, HTTP redirection, and web page requests.

The obvious problem is that modern enterprise networks can easily produce terabytes of packet-level data each day, which makes efficient analysis of payload information difficult or impossible even in the best circumstances. Although several approaches have been developed to capture and store network flow (i.e., NetFlow) connection summaries (e.g., [6, 8, 12, 13, 16, 24]), these systems are inappropriate to the task at hand since they necessarily discard the application-layer information that is so important to the forensic analysis process. Furthermore, packet payload data introduces several unique challenges that are not easily addressed by traditional database systems. For one, there are thousands of application-layer protocols, each with their own unique data schemas[1]. Many of these protocol schemas are also extremely dynamic with various combinations of fields and use of complex data types, like arrays. Of course, there are also cases where the existence of proprietary or unknown application-layer protocols may prevent us from parsing the payload at all, and yet we still must be able to support analysis of such data.

Using a standard relational database solution in this scenario, be it column or row-oriented, would require thousands of structured tables with tens or hundreds of sparsely populated columns. Consequently, these tradi-

[1]As a case in point, Wireshark can parse over 10,000 protocols, fifty percent of which have more than 20 fields.

tional approaches introduce significant storage overhead, require complex join operations to relate payload fields to one another, and create bottlenecks when querying for complex payload data (e.g., arrays of values). While the availability of distributed computing frameworks [5, 11] help tackle some of these issues, the query times required for interactive analysis might only be realized when hundreds of machines are applied to the task – resources that may be unavailable to many analysts. Moreover, even when such resources are available, data organization remains a critical issue. Therefore, the question of how to enable efficient and interactive analysis of long-term payload data remains.

In this work, we address these challenges by extending existing network flow data storage frameworks with a set of summary payload objects that are highly compressible and easy to index. As with many of the network flow data stores, we aggregate network data into bidirectional flows and store the network-level flow data (e.g., IP addresses, ports, timestamps) within an indexed, column-oriented database structure. Our primary contribution, however, lies in how we store and index the payload information, and then attach that information to the column-oriented flow database. By using our proposed storage technique we are able to reduce the volume of data stored and still maintain pertinent payload information required by the analyst. While our ultimate goal is to create a data store for arbitrary payload content, we begin our exploration of the challenges of doing so by focusing on the storage and querying of packet payloads with well-defined header and content fields (e.g., DNS and HTTP). These protocols have many of the same issues described above, but represent a manageable step toward a more general approach to efficient payload querying.

To achieve our performance goals, we move away from the relational model of data storage and instead roll-up packet payloads into application-layer summary objects that are encoded within a *flexible, self-describing object serialization framework*. In this regard, our contributions are threefold. *(1)* We index the summary objects by treating them as documents with the application-layer field-value pairs acting as words that can be indexed and efficiently searched with bitmap indexes. This approach allows us to store and index heterogenous payload information with highly-variable protocol schemas — a task that is difficult, if not impossible, with previous network data storage systems. The bitmap indexes also make it possible to relate the payload objects to network flow records using only simple bit-wise operations. (*2*) We introduce a *hierarchical indexing* scheme that allows us to answer certain queries directly from in-memory indexes without ever having to access the data from disk, thereby enabling the type of lightweight iterative querying needed for forensics tasks. (*3*) We take advantage of the inherent redundancy found in many types of application-layer traffic to devise an effective *dictionary-based string compression* scheme for the payload summary objects, while standard block-level lossless compression mechanisms are utilized to take advantage of inter-field relationships in objects in order to minimize storage overhead and disk access times.

Given the privacy-related challenges in gaining access to network data containing complete payload information, we chose to evaluate our approach with two datasets containing DNS and truncated HTTP data that we collected at the University of North Carolina (UNC). The first dataset contains over 325 million DNS transactions collected over the course of five days. We use that dataset to compare our performance to that of the `SiLK` network flow toolkit [25] and Postgres SQL database, both of which are commonly used within the network analysis community to perform security and forensic analysis tasks [14, 23]. The second dataset is collected over 2.5 hours and contains over 11 million DNS and HTTP connections making up 400GB of packet trace data. We use that dataset to explore how well our approach handles heterogenous objects. The results of our experiments show that our approach reduces the volume of the DNS traffic that need be stored by over 38% and HTTP traffic by up to 97%, while still preserving the ability to perform detailed queries on the data in a matter of seconds. By comparison, using a relational database approach containing tables for the flow and payload schemas *increases* the data volume by over 400% and results in extremely slow response times.

2 Background and Related Work

One of the most common data storage approaches for network data is the so-called scan-and-filter paradigm. In this approach, raw network data is stored in flat files with standardized formats. To fulfill a given query, the analysis tool reads the entire data file from disk, scans through the contents, and applies one or more filtering criteria to select the pertinent records. The `SiLK` toolkit [25] and `TCPDump` are examples of this approach for network flow logs and packet traces, respectively. The ownside is that it requires the entire dataset to be read from disk, which is a relatively low-bandwidth process.

The simple scan-and-filter approach can be improved through the use of indexing methodologies to target disk access to only those records that meet the requirements of the user's query. As an example, the `TimeMachine` system [17] stores truncated packet data and uses hash-based indexing of packet headers to improve performance. However, `TimeMachine` is unable to index payload contents and each file found with the index must still be read from disk in its entirety even if only a single packet is requested.

One natural extension to the above is to divide packet data across machines and parallelize queries using distributed computing frameworks, such as `MapReduce` [11] or `Hadoop` [5]. Unfortunately, the computing resources necessary to take advantage of these frameworks are not always easily accessible to forensic analysts, and even organizations with significant computational resources (e.g., Google, Twitter) have acknowledged that simply parallelizing brute force data retrieval methods will not provide adequate performance [19]. In fact, Google developed a non-relational, column-oriented data store, called Dremel [18], specifically to address the issue of efficient data organization in distributed computing environments. Even so, Dremel does not support indexes and still resorts to scan-and-filter approaches to match attributes from different columnar files.

Beyond the simple scan-and-filter approach, traditional relational databases are also often used to store and query network data [9, 15]. A relational database allows data that conforms to the same static schema to be grouped together into tables, often with each row of data stored contiguously on disk (i.e., row-oriented). These databases provide a generic framework for storing a wide variety of data, and offer advanced indexing features to pinpoint exactly the records requested by the user. Although the relational databases offer these useful features, the row-oriented organization of the data is inefficient for network data queries that inspect only a small subset of discontiguous fields [6]. Additionally, the rigid schema structure of the database would result in hundreds of tables (one per application-layer protocol) with sparsely filled columns that require the use of expensive join operations to retrieve payload fields.

A more efficient approach for storing network data is to use a column-oriented database, where each column is stored independently as a sequence of contiguous values on disk. Column-oriented databases follow the same relational structure of rigid data tables, but enable efficient queries over multiple fields. For that reason, several column-based data stores have been developed specifically for network flow data [6, 12, 16, 24]. These systems create columns for each of the standard fields found within all flow records (e.g. IP, port, etc.), and then apply indexing methods to quickly answer multidimensional queries over several fields. When considering the variety and variability of payload data, however, the column-oriented approach suffers from some of the same shortcomings as row-oriented relational databases; namely, they require thousands of sparsely populated columns and table joins to store payload data for the most common application-layer protocols.

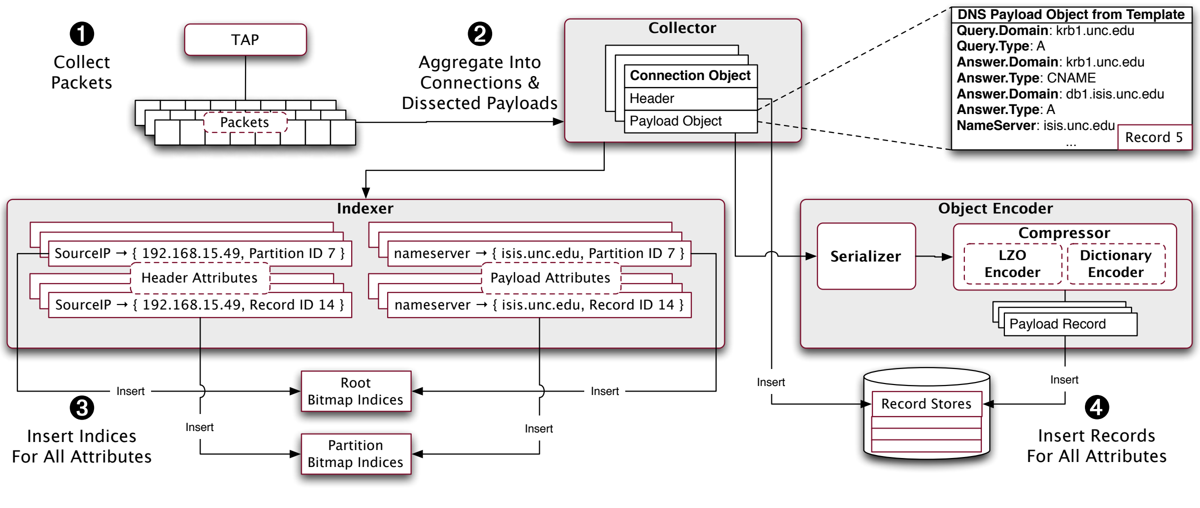
To our knowledge, only one other work has examined the problem of enabling forensic analysis of packet payload data. In this work, Ponec et al. [21] discuss how to capture short, overlapping windows of payload data and encode that data in memory-efficient data structures, similar to Bloom filters. The analyst then queries the data structure for a known byte sequence to determine if the sequence occurs within the data. While this solution can significantly reduce the storage requirements for payload data, it limits opportunities for exploratory data analysis since it can only be used to determine whether a previously-known byte sequence exists and cannot actually return the raw data (or any summaries thereof).

In designing our approach we purposely move away from the rigid relational database paradigm and instead draw inspiration from document-oriented databases (e.g., [1, 3]) that store self-describing document objects with flexible data schemas. By using this approach, we accommodate the strongly heterogenous nature of payload data and the variability of application-layer protocols. We combine these technologies with proven column-oriented database technologies, like bitmap indexes [13], to create a hybrid system that combines efficient storage of flow data in a column-oriented format with flexible storage of packet payload data.

3 Approach

The primary goal of our network data storage system is to enable fast queries over both network flow data and packet payload information. Our intention in this work is to develop an approach that takes advantage of the properties of network data to intelligently reduce workload and make payload analysis accessible to analyst with limited resources. As a result, we focus our evaluation on a single machine architecture to gain a better understanding of the key bottlenecks in the query process so that they may be mitigated. At the same time, we ensure our framework can naturally scale to take advantage of distributed computing resources by partitioning the data and developing a hierarchical system of indexes for both flow and payload information.

Before delving into the specifics of our approach, we first provide a basic overview of the storage and retrieval components. In the storage component, shown in Figure 1, incoming packets and their payloads are collected into bidirectional network flows in a manner similar to that proposed by Maier et al. [17]. After a period of inactivity, the flows are closed and packet payloads are aggregated. If the application-layer protocol contained within the aggregated payload is known to the storage system, it dissects the protocol into its constituent fields and their values, resulting in a set of *flow* and *payload* fields. When a sufficient number of closed flows have accumulated, the flows are indexed and written to one of many horizontal data partitions on the disk. The flow fields are written into a column-oriented store, while the associated payload attributes are serialized and compressed into flexi-



❶ Collect Packets
TAP
Packets
❷ Aggregate Into Connections & Dissected Payloads
Collector
Connection Object
Header
Payload Object
DNS Payload Object from Template
Query.Domain: krb1.unc.edu
Query.Type: A
Answer.Domain: krb1.unc.edu
Answer.Type: CNAME
Answer.Domain: db1.isis.unc.edu
Answer.Type: A
NameServer: isis.unc.edu
...
Record 5
Indexer
SourceIP → { 192.168.15.49, Partition ID 7 }
Header Attributes
SourceIP → { 192.168.15.49, Record ID 14 }
nameserver → { isis.unc.edu, Partition ID 7 }
Payload Attributes
nameserver → { isis.unc.edu, Record ID 14 }
Object Encoder
Serializer
Compressor
LZO Encoder
Dictionary Encoder
Payload Record
Insert
Root Bitmap Indices
Partition Bitmap Indices
❸ Insert Indices For All Attributes
Record Stores
❹ Insert Records For All Attributes

Figure 1: Storing payload data.

ble payload objects. Meanwhile, the flow and payload attributes are indexed in a hierarchical indexing system where the *root index* collects coarse information about the entire data store and the *partition indexes* collect more refined information about each of the horizontal partitions that the data is split into.

The analyst queries the data store using a SQL-like language with extensions that allow queries on payload data and queries that are answered directly from the hierarchical indexes. In Figure 2, for example, the analyst issues a query to find source IP addresses and DNS query types (e.g., MX records) for all traffic with destination port 53 and domain `www.facebook.com`. The query predicates (e.g., destination port and domain) are first compared to the in-memory root index to quickly return a list of data partitions that contain at least one record that matches those predicates. Next, the indexes for the matching partitions are examined to determine the exact locations of the records that match the given query predicate. Finally, the data store seeks to the given locations and retrieves the source IP from its column and the query type from the packet payload object.

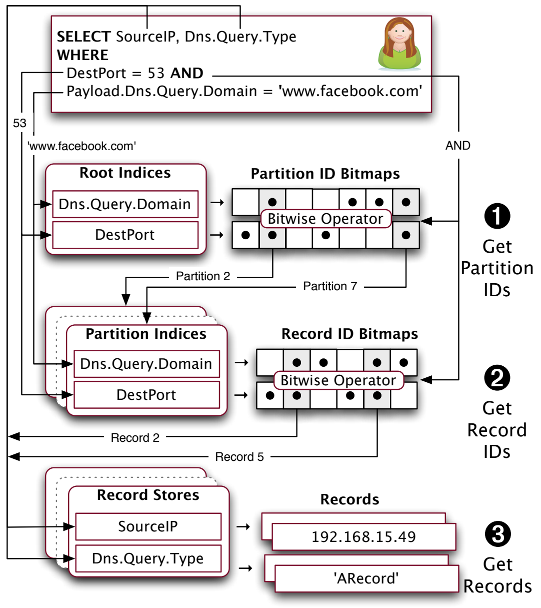


Figure 2: Processing a query.

3.1 Storage

Aggregation and Partitioning. Intelligently storing data on disk improves performance by simultaneously reducing the storage overhead and the amount of data that must be retrieved from high-latency disks. In order to reduce the data footprint, we aggregate all of the packets in a connection into network flows according to the five-tuple of source and destination IP address, source and destination ports, and protocol. As with previous approaches [6, 12, 13, 24], we utilize a column-oriented data store for the standard set of integer-based flow fields that occur in every network flow record[2], as it provides the best performance for the type of random-access queries made by forensic analysts [6, 23].

Our approach to storing payload information is to receive dissected payloads from one or more sensor platforms (e.g, Bro [28]) and extract the values from certain fields deemed important by the analyst. These field values are then stored in summary payload objects that

[2]Currently we support attributes of source and destination IP, source and destination ports, protocol, start time, duration, TCP flags, byte and packet count, but other attribute can be incorporated easily.

are instantiated from a set of application-layer protocol templates as illustrated by the DNS template example in Figure 3. Once the object is generated in memory, it is serialized to disk with a lightweight object serialization framework with self-describing data schemas [4]. The object serialization framework allows us to accommodate for the strongly heterogenous nature of payload data by individually defining the output schema for each payload object we serialize based on the extracted fields and values, rather than forcing the data to adhere to a static schema, as is the case in traditional relational databases. Moreover, if the object dissector encounters a payload for which it does not have an object template, we can still store the raw payload data in the same way by using an object with a single field (i.e., "raw payload") whose value is the byte sequence for the aggregated payload.

One unique feature of network data storage is that it does not require the updating capabilities normally associated with traditional databases. That is, once the data is written to disk, there is no need to update it. To take advantage of this property, we implement a horizontal partitioning scheme where groups of records are broken into independent partitions. In doing so, we can write all records in a partition at once, create indexes on that data, and never have to perform writes to that partition or index ever again. The use of horizontal partitioning also has the added benefit of providing natural boundaries where the data can be distributed among multiple servers or archived when space is limited.

```
struct dns_rr {
    std::string domain;
    proto::RecordType type;
    int32_t ttl;
    std::string response;
};

struct DNSMessage {
    std::string queryDomain;
    int32_t queryId;
    proto::RecordType recordType;
    array<proto::dns_rr > additional;
    array<proto::dns_rr > answer;
    array<proto::dns_rr > nameServer;
};
```

Figure 3: Example Object Definition.

Compression. Aside from aggregation of packet information into flow-based fields and payload objects, the nature of many application-layer protocols makes them particularly well-suited to the use of lossless compression mechanisms. In particular, many application-layer protocols, such as DNS and HTTP, exhibit high levels of string redundancy [10]. To take advantage of this redundancy, we apply two compression mechanisms to our payload data to ensure we need only read a minimum of information from disk.

The first compression scheme employs a dictionary-based encoding of all strings in the payload objects. In a dictionary-based encoding scheme, long byte strings are replaced with much smaller integer values that compress the space of strings accordingly. Undoubtedly, we are not the first to take advantage of this observation. Binnig et al. [7], for example, showed that string compression can be effective even for data with extremely high cardinality. However, their approach cannot be readily applied in our setting as it is not well suited for streaming data sets, like network traffic. Fortunately, we found that a rather straightforward hash-based approach that maps strings to integer values as the packets are dissected works extremely well in practice.

Our second compression mechanism uses standard Lempel-Ziv-Oberhumer (LZO) encoding compression [32] to compress the serialized payload objects after they have been dictionary-encoded. LZO is notable for its decompression speed, which is ideal for network data that will be written once and read multiple times. It is also particularly efficient at compressing highly-redundant but variable-length patterns.

As we show later, this integration of compression techniques dramatically reduces response times through reduced data footprint, and helps offset the storage overhead introduced by indexing methods that help us quickly retrieve data.

3.2 Indexing and Retrieval

Avoid Scan-and-Filter. Scan-and-filter approaches to data retrieval are slow because of the need to read the entire dataset to find records of interest. The standard solution to this problem is to apply indexing to the data to quickly determine the location of the data that satisfies the query. Construction of these indexes, however, requires careful planning in our setting. For one, network data is multi-dimensional and an analyst will often not know exactly what she is looking for until she has found it [6, 23]. Therefore, we must support indexes across all of the stored flow and payload fields, as well as combinations of those fields. Furthermore, it is also important that these indexes can be built quickly and effectively support large data sets.

In this work, we use two separate indexing mechanisms; one for flow fields and another for payload fields. For flow fields, we use bitmap-based indexes [6, 12, 13, 24]. Simply put, a bitmap index is a pair containing a field value and a bit vector where a bit at position i indicates the presence or absence of that value in record i. The bitmap indexes benefit from being highly-compressible [31], enabling combination of indexes through fast bit-wise operations (i.e., AND and OR), and allowing real-time generation as data is being stored [13]. Each flow field stored has its own bitmap index. We refer the interested reader to Deri et al. [12] for an in-depth discussion on bitmap indexes.

In contrast, packet payload data is far more difficult to index because it is composed of an arbitrary number of attributes that have high cardinality and varying length. Therefore, we cannot take the same approach used in indexing flow data. However, we can model our summary packet payload objects as documents by considering the field-value pairs (e.g., '`query domain = www.facebook.com`') as words to be indexed. Specifically, we create an inverted-term index that stores the field-value pairs across all payload objects in lexicographical order, grouped by field name. Each field-value pair is associated with a bitmap indicating the records where the pair is found, as shown in Figure 4. In this way, we can now support the ability to search for specific criteria within the payload, including wildcards (e.g., '`Http.UserAgent:Mozilla/5.0*`'). Additionally, the payload bitmap index can be combined with those of the network flow columns to tie both data stores together.

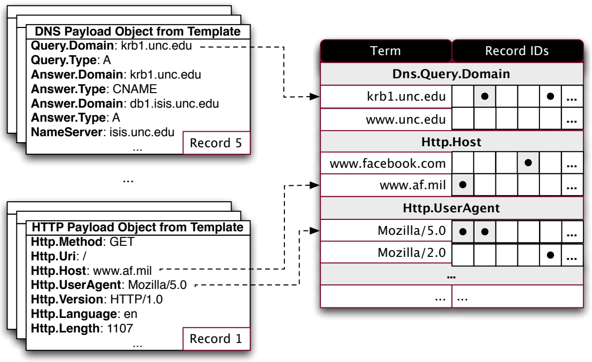


Figure 4: Indexing packet payload content.

Abstraction via Indirection. Although it is possible to use a single, monolithic index for each of the flow and payload fields in the data store, they are likely to become unmanageably large after a few million records have been added. Once the cardinality of those indexes becomes sufficiently large, even reading the index from disk would take a non-trivial amount of time and would likely be difficult to fit into memory. Another approach would be to have indexes associated with each of the horizontal partitions of the data. In that case, one would incur significant disk access penalties by reading the indexes for every partition even if it contains no records of interest. To address these issues, we designed a hierarchical indexing approach, similar to that of Sinha and Winslett [26], to efficiently exclude partitions that cannot satisfy a query. Our framework uses a root index to pick candidate partitions, and then processes the partition-level indexes to resolve queries to specific records.

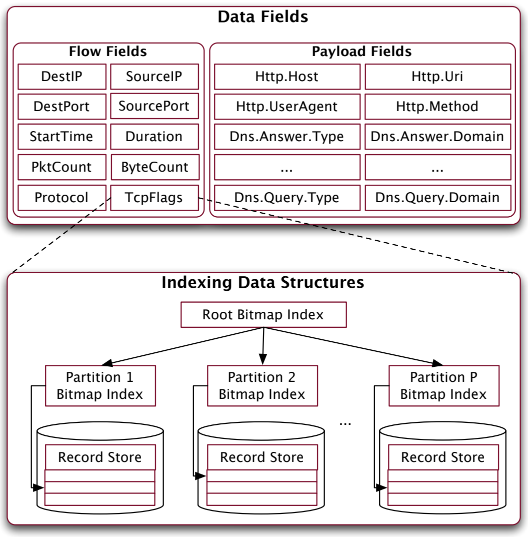


Figure 5: Structure of the datastore

Each of the network flow columns, plus the payload store, has its own root index for locating partitions that satisfy queries, as well as partition-level indexes for locating matching records. Both the root and partition-level indexes are organized as described above, with each value associated with a bitmap indicating the partitions and records it occurs in, respectively. The key difference between the root and partition index is that the root index is constantly updated as records are added to the data store, whereas partition-level indexes are written all at once when the partition is written to disk. Therefore, we make use of a B-tree data structure at the root index for each flow field, which allows us to efficiently insert and update the bitmaps associated with the field values. Meanwhile, the root index for payload objects is structured as a document index similar to those found in the partitions, except that the bitmaps for each of the field-value pairs point to the appropriate partitions.

The root and partition-level indexes are ideal for finding attributes that appear rarely in the data. Therefore, we may use them to answer counting queries using only the in-memory indexes. The advantage of these types of index-only queries is that they are relatively fast compared to standard queries because they do not have to access data from high-latency disks. One example of such a query might be: '`SELECT Count(*) WHERE Protocol=6 AND DestPort=80`'.

The ability to support index-only queries is important, if for no other reason than it improves the interactive feedback loop that is common in forensic analysis [6, 23]. There, the typical modus operandi is to start with a single query that may return far too many records, and then repeatedly refine the search criteria until an acceptable number of matches are brought forth. Note also that it is possible to extend the current two-level index hierarchy to an arbitrary number of levels, which facilitates indexing of partitions that are hierarchically distributed within a computing cluster, thereby enabling multiple levels of resolution in the index-only queries.

4 Evaluation

In this section, we present an empirical evaluation of our system using two separate network traces. First, we collected approximately 122 GBs of DNS traffic between the UNC DNS servers and external servers during five days in March of 2011. We also collected over 400 GBs of DNS and truncated HTTP traffic over a few hours during a week day in January 2012. The specifics of both datasets are shown in Table 1.

Trace 1: DNS traffic	
Length of Trace	5 days
Average DNS queries per day	66.5 M
Average no. of clients per day	272,945
Original raw trace	122 GB
LZO compressed raw trace	55 GB
Data store (uncompressed)	155 GB
Data store (dict. compression)	83 GB
Data store (dict. + LZO)	75 GB
Trace 2: DNS + HTTP traffic	
Length of Trace	2.5 hrs
Number of Connections	11.1 M
Original raw trace	400 GB
LZO compressed raw trace	358 GB
Data store	12 GB
Number of payload fields	1700+
Number of distinct payload field values	11 M+

Table 1: Data Summary

Our framework was built as a shared-library and was designed to receive events from front-end data collection and sensor products. For the purposes of our evaluation, we integrated our framework with the Bro Intrusion Detection System and collected both DNS and HTTP events. The packets from these events were aggregated offline into flow records with associated payload objects, as described in Section 3. The size of each of our horizontal partitions was set to one million records ($k = 1M$). We chose this value based on empirical results that showed that this choice provided a reasonable balance between large index file sizes and the overhead induced by opening large numbers of files.

All experiments discussed in this section were performed on an Ubuntu Linux server with dual Intel Xeon 2.27 GHz processors, 12 GBs RAM, and a single 2TB 7200 RPM local SATA drive. We chose this configuration because it resembles platforms typically used by researchers and practitioners for network forensic investigation. The results of our experiments were averaged over five runs of each query. Memory and disk caches were cleared between each query.

4.1 Query Types

To better assess the benefits of our approach in quickly retrieving network data, we performed a series of queries that span three categories typically seen in forensic investigations. These queries are similar to those found in previous work focusing on the storage and retrieval of network data [6, 17, 20, 24], and represents the types of queries used by security analysts [23]:

- *Heavy Hitters*: Returns the majority of records in the datastore. Such queries are typically used to gather global statistics about the dataset. This class of queries serves as a good stress test for our approach, and also serves as a point of comparison for sequential scanning techniques. Heavy Hitter queries are also used as a baseline for showing how indexes can affect query times. An example of such a query might be: '`SELECT SourceIP WHERE Protocol = 17 OR Protocol=6`'.

- *Partition Intensive*: Returns records from each partition, but not the majority of the records from those partitions. Analysts might use these types of queries in the early stages of their investigations (e.g., when looking for a specific IP address responsible for a significant amount of traffic or for activity on a common port). Partition Intensive queries are used to show the speedup achieved because of our indexing structures. An example of such a query might be: '`SELECT Dns.Query.Type, Dns.Query.Id WHERE Payload.Dns.Query.Domain = www.facebook.com`'.

- *Needle in a Haystack*: Returns a few records from the datastore. These types of queries might arise in cases where an analyst is searching for a rare event (e.g., traffic to a rogue external host on certain ports). This class of queries demonstrate the effectiveness of hierarchical indexing, as well as the overhead involved with the querying system. An example of such a query might be: '`SELECT SourceIP, SourcePort WHERE Payload.Dns.Query.Domain = www.dangerous.com`'.

4.2 Results

Our first set of experiments were performed on the 122 GB DNS traffic trace. This dataset is interesting because it has a large volume of connections (over 325 million). Since the DNS payload objects store almost all attributes from the DNS packets, the data stored within our system is extremely dense (i.e., large number of payload attributes per flow record), and thus serves as a good test of data retrieval capabilities.

To gain a deeper understanding of the parameters that affect overall query performance, we first vary the number of attributes ($n \in \{1,2,4\}$) returned in the `SELECT` clause while keeping the number of attributes specified in the `WHERE` clause constant. Furthermore, we measure the differences between flow and payload attributes in the queries. Four queries were issued for each of the categories listed in Section 4.1. These queries return flow and payload attributes using the available indexes.

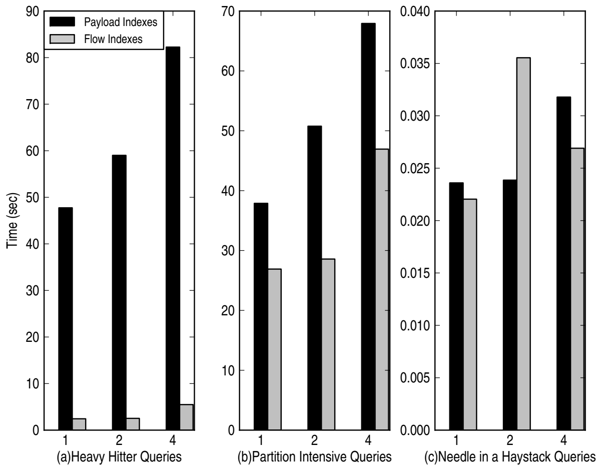


Figure 6: Response times for returning flow attributes filtered using flow indexes (grey) and the payload index (black).

Returning Flow Attributes. Figure 6 (grey bars) shows the performance when returning 1, 2, or 4 flow-based attributes filtered by a flow index. Response times are fast because queries operate on small column files and indexes; therefore, there is no parsing overhead and disk I/O times are reduced. On average, Heavy Hitter query times using flow indexes (6(a)) are lower than Partition Intensive queries (6(b)). While this may seem odd, the results can be explained by the fact that there is significant overhead associated with reading the many attribute indexes used in the `WHERE` clause of the Partition Intensive query, whereas the Heavy Hitter query used only a single, low-cardinality index.

Figure 6 (black bars) shows the query performance when returning 1, 2, or 4 flow-based attributes filtered by the payload index. These types of queries are slower because payload indexes are much larger in size than those for flows because of their cardinality and the variability in length of the indexed values. Even though the queries are slower, notice that even a Heavy Hitter query that returns well over 200M records ($n = 4$ in Figure 6(a)) still completes in roughly one minute. In addition, because our hierarchical indexes efficiently prune irrelevant partitions, Needle in the Haystack queries are extremely fast in all cases — each with sub-second response times.

Returning Payload Attributes. Next, we investigate the query performance when returning payload-based attributes instead of flow-based attributes in the query. Payload-based queries are slower than flow-based queries because payload object files are larger, and such objects have parsing overhead. In order to improve query performance, we applied various compression techniques and compared the resulting query times. Specifically, we compared uncompressed, dictionary compressed, and dictionary+LZO compressed payload versions of our data store. Varying n in our experiments had little impact since the majority of the performance bottleneck can be attributed to loading and parsing the payload objects. Therefore, we only show results for $n = 2$ since the results for $n = 1,4$ are similar.

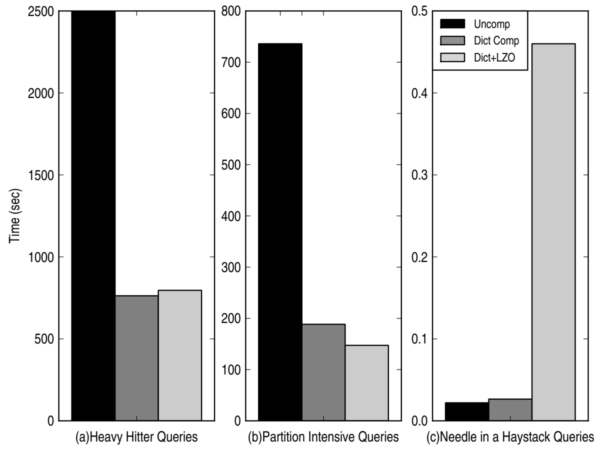


Figure 7: Return payload attributes filtered using a flow-based index (payload queries $n = 2$).

Figures 7 and 8 shows the performance on payload queries for $n = 2$ in experiment. For Heavy Hitter queries on flow indexes (Figure 7), we achieve more than 3 times speedup with dictionary-based and LZO compression enabled, and a 5-fold improvement for Partition Intensive queries. This improvement is a direct result of the reduction in the size of payload files as stored on disk. Needle in a haystack queries, on the other hand, retrieve little data from disk so there is little overhead when operating on uncompressed stores. In fact, the opposite is true: dictionary+LZO compressed queries are slower because the storage manager must decompress the entire file before reading payload records from disk.

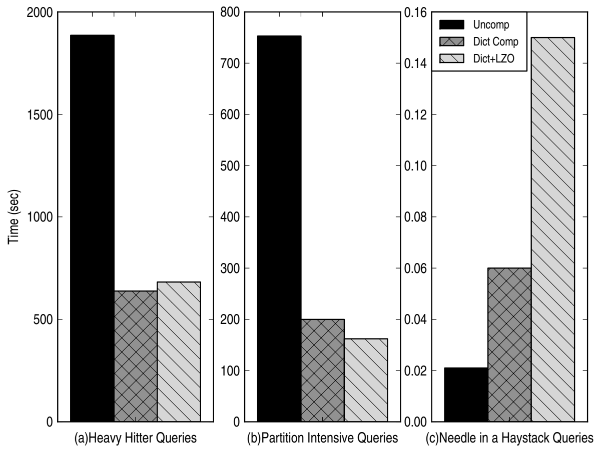


Figure 8: Return payload attributes filtered using a payload-based index (payload queries $n = 2$).

Figure 8 shows the results for payload-based queries using payload indexes. The results suggests that the overhead of processing payload indexes has a small impact compared to the overhead of reading and parsing the payload objects. Heavy hitter, compressed payload queries are nearly 2.8 times as fast as uncompressed payloads, while Partition Intensive queries are over 4.6 times faster.

Overhead Analysis. Next, we consider the increase in size of our data store over time and the component-wise overhead when performing payload queries. Figure 9 shows the growth of various framework components (using dictionary+LZO compression) over the duration of data collection for the DNS dataset. Note that due to space constraints the results are depicted on a log-linear plot. The graph shows that components grow at a very slow linear scale as new data is added. Some components, like the root flow indexes and the dictionary used for string compression, experience incredibly slow growth because of the reuse of field values that naturally occurs in network protocols.

Table 2 presents the average processing time spent in various components for a set of payload-based Heavy Hitter queries using payload indexes. The results show that the majority of time is spent in decoding objects and performing memory management tasks related to creating the result sets to be returned to the client — both of which are areas where optimizations are required to further improve query performance.

Performance Comparison. For comparison purposes, we also examined the performance of traditional relational database systems (e.g., [9, 15]) and toolkits for network forensics [14]. Specifically, we use Postgres version 9.0 and SiLK version 2.4.5. In the case of Postgres, we created four tables: a table for flow attributes, and three tables to hold DNS Answer, Name Server, and Additional resource records, respectively (see Figure 3). The tables are linked to support joins, and indexes were generated on all fields. The resulting data store was almost 5 times as large as the original dataset, and took 8 days to generate. We used a custom application to efficiently query the Postgres database. For the SiLK experiments, we generated the data store using `rwflowpack`, and all records were partitioned hourly following the typical usage of SiLK [25]. The resulting datastore was 7.1GBs. For our queries, the accompanying `rwfilter` utility was used[3].

Component	%
Payload Object Decoding	37.0%
Memory Allocation for Result Set	30.0%
Payload I/O	20.0%
LZO Decompression	6.4%
Miscellaneous	3.4%
Processing Payload Indices	3.2%

Table 2: Processing breakdown of a Heavy Hitter payload query using the payload index.

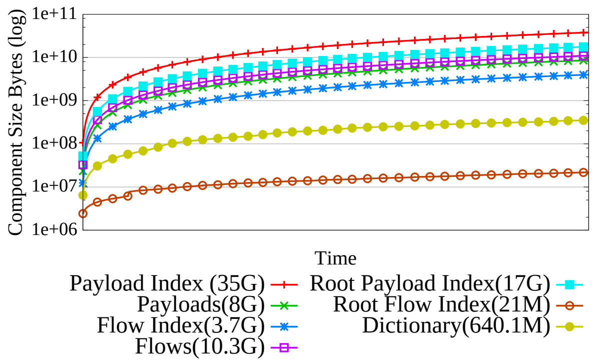


Figure 9: Growth of the components over time for Trace 1.

The results for a series of queries following the categories given in Section 4.1 are provided in Table 3. Notice that the relational database approach consistently performs the worst in all but the Needle in a Haystack queries on flow-based attributes. While still slower than our approach, the use of indexes enable it to avoid scanning all records as is the case with SiLK. SiLK's performance remains constant across all the queries because it uses a sequential scan-and-filter approach whose performance is linear to the size of the data.

Our efforts to compare the performance of payload-based queries was also quite telling. While we had originally hoped to explore Heavy Hitter queries that involved joins between the flow table and the DNS response ta-

[3]For fairness, all output was directed to /dev/null to minimize overhead of console output.

Flow-based Queries	**Heavy Hitters**	**Partition Intensive**	**Needle Haystack**
Postgres	18.1m	9.5m	2.0s
`SiLK`	1.8m	1.8m	1.8m
Our approach	2.5s	30.5s	0.04s
Simple queries, no joins required			
Payload-based Queries	**Heavy Hitters**	**Partition Intensive**	**Needle Haystack**
Postgres	7.6m	9.7m	1.6s
Our approach	9.7m	3.3m	0.1s
Complex queries, joins required			
Payload-based Queries	**Heavy Hitters**	**Partition Intensive**	**Needle Haystack**
Postgres	$> 2h$	$> 2h$	3s
Our approach	30m	2.5m	0.1s

Table 3: Comparison to other approaches.

bles in the Postgres database, the response time was so slow that we terminated most of the queries after two hours (shown in Table 3 as $> 2h$). For a more simplified evaluation, we manually altered the flow table to simply include certain DNS-related fields directly (namely, domain and record type) and then issued queries directly on this altered table to avoid the costly join operations. In this case, the Heavy Hitter payload queries performed similarly in both Postgres and our data store. However, for multi-dimensional Partition Intensive and Needle in a Haystack queries, we again outperform Postgres, returning results in sub-second time in certain cases.

Storing Multiple Object Types. Having established the performance results of the proposed network data store, we now turn our attention to investigating the impact of heterogeneous objects on storage and query performance. In the experiments that follow, we use the DNS+HTTP dataset that contains over 400 GBs of DNS and truncated HTTP traffic. Due to privacy reasons, we were limited to storing only 500 bytes of each HTTP payload and were only allowed to collect that data for a short time period. To explore the impact of heterogeneous data, we use two summary payload objects in addition to the standard network flow store: the original DNS object (as in Figure 3) and an HTTP object which stored the method, URI, and all other available request and response headers as allowed by the truncated HTTP packets. All fields were then indexed and compressed.

Unlike the DNS dataset examined earlier in this section, this dataset contains only 11.1 million connections and a large portion of the traffic contents (i.e., actual web content) are excluded from the payload storage, making the dataset far less dense in terms of stored information. As a result, the 400GB packet trace is converted into a 12GB data store, including indexes and compressed data files. We tested the query performance of the more diverse data store using a select set of queries that mirror the three query classes used earlier. Our Heavy Hitter query returned all 11.1 million records in the data store in 52 seconds on average. The Partition Intensive query returned 6,000 records in 7.6 second on average. Finally, the Needle in a Haystack query returned one HTTP and one DNS record in under 0.4 seconds.

We believe this extended evaluation aptly demonstrates the flexibility and querying power of our approach. In particular, it shows we can store and index arbitrary object schemas, and provide high performance, robust queries across payloads. Furthermore, we are able to customize our summary payload objects and utilize compression techniques to considerably reduce the amount of data stored, while still storing important fields that are useful for forensic analyses.

5 Case Study

As alluded to earlier, post-mortem intrusion analysis has become an important problem for enterprise networks. Indeed, the popular press abounds with reports documenting rampant and unrelenting attacks that have led to major security breaches in the past. Unfortunately, many of these attacks go on for weeks, if not months, before being discovered. The problem, of course, is that the deluge of data traversing our networks, coupled with the lack of mature network forensic platforms, make it difficult to uncover these attacks in a timely manner. In order to further showcase the utility of our framework, we now describe how it was used in practice to identify UNC hosts within the DNS trace that contacted blacklisted domains or IP addresses. To aide with this analysis, we also obtained blacklists from a few sources, including a list of several thousand active malicious domains discovered by the Notos system of Antonakakis et al. [2].

First, for each entry in the blacklist, the root payload index was queried to assess whether these blacklisted domains appeared in the trace. Since the root index is sorted lexicographically by field value and there is no need to touch the partitions themselves, these queries return within milliseconds. The result is a bitmap indicating which partitions the domain appears in. Using these queries, we quickly pruned the list to 287 domains.

Next, we investigate how many records were related to DNS requests for these blacklisted domains by issuing a series of index-only queries (e.g., ‘`SELECT Count(*) WHERE DNS.QueryDomain = www.blacklisted.com`’) to count the number of matching records. That analysis revealed that over 37,000 such requests were made within a one week period. Digging deeper, we were able to quickly pinpoint which internal IPs contacted the blacklisted domains (e.g., ‘`SELECT SourceIP, Time WHERE DNS.QueryDomain = www.blacklisted.com`’). To our surprise, we found at least one blacklisted request in *every* minute of the dataset. More interestingly, roughly

33% of those requests came from a single host attempting to connect to a particular well-known malicious domain name. Correlation with external logs showed that the machine in question was indeed compromised.

Encouraged by the responsiveness of the data store, we turned our attention to looking for additional evidence of compromises within the data. In this case, we issued wildcard queries for domains found among a list of several hundred domains extracted from forensic analysis of malicious PDF documents performed by Snow et al. [27]. Many of these domains represent malicious sites that use domain generation algorithms (DGAs)[4]. To search for the presence of such domains, we simply issued wildcard queries on payload fields. For instance, to find domain names from the `cz.cc` subdomain, which is known to serve malicious content [22, 29], we issued a query of the form: '`SELECT SourceIP, Time, DNS.QueryDomain WHERE DNS.QueryDomain = *.cz.cc`' and discovered 1,277 matches within the trace. While we strongly suspect that many of the connections identified represent traffic from compromised internal hosts, we are unable to confirm that without additional network traces. Nevertheless, all of the aforementioned analyses were conducted in less than fifteen minutes, and yielded valuable insights for the UNC network operators. Without question, the framework was particularly helpful in supporting interactive querying of network data.

6 Summary

Packet payload data contains some of the most valuable information for security analysts, and yet it remains one of the most difficult types of data to efficiently store and query because of its heterogeneity and volume. In this paper, we proposed a first step toward a fast and flexible data store for packet payloads that focuses on well-defined application-layer protocols, with a particular emphasis on DNS and HTTP data. To achieve these goals, we applied a combination of column-oriented data stores, flexible payload serialization, and efficient document-style indexing methods. Our evaluation showed that our approach for storage of payload content was faster and more flexible than existing solutions for offline analysis of network data. Finally, we underscored the utility of our data store by performing an investigation of real-world malware infection events on a campus network.

Overall, our evaluation brought to light several important insights into the problem of large-scale storage and querying of network payload data. The performance of our data store in returning flow attributes, for instance, serves as independent confirmation of the benefits of aggregating packet-level data and using column-oriented approaches to store data with well-defined schemas. Likewise, our results illustrated the power of hierarchical indexing and fixed-size horizontal partitions in both minimizing high-latency disk accesses and enabling the fast index-only queries that are key to interactive data analysis. It is also clear that document-based indexing and flexible object serialization are promising technologies for storing network payloads with highly dynamic data schemas and complex data types. This is particularly true when considering network data with high levels of redundancy, where we can use dictionary-based compression to limit storage overhead and the cardinality of payload indexes. Unfortunately, the document-oriented approach is not without its own pitfalls, since our evaluation also indicated that there are non-trivial amounts of overhead associated with deserializing the payload objects. Moving forward, we hope to build off of the insights from our offline data storage framework to incorporate real-time storage capabilities and to extend the system to distributed computing environment.

[4]See, for example, "*How Criminals Defend Their Rogue Networks*" at `http://www.abuse.ch/?tag=dga`

Acknowledgments

We are especially grateful to Michael Bailey, Douglas Creager, and Srinivas Krishnan for fruitful discussions regarding this work. Thanks to Hank Levy, Kevin Snow and the anonymous reviewers for their insightful comments and suggestions for improving an earlier draft of this paper. We also thank Murray Anderegg and Bil Hayes for their help in setting up the networking infrastructure that supported some of our data collection efforts. This work is supported in part by the National Science Foundation under award 1127361 and the Natural Sciences and Engineering Research Council of Canada (NSERC) Postgraduate Scholarship Program.

References

[1] 10gen Inc. MongoDB. Available at `http://www.mongodb.org/`.

[2] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster. Building a Dynamic Reputation System for DNS. In *USENIX Security Symposium*, 2010.

[3] Apache Software Foundation. Apache CouchDB. See `http://couchdb.apache.org/`, 2008.

[4] Apache Software Foundation. Apache Avro. `http://avro.apache.org/`, 2011.

[5] Apache Software Foundation. Apache Hadoop. See `http://hadoop.apache.org/`, 2011.

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

[7] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD International Conference on Management of Data*, pages 283–296, 2009.

[8] E. Cooke, A. Myrick, D. Rusek, and F. Jahanian. Resource-aware multi-format network security data storage. In *SIGCOMM Workshop on Large-scale Attack Defense*, pages 177–184, 2006.

[9] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *ACM SIGMOD International Conference on Management of Data*, pages 647–651, 2003.

[10] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW Client-based Traces. Technical report, Boston University, 1995.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, 2008.

[12] L. Deri, V. Lorenzetti, and S. Mortimer. Collection and exploration of large data monitoring sets using bitmap databases. In *TMA*, pages 73–86, 2010.

[13] F. Fusco, M. Vlachos, and M. Stoecklin. Real-time creation of bitmap indexes on streaming network data. *The VLDB Journal*, pages 1–21, 2011.

[14] C. Gates, M. Collins, M. Duggan, A. Kompanek, and M. Thomas. More netflow tools: For performance and security. In *18th Conference on Systems Administration*, pages 121–132, 2004.

[15] R. Geambasu, T. Bragin, J. Jung, and M. Balazinska. On-demand view materialization and indexing for network forensic analysis. In *USENIX International Workshop on Networking Meets Databases*, pages 4:1–4:7, 2007.

[16] P. Giura and N. Memon. NetStore: An Efficient Storage Infrastructure for Network Forensics and Monitoring. In *Intl. Conf. on Recent Advances in Intrusion Detection*, pages 277–296, 2010.

[17] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider. Enriching network security analysis with time travel. In *ACM SIGCOMM Conference on Data Communication*, pages 183–194, 2008.

[18] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *Intl. Conf. on Very Large Databases*, 2010.

[19] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD international conference on Management of data*, pages 165–178, 2009.

[20] T. Plagemann, V. Goebel, A. Bergamini, G. Tolu, G. Urvoy-keller, and E. W. Biersack. Using data stream management systems for traffic analysis - a case study. In *Passive and Active Measurements*, pages 215–226, 2004.

[21] M. Ponec, P. Giura, J. Wein, and H. Brönnimann. New Payload Attribution Methods for Network Forensic Investigations. *ACM Transactions on Information Systems Security*, 13:1–32, 2010.

[22] N. Provos. Top 10 malware sites. Available at `http://googleonlinesecurity.blogspot.com/2009/06/top-10-malware-sites.html`, 2009.

[23] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, and J. M. Hellerstein. Efficient analysis of live and historical streaming data and its application to cybersecurity. LBNL Technical Report 61080, 2006.

[24] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, and J. M. Hellerstein. Enabling Real-Time Querying of Live and Historical Stream Data. In *International Conference on Scientific and Statistical Database Management*, page 28, 2007.

[25] T. Shimeall, S. Faber, M. DeShon, and A. Kompanek. *Analysts' Handbook: Using SiLK for Network Traffic Analysis*. CERT Network Situational Awareness Group, 2010.

[26] R. R. Sinha and M. Winslett. Multi-resolution Bitmap Indexes for Scientific Data. *ACM Transactions on Database Systems*, 32, August 2007.

[27] K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos. Shellos: enabling fast detection and forensic analysis of code injection attacks. In *20th USENIX conference on Security*, pages 9–9, 2011.

[28] R. Sommer and V. Paxson. Enhancing Byte-level Network Intrusion Detection Signatures with Context. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 262–271, 2003.

[29] Sucuri Research. Malware from .cz.cc domains. Available at `http://sucuri.net/malware-from-cz-cc-domains.html`, 2011.

[30] J. Vijayan. Oak Ridge National Lab shuts down Internet, email after cyberattack. Available at `http://www.computerworld.com/`, 2011.

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

[32] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. on Information Theory*, 24(5):530 – 536, 1978.