Selective Capping of Packet Payloads for Network Analysis and Management

Víctor Uceda, Miguel Rodríguez, Javier Ramos, José Luis García-Dorado, Javier Aracil

Selective capping of packet payloads for network analysis and management

Víctor Uceda, Miguel Rodríguez, Javier Ramos, José Luis García-Dorado and Javier Aracil

High Performance Computing and Networking,
Universidad Autónoma de Madrid, Spain
{vic.uceda,miguel.rodriguez01}@estudiante.uam.es,
{javier.ramos,jl.garcia,javier.aracil}@uam.es

Abstract. Both network managers and analysts appreciate the importance of network traces as a mechanism to understand traffic behavior, detect anomalies and evaluate performance in a forensic manner, among other applications. Unfortunately, the process of network capture and storage has become a challenge given the ever-increasing network speeds. In this scenario, we intend to make packets thinner to reduce both write speed and storage requirements on hard-drives and further reduce computational burden of packet analysis. To this end, we propose to remove the payload on those packets that hardly could be interpreted afterwards. Essentially, binary packets from unknown protocols fall into this category. On the other hand, binary packets from well-known protocols and protocols with some ASCII data are fully captured as potentially a network analyst may desire to inspect them. We have named this approach as selective capping, which has been implemented and integrated in a high-speed network driver as an attempt to make its operation faster and more transparent to upper layers. Its results are promising as it achieves multi-Gb/s rates in different scenarios, which could be further improved exploiting novel low-level hardware-software tunings to meet the fastest networks' rates.

1 Introduction

Traffic monitoring at multi-Gb/s rates poses significant challenges not only for traffic capturing but also for traffic storage and processing. On the one hand, traffic capture at high-speed requires either ad-hoc network drivers that incorporate sophisticated prefetching, core affinity and memory mapping techniques [3] or specifically tailored network interface cards based on network processor devices or FPGA [1]. On the other hand, once the packets have been captured, they must be swiftly transferred to hard disk at the same pace that they are received from the network.

A most important issue in traffic dumping to hard disk is packet size, the larger the packet the more prone the traffic losses and the higher the storage investments. To circumvent this issue, the packet payload is capped to a predefined *snaplen* size. The aim is to reduce the offered write throughput to the hard

disk, which is the hard disk performance figure of merit and bottleneck. Needless to say, a large share of packets contain a payload that is useless for subsequent analysis, for example encrypted payload packets. Precisely, the motivation of this research is to drop useless payload packets, by selectively capping their length to the mere packet header. The benefit of this approach is threefold: first, the hard disk bottleneck is alleviated, since the write speed requirements decrease. Second, we reduce the storage space, which is very large for a high speed network, even if the capture duration is small. Third, the computational burden required to analyze capped packets is lower, for example the RAM memory requirements of NIDS applications, such as Snort, are less stringent. We call our data thinning technique selective capping. In more detail, we propose marking a packet payload of interest according to two conditions.

First, if it is binary from a well-known protocol –i.e., if it can be interpreted by a traffic dissector. To do so, one has to identify such network services beforehand and only apply selective capping on such services, while leaving the rest untouched –e.g., using flow director filters [5] on port numbers or IP address ranges.

A packet is also interesting if it contains human readable data (more formally, ASCII, UTF-8, UTF-EBCDIC or equivalent) –which can be interpreted by a network analyst or by the application designer. The human readable, ASCII in what follows, case entails a harder work as it embraces not only all-ASCII protocols –e.g, SIP–, but protocols with both binary and ASCII interleaved parts. This is the case of HTTP for example, in which binary content (pictures, videos, etc.) is interleaved with ASCII text. Such binary content is not useful for the most performance analysis such as web profiling or HTTP server response times. Our findings show that for typical HTTP traffic up to 60% of the traffic is made up by binary content. Consequently, if the binary content could be removed on-the-fly then the hard disk input rate would be largely reduced. Similarly to HTTP there is a number of popular protocols that merge binary and ASCII which span all Internet activities. Banking networks leverage protocols such as FIX. Furthermore, routing and login protocols such as Radius and IS-IS, monitoring-oriented protocols like IPFIX and database management systems such as TNS are examples of this behavior. In the case of encrypted protocols such as HTTPS, all packet payloads are useless which motivates even more to discard them. However, it is worth remarking that encryption is not strongly present in enterprise scenarios, where monitoring is performed inside the local network and traffic is unencrypted.

Detection of ASCII packets should be as simple as inspecting the whole payload and checking if every single byte belongs to a given ASCII alphabet. However such alphabets typically encompass close to half possible byte values. As an example, ASCII encodes 128 specified characters into 7-bit binary integers which makes a random byte to fall into the alphabet range one out of two times. In this light, the following two mechanisms are proposed. The first one seeks for a set of consecutive ASCII characters on the payload, namely a run, as an approximation to the idea of a word in the natural language. The second one is

based on the percentage of bytes candidates to be classified as ASCII. We have elaborated on these two mechanisms and we present a formal description of the false positive (FP) rate of both. Specifically, we show how to parameterize them to achieve a given error/FP target.

The contributions of this paper go beyond the proposal and evaluation of the selective capping algorithm. The novel algorithm is integrated in a high-speed network driver and the results show multi-Gb/s network-to-hard disk sustainable throughput, whereas the state of the art analysis reveals that no selective capping algorithms have been proposed to date but session-level or flow-record approaches. Those approaches require constructing sessions or flows which is in itself a challenging task on multi-Gb/s scenarios [1].

The rest of this section is devoted to the thorough definition of the problem and the related work on data thinning techniques for traffic captures. Section 2 provides a description of the two proposed methods, while Section 3 presents the architecture and implementation details of a traffic sniffer equipped with selective capping. Section 4 presents the results and discussion, both in terms of compression and speed. Finally, Section 5 outlines the conclusions that can be drawn from this paper and the future work lines.

1.1 Problem statement

We define a packet payload to be of interest if:

- It is entirely-binary from a well-known service.
- It contains ASCII data –note that we are using the term ASCII as a synonym of human-readable data regardless its codification.

The rationale behind these two conditions is that the payload of a packet (over transport layer) is of interest only if it can be interpreted afterwards. In other words, network analysts may only found of interest those payloads that can be interpreted and eventually turn out useful in their tasks of understanding traffic behavior and its dynamics, detect anomalies or evaluate performance issues among others. Going back to the banking network example, one may be interested in reading error messages from certain transactions, which may be written in plain ASCII in the application-level payload.

The first condition is met by applying network and transport layers filters over traffic. The second one requires identifying ASCII traffic. After inspecting a diverse set of traces that includes academic networks and private networks from banks and large enterprises [10], we have found that the Internet carries ASCII data in diverse ways as so are protocols, services and scenarios on the Internet. Figure 1 shows the taxonomy of the findings. The figure depicts as black the ASCII part of each packet of a given class of protocols while the white part represents binary bytes. Let us elaborate in each of these classes:

- *Class I*: It represents purely binary protocols. RTP and encrypted protocols such as SSL or SSH are examples.

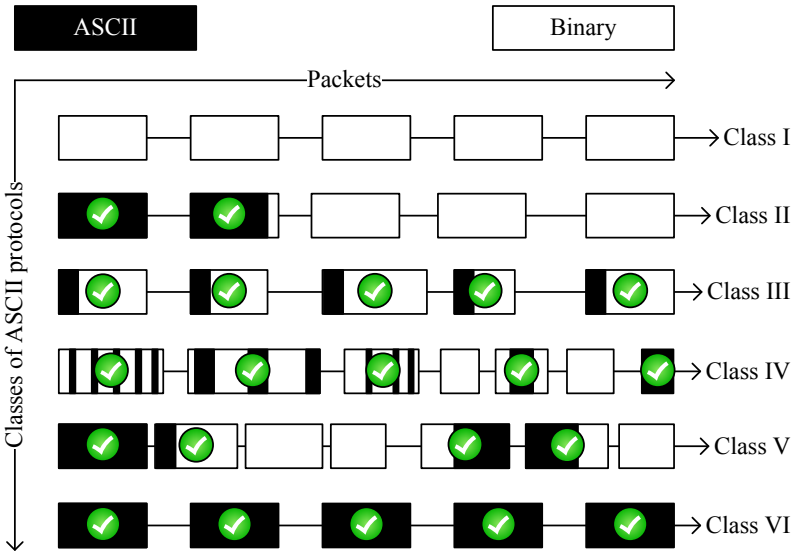

Fig. 1. Taxonomy of classes of protocols according to how ASCII data is carried.

- *Class II*: Some protocols exchange ASCII data at the beginning of a connection, typically signaling, and after some sort of content is sent. This content is usually binary such as pictures, documents, or any other object. Often protocols in this class are grouped into the term flow oriented. Non-persistent HTTP is an example of this. Given its importance on Internet aggregates, the amount of bytes that can be saved by cutting the binary part of the connections is promising.
- *Class III*: As third class, we classify protocols where each packet carries an ASCII header along with binary content. They are often referred as packet oriented, Universal Plug-and-Play protocol (UPnP) is a significant example.
- *Class IV*: We define as the fourth class protocols that interleave ASCII and binary content with short runs. In this way, typically most of the packets have a binary and ASCII part and only a small fraction would be either all-ASCII or all-binary. DNS and Skinny Call Control Protocol (SCCP) are examples of this. Especially representative of this pattern are TLV-based (Type-Length-Value) protocols. In these protocols, typically, a binary code states the meaning of the subsequent values, often, ASCII values. Other significant protocols such as Lightweight Directory Access Protocol (LDAP), SNMP, Remote Authentication Dial-In User Service (RADIUS), IS-IS and H.323 among others follow this functionality. In this case, the problem cannot be addresses by keeping only the ASCII bytes of a given packet, but also it is necessary to capture the surrounding binary values which give meaning to the ASCII data. To do so we decided to mark the entire packet of interest as a faster approach to rule out several parts of a packet. In addition, once a piece of TLV behavior is found, it is likely that there will be more occurrences.

– *Class V*: Similar to the previous class but more frequent, we have found protocols that interleave both binary and ASCII data but with long runs of each of them. The most significant example is HTTP-persistent given its contribution on Internet aggregates volumes. According to the measurements in [12] about 60% of all HTTP requests are persistent and they represent more than 30% of the total transferred volume over HTTP. In addition, there are a number of both management and banking protocols that follow this pattern. Such protocols are of paramount importance for network analysts on bank networks as some of them are close protocols and reverse engineering is required to carry any study on them. An important example of them is Oracle's TNS (Transparent Network Substrate) used by databases' transfers which encompass a request –which typically includes ASCII data– and a bulk data transfer for requested objects –a file, a list of records, for example. Other examples are proprietary bank transfers' accounting protocols or communication protocols such as Link Layer Discovery Protocol (LLDP) among others.
– *Class VI*: Finally, this class states for all-ASCII protocols, for example SIP.

As conclusion, the problem we are facing is on the one hand to deploy hardware filters to forward well-known service to hard-drive; and on the other hand, in the rest of the traffic to detect those packets that include ASCII data bearing in mind the diversity of ways ASCII data is carried.

1.2 State of the art

How to reduce the amount of stored traffic while keeping its most significant pieces of information has received notable attention by the research community given the importance of traffic traces on monitoring tasks [6, 7, 11, 13]. The common factor of these novel works is that they first capture traffic and construct its respective flows. Then they decide what packets or fraction of payloads to rule out or how to apply compressing mechanisms on the headers and payloads of a given flow.

More specifically, the authors in [7] developed a system, named Time Machine, that rules out the last packets of the flows –i.e., packets beyond a arbitrarily-fixed threshold. The rationale behind this proposal is that such packets are often less discriminant for monitoring purposes (e.g., the signaling tends to be at the beginning of communications). This, together with the heavy-tail nature of the Internet flow sizes whereby a small fraction of flows account for most of the traffic, makes that by fixing a maximum flow size of 15 KB the required capacity translates into less than 10% of the original size, while keeping records for most of the flows. Afterwards, the authors in [6] extended the set of possible thresholds to the maximum number of bytes per packet and packets per flow with similar purposes and motivation.

An alternative approach to the problem is to compress packet headers or data. In this sense, the authors in [2] exploited the particularities of network traffic

to overcome the compressing capacity of standard tools over traffic headers – such as zip or rar. Specifically, traffic follows a very specific format where some fields appear in the same position and with similarity (e.g., in a capture IP addresses tend to share a prefix and appear in the same positions). Similarly, the authors in [13] leveraged dictionary-based mechanism to reduce workload on both HTTP and DNS traffic. Strings found in such protocols are hash-mapped to numbers and replaced in the capture traces previously indexed in a database. This makes that the trace cannot be accessed directly by typical packet-oriented libraries such as libpcap nor libpcap-like flavors [3]. This is not necessarily an inconvenient in terms of both accessibility and storage capacity, but becomes a challenge in performance terms.

Precisely in this regard, the authors in [11] focused its work on high-speed networks, that is, multi-Gb/s networks. They exploit the modern NICs' capacity to configure hardware filters on-the-fly. Their proposal, Scap, constructs flows similarly to [7] but once the maximum flow size is exceeded, a NIC filter is raised to rule out subsequent packets of the corresponding flow. This makes packets that were going to be rule out at application layer be ruled out before in the network stack thus saving resources. As a result, Scap is able to deal with traffic at ranges between 2.2 and 5.5 Gb/s depending on traffic patterns and configuration. On the downside, note that discarding packets at the lowest level is often an inconvenient as such discarded packets (or at least their headers) may be of interest for monitoring purposes. Additionally, real-time filter reconfiguration becomes a challenge as setting a hardware filter takes 55 μs [5] while the inter-arrival packet can be as small as 68 ns in 10 GbE networks.

We propose to make the decision if the payload of a given packet can be potentially of interest and consequently, entirely captured, as a first step to any other tasks, which are normally resource-intensive. In this way, instead of capping the number of packets or its payload up to an arbitrarily threshold, the byte number reduction is attained by storing only those bytes that have a chance to be analyzed in the future. As mentioned before, we have termed such processed as selective capping and its explained throughout this paper.

With the aim of applying selective capping as soon as possible in the network stack and, importantly, in a transparent way to users, we have entrusted the network driver layer with this task. This implies a carefully low-level hardware-software interaction, but on the upside, a possible way to achieve multi-Gb/s rates in real traffic traces. Additionally note that by modifying driver level, we ensure that all the above-introduced stream-oriented mechanisms still remain valid. That is, after our thinning process, upper-layer proposals can be further applied for additional storage capacity cuts.

2 Detection of ASCII traffic

Prior to ASCII traffic detection, we must realize ASCII standard and other equivalent text representations span a large fraction of the total 256 possible values for a byte. ASCII codification serves well as a generalization for human-

readable data, as UTF-8 is the most used encoding over the Internet, and it uses ASCII representation for Latin characters. In particular, in ASCII alphabet such a fraction accounts for about 40%. Consequently, there are significant chances a random byte falls into the ASCII range regardless it represents an ASCII character or not.

In this scenario, we make two observations that we have translated into two different methods. Essentially, ASCII characters tend to be consecutive one another as they often represent words in natural language. We have named the method based on this observation as ASCII-Runs threshold. On the other hand, we note that is very unlikely that a large set of ASCII characters fall by chance in a randomly-payload packet. In this way, we parameterize the possibility of such a random packet containing more than a given fraction of ASCII bytes by chance. We refer this method as to ASCII-Percentage threshold.

Let us detail these two methods, explain how to parameterize them and provide a final proposal combination of both.

2.1 ASCII-Runs threshold

In most of the text-based protocols the ASCII characters represent words –often keywords in English such as GET or POST in HTTP case. In a genuine ASCII packet, ASCII will not be randomly distributed, but there will be runs –i.e., ASCII characters located consecutively. Based on this observation, we propose a first method to seek runs of ASCII bytes in packet payload. If at least one significant run is found, we mark such packet as ASCII and otherwise as binary. We formally define a significant run as such a run whose length ensures a parameterized maximum error with respect to a randomly payload distribution in a packet. In other words, this error is the False Positive (FP) ratio, i.e., the probability of a random payload packet to be marked as ASCII when it is binary.

Let us consider a binary packet formed by random bytes. If the byte values are uniformly distributed through 0 to 255, the probability of a byte corresponding to an ASCII character will be around 0.4 (p). Then, error can be modeled using a Markov chain with L+1 states, each of it corresponding to have read an ASCII run of L consecutive characters, being then the last state absorbent. The stochastic matrix for this Markov chain is:

$$M_L = \begin{pmatrix} 1-p & p & 0 & 0 & \cdots & 0 \\ 1-p & 0 & p & 0 & \cdots & 0 \\ 1-p & 0 & 0 & p & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1-p & 0 & 0 & 0 & \cdots & p \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

The i-th row (being zero-row the top one) represents the state where i ASCII characters have been read consecutively. The probability of finding such a run of length L can be computed by observing this Markov chain evolution in n steps, being n the length of the packet. The probability of finding an ASCII

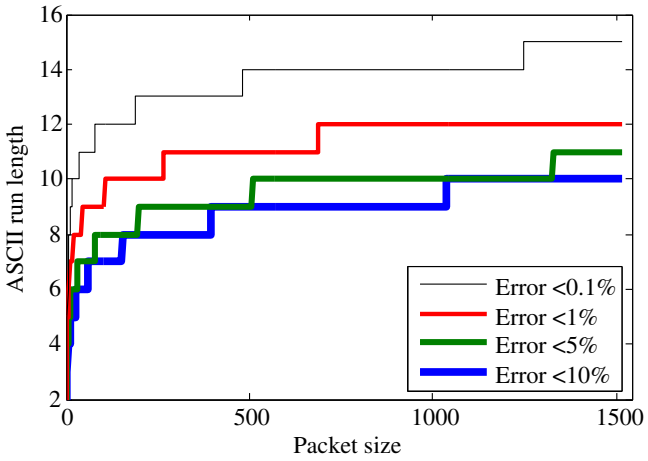

Fig. 2. Required run lengths for several error thresholds

run of at least length L is then located on the upper right corner of $(M_L)^n$. By knowing this probability we can state the minimum ASCII run required to have a pre-parameterized error probability –in the FP sense, likely a low error probability.

Figure 2 shows the required run length for different error threshold and packet sizes. As an example, a false positive ratio of one packet out of 1000, assuming 1000-byte packets entails that a run of 14 consecutive characters that fall into ASCII alphabet range should be found.

2.2 ASCII-Percentage threshold

The ASCII-Runs threshold method fits with most of the classes showed in the taxonomy section. However, in TLV protocols chances are that the value of a given field is below the required run length. Moreover, it is unlikely that a random payload packet contains many characters falling into ASCII range. Thus, this method works out ASCII characters percentages, and then decides to mark a packet as ASCII contrasting such a percentage with a significant threshold. To calculate this significant threshold, let us consider a packet of length n as a sequence of bytes each with probability p of being 1 (which would correspond to an ASCII character) and probability 1-p of being 0 (some binary value). With this information we can compute the error rate according to the packet length.

Such an error follows a binomial distribution with parameters n and p. Its CDF $F(x)$ represents the probability of having less than x ASCII characters on a packet of length n. By studying the quantiles of this distribution, we work out the minimum ASCII percentage required to guarantee an error threshold on classifying packets. Such percentages depend on the packet length. Figure 3 shows the required ASCII percentages for different packet sizes depending on the chosen error threshold.

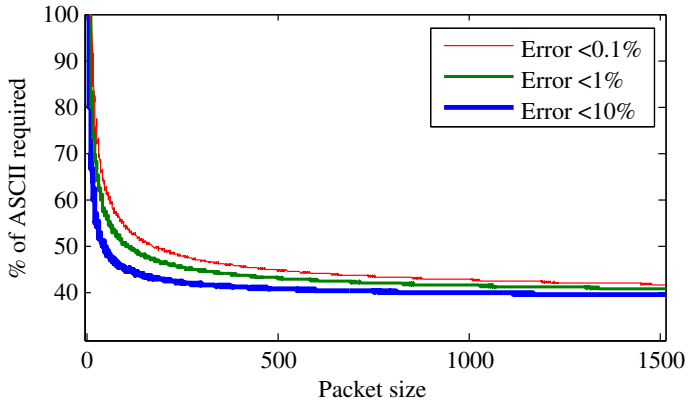

Fig. 3. Required ASCII percentages for several error thresholds

2.3 Multiple thresholds

Both proposed methods are complementary and they are tailored to different ways ASCII data is distributed over packet payloads. Therefore, we propose to apply both of them and mark a packet as ASCII if any of the methods mark the packet as ASCII. The error threshold in this combination will be at least smaller than the largest one –e.g., if the error thresholds are 0.1 and 1%, respectively, the effective error will be strictly smaller than 1%. This motivates to apply them with the same error threshold –e.g., a low figure of 0.1%. As both are simple and based on similar principles –to seek bytes falling in a given range–, executing both concurrently does not may cause extra overhead on the system, being the slowest which sets the pace.

Note that in our approach, False Negative (FN) cases span those packets that being semantically ASCII have not met our decision thresholds. Such a semantic approach should be based on an in-depth and empirically characterization of run lengths and percentage occurrence of ASCII data in each protocol of each described class –Section 1.1.

3 Selective capping sniffer architecture

The selective capping sniffer follows a two-step architecture. The first step consists on splitting incoming traffic into two categories by means of hardware filters such as Intel Flow Director [5]. Traffic is divided into well-known (in the protocol/port sense) binary traffic that network managers do want to keep, and other traffic over which managers want to cap to transport header or keep in its own if it is ASCII related. Each of these traffic sets is redirected to a different RSS (Receive-Side Scaling) driver queue. The second step is to analyze at driver level the capping candidate traffic and mark packets accordingly. Finally, both capped and uncapped packets are delivered to user level to perform the traffic storage. Figure 4 illustrates the architecture of the proposed system.

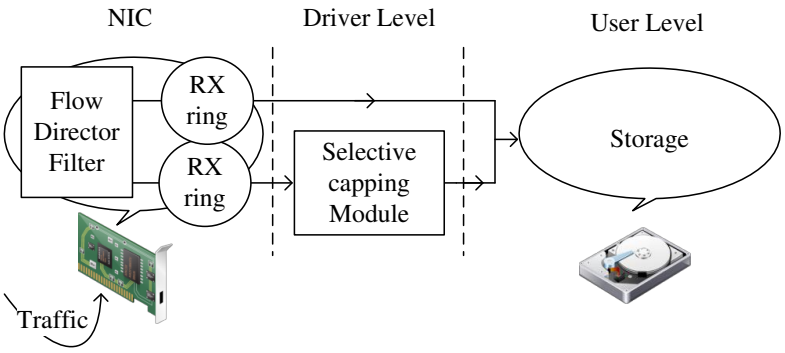

Fig. 4. Selective capping traffic sniffer architecture

Our implementation leverages HPCAP [8] as store engine. Over this base, we have added as a driver level module our implementation of the two capping methods explained previously. Note that neither flow analysis nor DPI can be performed due to time restrictions at this stage, and only simple per-packet operations can be carried out. On the upside, this would provide user level with a transparent thinning process in addition to save the resources that capped bytes do not use along network stack. Furthermore, this also allows for a higher capture throughput.

Actually, this implementation must be extremely efficient in order to cope with multi-Gb/s rates and beyond. Algorithm 1 presents it in pseudocode.

First, both ASCII runs threshold and ASCII percentage threshold are calculated offline using the models described in the previous section to provide a given error (e.g., 0.1%). After, the algorithm traverses all the payload of each packet checking if each byte falls into the printable ASCII value range.

When a run of ASCII characters of length equal to the run threshold is found, the packet is marked for full-content storage. Similarly, if the packet contains a percentage of ASCII characters which is equal to or larger than the percentage threshold, the packet is marked for full-content storage.

Finally, if none of the conditions are met the packet is capped to its transport header length, and only such data is preserved for subsequent analysis. Importantly in terms of performance, to carry out the capping to header, our implementation exploits the advanced packet-descriptor features that modern Intel's NICs offer. Such descriptors provide the protocol stack and protocol header lengths coded in NIC hardware.

After this processing, packets are delivered to user level. Then, an application stores the captured traffic along with a PCAP-like header on a high-performance store solution for subsequent access thereof (e.g., [8]). Alternatively, traffic can be first forwarded to an on-the-fly analysis system [9] or a general-purpose compression system –e.g., gzip, which is specially effective over ASCII content.

Algorithm 1 Selective Capping Algorithm

```
runASCII=0
totalASCII=0
for all bytes in payload do
  if MIN_PRINTABLE_ASCII<= byte_value <=MAX_PRINTABLE_ASCII then
    runASCII++
    totalASCII+=100
    if runASCII >= ASCII_RUNS_THRESHOLD then
      Do not cap packet and process next one
    end if
  else
    runASCII=0
  end if
end for
if totalASCII >= ASCII_PERCENTAGE_THRESHOLD * packet_length then
  Do not cap packet and process next one
end if
Cap packet and process next one
```

Table 1. Evaluation traces summary

Trace	Avg packet size (bytes)	Number of packets	Info.
1	164	1000000	DNS 34%, HTTP 21%, SSH 20% Dropbox 15%, Others 10%
2	786	1000000	HTTP 100%
3	927	4718531	HTTP 43%, HTTPS 13%, SSH 25% Banking protocol 17%, DNS 2%

4 Results and discussions

We have evaluated both performance and compression ratio using three different traces. To test the performance, the traces have been replayed at different speeds above the original rate in order to explore the limits of our proposal. To test the compression rate, the traces have been replayed at the original rate and the number of capped bytes and packets have been counted. For evaluation purposes, the used flow-director filter redirects all traffic to the selective capping module.

Table 1 shows the most relevant information for each used trace. Trace 1 has been captured in an academic link and contains HTTP, DNS, SSH and Dropbox traffic. Trace 2 contains only HTTP traffic captured from the enterprise network of an important insurance company. Trace 3 has been captured in a large commercial bank network and contains HTTP, HTTPS, SSH, proprietary banking protocols and DNS.

The reception evaluation has been performed on an server with 128 GB of DDR4 memory and two 6-core Intel Xeon processors running at 2.30 GHz. The server motherboard model is Supermicro X9DR3-F. Additionally, to perform the packet reception, a 10 Gb/s NIC based on an Intel 82599 chip has been used.

Table 2. Compression ratio

Trace	Compression ratio	% of capped packets
1	4.28	45
2	3.33	74
3	3.24	81

Table 3. Average capping throughput

Trace	Avg. throughput (Gb/s) ± Std. Dev.	Avg. packet rate (Kpps) ± Std. Dev.
1	3.1 ± 0.13	2221 ± 100
2	2.5 ± 0.08	856 ± 29
3	3.2 ± 0.15	428 ± 21

This NIC is connected using a PCIe 3.0 slot. On the sender side, traffic has been replayed using another Intel 82599 card and a custom software traffic generator based on PacketShader [4] API. Such generator is able to replay PCAP traces at variable rates.

4.1 Compression Ratio

First the compression level of our proposal is evaluated. To this end, the aforementioned traces have been replayed at original rates while capturing and storing into disk. After the replay is complete, the original and captured traces are compared in order to obtain the compression ratio. Table 2 shows the compression ratio values in terms of stored bytes and the amount of capped packets. As it can be observed in Trace 1, almost every packet is capped providing a compression ratio of 4.28. In the case of Trace 2, only 74% of the packets are capped obtaining a compression ratio of 3.33. Finally, in Trace 3 81% of the packets have been capped obtaining a compression ratio of 3.24.

It is worth remarking that although not all the traffic in Trace 3 is HTTP, a large amount of packets have been capped including bank protocols which provides a good idea of the applicability of the capping techniques in both commercial and academic networks where heterogeneous traffic is found.

4.2 Performance Evaluation

Once the compression ratio has been assessed, the aforementioned traces have been replayed at different rates until lossless packet capture and disk storage is achieved, in order to test the performance of the proposed method. The duration of each experiment is 30 minutes. Table 3 shows the average packet capture throughput including our capping methodology for each 30-minute experiment. For the sake of completeness also observed standard deviation is reported.

As shown, our proposal achieves multi-gigabit capture rates. Depending on the payload and packet-rate of each trace different capture rates are achieved.

For example, if a trace contains a large amount of binary packets, the workload of the system is greater as each single byte of the binary packet has to be checked before packet is capped.

On the other hand if a trace contains a large amount of text packets, only a small byte-run must be checked before packet is fully captured. An example of this case is Trace 1 where 45% of the packets are text packets. Trace 2 contains a large amount of binary packets and presents a significant average packet rate which results in a reduced performance. Despite the large amount of binary packets present in Trace 3, the average packet rate is smaller than Trace 2 which results in better performance as fewer packets per second must be checked.

Finally, regarding memory consumption, our solution makes use of a static 1GB kernel packet buffer to receive and analyze incoming traffic.

5 Conclusions and Future Work

We have presented a solution for selective packet capping –on-the-fly– to reduce the amount of stored data in multi-Gb/s networks. Our proposal focuses on keeping the payload of those packets that are worth interpreting by network managers and analysts. As a consequence our proposal stores both well-known (in the protocol/port sense) binary protocols and ASCII protocols. The latter has received most of our attention given the difficulties to address it in light of the protocol-diverse and high-speed nature of current business applications. To this end, two methods for ASCII packet identification have been implemented at driver level by modifying a novel high-performance capture engine. The implementation of selective capping allows us to provide a clean and transparent mechanism to cap packets without user-level interaction/tuning. Performance and compression ratio have been assessed using both academic and commercial traffic obtaining compression ratios between 3 and 4. On the other hand, the performance results achieve remarkable multi-Gb/s rates –ranging from 2.5 Gb/s to 3.2 Gb/s– which do not suffice for the fastest network interfaces rates –10, 40 and even 100 Gb/s. Nonetheless, our solution has proven to cope with a real-world OC-192 link such as the one described in [14].

Therefore, as future work we first plan to attack the capping problem in full-loaded 10 GbE links. To this end, we are studying the use of parallelism paradigms, likely at user level and hardware solutions such as NetFPGAs or GPUs. Similarly, we are studying how to reduce the burden of looking for ASCII data. For example, instead of inspecting full packet payload, we could limit the inspection to one or several randomly-chosen windows.

Moreover, we have realized that packet payloads sometimes contain constant values for long runs. Such runs do not provide network analysts with any interesting piece of information and should be capped. As a result, we are measuring the dispersion of the values of bytes in payloads as an attempt to find a formal threshold below which packet payloads render useless. Similary, we are studying if some legibility indicators –e.g., vowels, punctuation or entropy– could be

useful to separate semantically-worth ASCII from the total traffic classified as ASCII.

Finally, throughout this paper we have focused on ASCII standard as an illustrative example of codification scheme. We are currently extending our work to other popular schemes such as Base64 and full UTF-8.

References

1. Forconesi, M., Sutter, G., López-Buedo, S., López de Vergara, J.E., Aracil, J.: Bridging the gap between hardware and software open-source network developments. IEEE Network 28(5), 13–19 (2014)
2. Fusco, F., Vlachos, M., Dimitropoulos, X.: Rasterzip: Compressing streaming network monitoring data with support for partial decompression. In: ACM Internet Measurement Conference. pp. 51–64 (2012)
3. García-Dorado, J.L., Mata, F., Ramos, J., Santiago del Río, P.M., Moreno, V., Aracil, J.: High-performance network traffic processing systems using commodity hardware. In: Data Traffic Monitoring and Analysis, chap. 1, pp. 3–27 (2013)
4. Han, S., Jang, K., Park, K.S., Moon, S.: PacketShader: a GPU-accelerated software router. In: ACM SIGCOMM. pp. 195–206 (2010)
5. Intel: 82599 10 Gbe controller datasheet (2012), `http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html`, [1 December 2014]
6. Lin, Y.D., Lin, P.C., Cheng, T.H., Chen, I.W., Lai, Y.C.: Low-storage capture and loss recovery selective replay of real flows. IEEE Communications Magazine 50(4), 114–121 (2012)
7. Maier, G., Sommer, R., Dreger, H., Feldmann, A., Paxson, V., Schneider, F.: Enriching network security analysis with time travel. In: ACM SIGCOMM. pp. 183–194 (2008)
8. Moreno, V., Santiago del Río, P.M., Ramos, J., García-Dorado, J.L., Gonzalez, I., Gómez-Arribas, F.J., Aracil, J.: Packet storage at multi-gigabit rates using off-the-shelf systems. In: IEEE Conference on High Performance and Communications. pp. 486–489 (2014)
9. Moreno, V., Santiago del Río, P.M., Ramos, J., Muelas, D., García-Dorado, J.L., Gómez-Arribas, F.J., Aracil, J.: Multi-granular, multi-purpose and multi-Gb/s monitoring on off-the-shelf systems. International Journal of Network Management 24(4), 221–234 (2014)
10. naudit: Detect-pro (2013), `http://www.naudit.es/`, [1 December 2014]
11. Papadogiannakis, A., Polychronakis, M., Markatos, E.P.: Scap: Stream-oriented network traffic capture and analysis for high-speed networks. In: ACM Internet Measurement Conference. pp. 113–124 (2012)
12. Schneider, F., Ager, B., Maier, G., Feldmann, A., Uhlig, S.: Pitfalls in HTTP traffic measurements and analysis. In: Passive and Active Measurement. pp. 242–251 (2012)
13. Taylor, T., Coull, S.E., Monrose, F., McHugh, J.: Toward efficient querying of compressed network payloads. In: USENIX Annual Technical Conference. pp. 113–124 (2012)
14. Walsworth, C., Aben, E., Claffy, k., Andersen, D.: The CAIDA anonymized 2009 Internet traces, `http://www.caida.org/data/passive/passive_2009_dataset.xml`, [1 December 2014]