Selective capping of packet payloads at multi-Gb/s rates

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

Department of Electronics and Communications Technology, Universidad Autónoma de Madrid, Madrid, Spain

Abstract

Network managers and analysts are well aware of the importance of network traces to understand traffic behavior, detect anomalies, and evaluate performance forensically, among others. However, the storage required for traffic traces has greatly expanded due to increasing network speeds. In this paper, we selectively cap the packet payload to reduce write speed and storage requirements on hard drives and further reduce the computational burden of packet analysis. The proposed techniques take advantage of most packet payloads being useless for analysis purposes, because they are either encrypted or in a proprietary application non-readable format. Conversely, non-ASCII packets from well-known protocols and protocols with some ASCII data are fully captured as they may be potentially useful for network analysis. We have named this approach as selective capping, and we have implemented and integrated it into a high speed network driver and a software module at user level, to make its operation more transparent and faster to upper-layer applications. The results are promising, selective capping achieves multi-Gb/s rates by exploiting low level hardware and software techniques to meet the fastest network rates.

Keywords: Packet Storage, Traffic capture, Selective Capping, Interleaved ASCII.

1 Introduction

Both network managers and analysts appreciate the importance of network traces as a mechanism to understand traffic behavior, detect and reproduce anomalies, as well as test performance forensically, among other applications. However, traffic capture at multi-Gb/s speeds poses significant challenges, and so does the corresponding data storage. Traffic capture at high speed requires either ad-hoc network drivers that incorporate sophisticated prefetching, core affinity, and memory mapping techniques [1] or specifically tailored network interface cards based on network processor devices or Field Programmable Gate Arrays (FPGAs) [2]. For FPGAs, packets must be swiftly transferred to hard disk at the same pace that they are received from the network, or packet losses will eventually happen, even though they are captured at the Network Interface Card (NIC).

Hard disks are sensitive to packet size for traffic dumping, the larger the packet the more likely are traffic losses and the higher the required storage investment. The aim is to reduce the write throughput to the hard disk, which is the hard disk performance figure of merit and bottleneck. Many studies have investigated this issue based on sampling [3], omitting packets fulfilling certain characteristics (e.g. the last packets of flows [4], non-failure packets [5], multimedia [6]), removing redundant traffic [7], data summarization [8] or, very often, capping packet payloads to a predefined snaplen size [9]. These various approaches are not necessarily mutually exclusive.

Regarding payload capping, many packets contain a payload that is useless for subsequent analysis, e.g. encrypted or proprietary application non-readable packets. In other words, network analysts may only find of interest those payloads that can be interpreted and provide information relevant to understanding traffic behavior and its dynamics, detect anomalies, evaluate performance issues, etc.

Since hard disk performance decreases with packet size, we propose a data thinning technique, *selective capping*, where useless payload packets are dropped by selectively capping their length to the packet header. The benefit of this approach is threefold:

1. The hard disk bottleneck is alleviated, since the write speed requirements decrease.

2. Storage space is reduced, which is very large for a high speed network, even if the capture duration is small.

3. The computational burden required to analyze capped packets is lowered, e.g. the RAM requirements of Network Intrusion Detection System (NIDS) applications, such as Snort, are less stringent.

We propose to mark a packet payload of interest, and record the whole packet, according to two conditions. The first one, the packet payload is from a well-known service, i.e., 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. Second, the payload contains human-readable data, i.e., ASCII, UTF-8, UTF-EBCDIC, or equivalent, which can be interpreted by a network analyst or application designer. For example, in a banking network, one may be interested in the error messages from certain transactions, which may be written in plain ASCII in the application level payload.

We remark that the human-readable case entails a harder workload as it embraces not only pure ASCII protocols, e.g. Session Initiation Protocol (SIP), but also protocols with interleaved binary (i.e., any value between 0 and 255) and ASCII parts. Indeed, there are a number of popular internet protocols that merge non-ASCII and ASCII formats. This is the case for HTTP for example, where binary content (e.g. pictures, videos, etc.) is interleaved with ASCII text. The binary part is not useful for most performance analysis, such as web profiling or HTTP server response times. Up to 60% of typical HTTP traffic is composed of binary content. Consequently, if the binary content were removed, the hard disk input rate would be greatly reduced. Similarly to HTTP, other protocols interleave binary and ASCII formats, e.g. the Financial Information eXchange (FIX) protocol is used for trading in banking networks and the data payload is amenable for further analysis, such as response times between query and reply. Routing and login protocols such as Remote Authentication Dial-in User Service (RADIUS) and IS-IS, monitoring oriented protocols such as IP Flow Information Export (IPFIX), and database management systems such as Transparent Network Substrate (TNS) are further examples of this behavior. In the case of encrypted protocols, such as HTTPS, all packet payloads are useless for network analysis, and may be discarded. However, encryption is not strongly present in enterprise scenarios, where monitoring may be performed after the IPSEC tunnel and traffic is captured unencrypted.

To sum up, we deem a packet payload of interest if:

1. it is from a well-known protocol, or

2. it contains ASCII data (where we use ASCII as a synonym of human-readable data, regardless its codification).

To address the first task, we propose to use flow director filters [10] on port numbers or IP address ranges. The use of flow director allows packets with a given destination/origin port range to be directed to separate queues in the NIC, which in turn can be treated differently at the user level.

Regarding the second point, detection of ASCII packets appears to be as simple as inspecting the payload and checking if every byte belongs to a given ASCII alphabet. However, collision with random binary bytes occur, because such alphabets typically encompass close to half the possible byte values, e.g. ASCII encodes 128 specified characters into 7 bit binary integers, so a random byte will fall within the alphabet range one out of two times. Therefore, some context information must be considered. We propose the following mechanisms:

1. Search for a set of consecutive ASCII characters on the payload, i.e., a run, which occurs for words in the natural language.

2. Identify the percentage of byte candidates to be classified as ASCII.

To conclude, our proposal is, on the one hand to deploy hardware filters to forward well-known services to the hard drive, and on the other hand to detect those packets in the rest of the traffic that include ASCII data, bearing in mind the diversity of ways ASCII data is carried.

The paper is organized as follows. First, Section 2 elaborates on previous data thinning techniques for traffic captures. Next, in Section 3, we present a taxonomy based on how the Internet carries ASCII data and describes the two different mechanisms to detect ASCII packets presenting a formal description of the false positive (FP) rate of both. Section 4 presents how such mechanisms translate into a vanilla selective capping algorithm, which is after optimized and both accuracy and cost figures are provided. Section 5 presents the architecture and implementation details of a traffic sniffer equipped with selective capping at driver and user levels. In Section 6, we evaluate the proposed algorithms in terms of processing and compression rate. They provide sustainable multi-Gb/s network to hard disk throughput. Finally, Section 7 outlines our conclusions and relevant results, and discusses possible future developments.

2 Related work

Reducing the amount of stored traffic (often referred as to data thinning) while keeping its most significant pieces of information has received considerable attention from the research community. The first approaches were typically based on transforming traffic into another thinner representation, as in the popular Netflow and subsequent optimizations [11], smart summarizations [8] or, simple time-series [12].

Given the importance of traffic traces for monitoring tasks, other works focus on mechanisms to keep the packet semantic intact [13, 4, 14, 15, 16]. Most have the common approach to first capture the traffic and then construct flows. Subsequently, they decide what packets or fraction of payloads to exclude and apply compression mechanisms to the headers and payloads for the (remaining) set of packets comprising each flow.

More specifically, the authors in [13, 4] developed a system, Time Machine, which excludes the last packets of a flow, i.e., packets beyond an arbitrary threshold. The rationale is that such packets are less informative for monitoring purposes, i.e., the relevant signal tends to be at the beginning of communications. Together with the heavy tail distribution of internet flow, where a small fraction of flows account for most of the traffic, fixing a maximum recorded flow size of 15 kB reduces the required capacity to less than 10% of the original size, while keeping adequate records for most of the flows. Subsequently, the authors in [14] extended the set of possible thresholds to the maximum number of bytes per packet and packets per flow with similar purposes and motivation.

Alternate approaches have been based on compressing packet headers or data. In this sense, the authors in [17] exploited the particularities of network traffic to overcome the compressing capacity of standard tools over traffic headers, such as zip or rar. Traffic follows a very specific format where some fields appear in the same position and with similarity, e.g. within a capture, IP addresses tend to share a prefix and appear in the same positions. Similarly, the authors in [16] employed a dictionary based mechanism to reduce workload for HTTP and DNS traffic. Strings found in such protocols were hash mapped to numbers previously indexed in a database, and replaced in the capture traces. However, this meant the trace could not be accessed by well-known packet oriented libraries, such as libpcap or libpcap-like [18]. This is not necessarily an inconvenience in terms of accessibility and storage capacity, but becomes a challenge for usability.

Precisely in this regard, the authors in [15] focused on high speed, i.e., multi-Gb/s, networks, and exploited the NIC's capacity to configure hardware filters on the fly. Their proposal, Scap, constructs flows similar to [4], but once maximum flow size is exceeded, an NIC filter is applied to remove subsequent packets from the corresponding flow. Packets that would have been removed at the application layer are removed in the network stack, which saves resources. Thus, Scap is able to deal with traffic rates up to 2.2–5 Gb/s, depending on traffic patterns and configuration. However, discarding packets at the lowest level is often inconvenient as such discarded packets, or at least their headers, may be of interest for monitoring purposes. Furthermore, real time filter reconfiguration becomes a time challenge, since setting the NIC filter takes 55 μs [10], while the inter-arrival times of packets can be as small as 68 ns in 10 GbE networks.

Other approaches focus on thinning the traffic traces after they are stored. For example, in [5] traffic is thinned only after ensuring that it contains the same failure events as the original trace. This approach allows testing for anomalous patterns.

In contrast to previous approaches, we propose to decide whether the payload of a given packet is potentially of interest, and, therefore, entirely captured, as the first step, prior to other, resource intensive, tasks. Indeed, other approaches, such as those reviewed above, may be subsequently applied for additional storage capacity reduction if desired. We call this process selective capping as rather than capping the number of packets or payload size to an arbitrary threshold, the reduction is attained by storing only those bytes that could usefully be analyzed in the future.

3 Detection of ASCII traffic

The ASCII standard and other equivalent text representations span a large fraction of the 256 possible values for a byte. ASCII codification serves well as a generalization for human-readable data, as UTF-8 is the most common encoding over the Internet, and it uses the ASCII representation for Latin characters, which accounts for approximately 40% of the possible byte configurations (ignoring non-printable characters). Consequently, there is a significant likelihood a byte falls into the ASCII range regardless of whether it represents an ASCII character or not. In this light, we need to find more complex patterns to consider a packet as ASCII. Given the diversity of ways ASCII data is carried, we first present a taxonomy of how ASCII bytes are distributed in internet traffic. Then, we translate the observations derived from the taxonomy into two different ASCII detection schemes.

3.1 Taxonomy

After inspecting a diverse set of traces including academic and private networks from banks and large enterprises [19], the Internet carries ASCII data with a variety of protocols, services, and scenarios, as summarized in Fig. 1. The ASCII part of each packet of a given class of protocols is marked in black, the white part represents binary bytes.

The classes shown in Fig 1 are:

- *Class I*: Purely binary protocols, e.g. Real-time Transport Protocol (RTP) and encrypted protocols such as Secure Sockets Layer (SSL) or Secure Shell (SSH).
- *Class II*: Protocols that exchange ASCII data at the beginning of a connection, typically signaling, with some sort of content that is sent subsequently. This content is usually binary, e.g. pictures, documents, etc. Typically, protocols in this class are grouped into the term flow oriented, e.g. non-persistent HTTP. Given its importance for internet traffic, the amount of bytes that can be saved by dropping the non-ASCII part of such connections is promising.

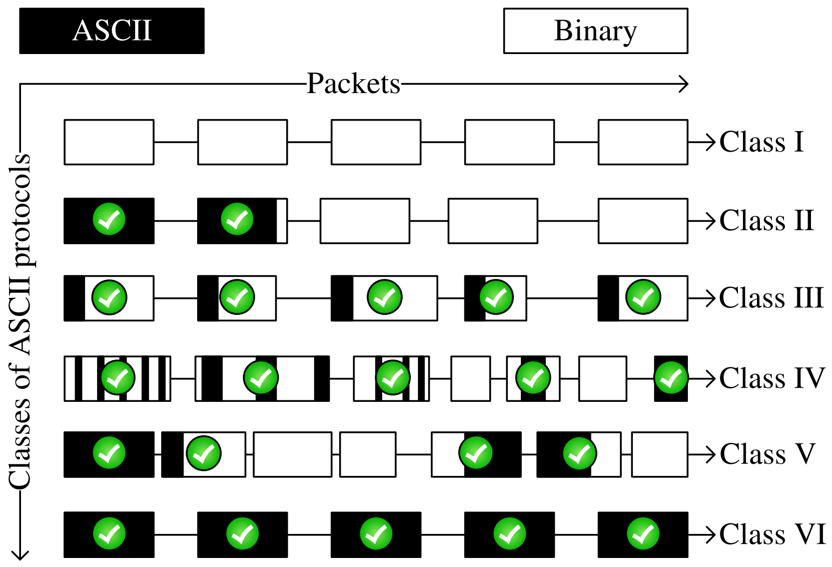


Figure 1: Taxonomy of classes of protocols according to how ASCII data is carried

- *Class III*: Protocols where each packet carries an ASCII header along with binary content. These are often referred as packet oriented, e.g. the Universal Plug and Play (UPnP) protocol.
- *Class IV*: Protocols that interleave ASCII and binary content with short runs having only a small fraction of pure ASCII or non-ASCII, e.g. Domain Name System (DNS) and Skinny Call Control Protocol (SCCP). In particular, Type Length Value (TLV) protocols are in this class. In TLV, a binary code provides the meaning of the subsequent, often ASCII, values. Other significant protocols, such as Lightweight Directory Access Protocol (LDAP), Simple Network Management Protocol (SNMP), RADIUS, IS-IS and H.323, etc., follow this functionality. In this case, the problem cannot be addressed by keeping only the ASCII bytes of a given packet, the surrounding non-ASCII values must also be captured to provide meaning to the ASCII data. Hence, the entire packet is of interest as it contains legible data. In addition, once TLV behavior is found, it is likely that there will be more such occurrences.
- *Class V*: Protocols that interleave both binary and ASCII data with long runs of each. The most significant example is persistent HTTP, given its contribution to internet traffic volumes. From [20], approximately 60% of all HTTP requests are persistent and represent more than 30% of the total volume transferred over HTTP. A number of management and banking protocols also follow this pattern. Such protocols are of paramount importance for network analysts on banking networks, as some are proprietary protocols and reverse engineering is required to study them. An important example of this class is Oracle's TNS, used for database transfers that encompass a request, which typically include ASCII data, and bulk data transfer for requested objects, e.g. a file, list of records, etc. Other examples are proprietary bank transfer accounting protocols or communication protocols such as Link Layer Discovery Protocol (LLDP) among others.
- *Class VI*: Pure ASCII protocols, e.g. SIP.

3.2 Detection Schemes

Two observations arise:

1. ASCII characters tend to be consecutive, as they often represent words in natural language. We refer to this proposed detection method as the ASCII-runs detection scheme.
2. It is very unlikely that a large set of ASCII characters falls by chance in a random packet payload. Thus, we may parameterize the possibility of a random packet containing more than a given fraction of ASCII bytes by chance. We refer to this method as the ASCII-percentage detection scheme.

which provide the following selective capping schemes.

3.2.1 ASCII-Runs detection scheme

In most text based protocols, ASCII characters represent words, often keywords in English such as GET or POST in the HTTP case. In such protocols, ASCII will not be randomly distributed, rather there will be runs, i.e., consecutive ASCII characters. Consequently, we propose a method to seek runs of ASCII bytes in a packet payload. If at least one significant run is found, we mark the packet as ASCII, and otherwise as non-ASCII.

We define a significant run as being when its length achieves as much as a parameterized FP rate with respect to a random payload distribution in a packet (i.e., bytes are independent and uniformly distributed random variables). Here, we define the FP rate, often referred as to type I error, as the probability of a random packet payload to be erroneously marked as ASCII when it is non-ASCII (as a run of arbitrary length appeared by chance). The following calculations are performed to determine the required ASCII run length threshold to achieve a certain FP value.

Let us consider a packet formed by random bytes. On the other hand, let us consider that a packet is tagged as ASCII, in accordance to the ASCII-runs detection scheme if, say, an ASCII-run of length L at least is found within the packet. In order to derive the FP rate versus the value of L, we wish to estimate the probability of finding a run of at least L ASCII characters by chance. If the byte values are uniformly distributed 0–255, the probability, p, of a byte corresponding to an ASCII character will be approximately 0.4. Let us denote by $\mathcal{X} = \{X_i \in \{0, \ldots, L\}, i \in \{0, \ldots, N\}\}$ the discrete-time finite Markov chain that represents the number of consecutive ASCII characters in a run of L bytes, being N the length of the packet under examination, and $N \geq L$. The chain serves to measure the event that at least L consecutive ASCII characters are observed in the packet, whereby all states are transient except the last state L which is absorbent. As it turns out, the event that at least L consecutive ASCII characters are found within the packet is equivalent to the event that the chain $\mathcal{X}$ eventually visits state L regardless of its future evolution. Consequently, 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 (the zero-row is the top row) represents the state where i ASCII characters have been consecutively read. The probability of finding such a run of length L can be computed by observing the Markov chain evolution over N steps, where N is the length of the packet. The probability of finding an ASCII run of at least length L is then located in the upper right corner of $(M_L)^N$.

From this probability, we can state the minimum ASCII run required to have a predetermined FP rate. Figure 2 shows the required run length for different FP values and packet sizes. As an example, an FP rate of one packet out of 100, assuming 1000-byte packets, requires an ASCII run of 12 consecutive characters.

3.2.2 ASCII-Percentage detection scheme

The ASCII-runs detection scheme method is suitable for most of the classes discussed in Section 3.1 but for Class IV protocols (e.g. TLV) it is likely the value field length is below the required run length. However, it is unlikely that a random payload packet contains more bytes falling into ASCII range than those into non-ASCII counterpart, when ASCII range accounts for only, approximately, 40% of the possible values. Thus, we propose a method that calculates ASCII character percentages, and marks a packet as of interest using a percentage threshold.

Similarly to the previous case, to estimate the percentage threshold value given a certain FP rate, 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 non-ASCII value). Thus, we can use a binomial distribution with parameters N and p to compute the FP rate. The cumulative distribution function (CDF), $F(x)$, represents the probability of having less than x ASCII characters in a packet of length N. The minimum ASCII percentage to guarantee a predetermined FP rate for classifying packets depends on the packet length. Figure 3 shows the required ASCII percentages for different packet sizes, depending on the FP rate.

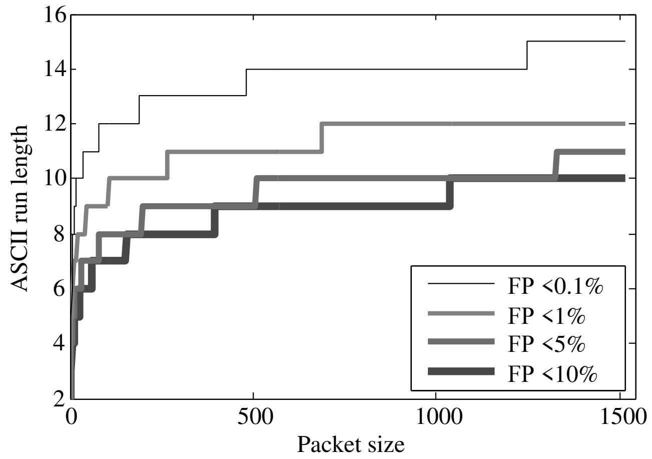


Figure 2: ASCII run lengths for several predetermined FP rates

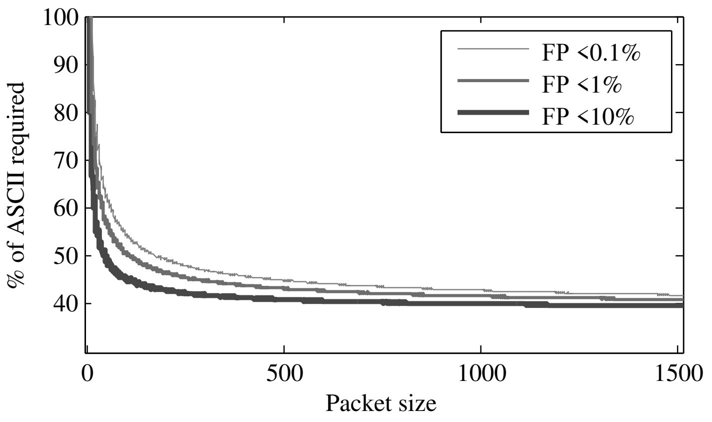


Figure 3: Required ASCII percentages for several predetermined FP rates

3.2.3 Multiple detection schemes

The two methods are complementary and are tailored to different ways ASCII data is distributed over packet payloads. Therefore, we propose to apply both methods, and mark a packet as ASCII if either one marks the packet as ASCII. The FP rate in this combination will be at least smaller than the largest, e.g. if FP rates are 0.1 and 1%, respectively, the effective FP rate will be less than 1%. Thus, there is good reason to apply the methods with the same FP rate, e.g. 1%. Since the methods are simple and based on similar principles, seeking in the ASCII range, executing both concurrently does not cause extra overhead on the system, and the slowest method sets the overall pace.

It is worth remarking that false negative (FN) cases can potentially span packets that, although semantically ASCII, do not meet our decision thresholds, i.e., are deemed useless for traffic analysis, even though they have ASCII characters. A semantic approach refers to the extraction of meaning from the content of a packet in the same way a human does. For example, consider a 1500-byte packet that only contains the word "ERROR" as ASCII content while the rest of the bytes are non-ASCII. In such a case, the proposed detection methods will not mark the packet as ASCII. However, a human will not only detect, visually, the ASCII characters but also extract a meaning from such characters making the packet potentially of interest. In this way, as a further step in our proposal, each below-threshold ASCII run could be looked up in an ad-hoc dictionary of interesting words (e.g. including protocol keywords and typical error messages, possibly in several languages).

4 Selective capping algorithm

We transform the detection methods described in Section 3 into an ASCII detection algorithm, presented as pseudocode in Algorithm 1. Let us refer to such an algorithm as vanilla algorithm. The algorithm traverses the payload of each packet checking if each byte is within the ASCII value range. Whenever a run of ASCII characters of length equal to the ASCII-runs threshold is found, the packet is marked for full content storage. Simultaneously, if the packet contains a percentage of ASCII characters larger than or equal to the ASCII-percentage threshold, the packet is also marked for full content storage. If neither condition is met, the packet is capped to its transport header length, and only that header is preserved for subsequent analysis.

The ASCII-runs and ASCII-percentage thresholds are calculated offline using the models described in the Section 3, targeting a specific FP rate (e.g. 1%). For such FP rate, either 60% ASCII bytes in a packet, or an ASCII run of length 12 or more characters are required to mark a packet as ASCII. In what follows, such figures will be used for illustrative purposes.

Applying Algorithm 1 is expensive in terms of time and computational power, as the packet is traversed byte by byte and ends only after a suitable ASCII run is detected or once the whole packet has been read. Inspecting every single byte of a packet is a demanding task. The maximum achievable speed for Algorithm 1, executing in RAM, with random packet payloads is slightly lower than 10 Gb/s [21]. To achieve higher rates, we tuned the vanilla algorithm to avoid the inspection of most of the bytes of each packet, accessing bytes located on positions divisible by a specific run length (L). This improves the algorithm performance without missing ASCII runs, i.e., if the i-th byte and the $i + L$th byte are both non-ASCII, then there is no L-byte ASCII run between them.

Algorithm 1 Vanilla 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
```

In practical terms, if an ASCII character is found after an L-byte jump, the payload is backwardly inspected searching for an L-byte long ASCII run. If a non-ASCII character is found, the algorithm continues from that position skipping the next L bytes.

The algorithm is illustrated in Fig. 4 for the previously selected run length ($L = 12$). In Packet 1, the algorithm starts at byte 12 and the payload is backwardly inspected searching for an ASCII run. In this case, a run is found between bytes 1 and 12 and the algorithm finishes marking the packet as ASCII. In Packet 2, a non-ASCII character is found at position 4 which causes the algorithm to jump 12 bytes ahead, i.e., to byte 16, then restarting the process. Eventually the packet is marked as ASCII after the run located between bytes 15 and 26 is found. Finally, Packet 3 does not contain any 12-byte length ASCII run which eventually triggers the ASCII-percentage detection scheme.

We note that to avoid the bias induced from analyzing nearby bytes instead of uniformly distributed bytes in such scheme, only the last byte of each L-byte jump are considered after no ASCII runs were found. As an example, in Packet 3, only bytes 12, 24, 35 and 47 would be considered for the ASCII-percentage detection scheme.

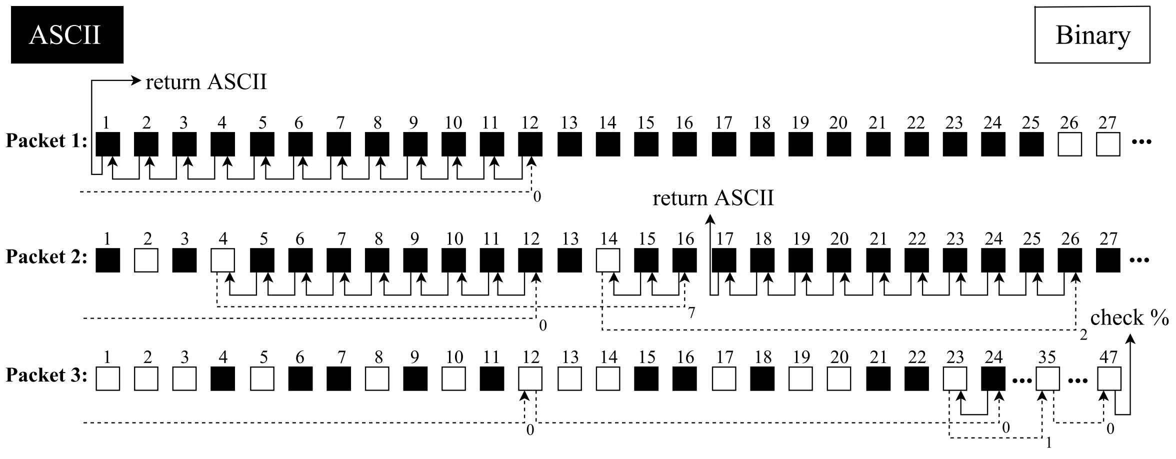


Figure 4: Optimized algorithm examples for packets with different distributions of ASCII bytes ($L = 12$)

4.1 Accuracy

Although the proposed optimization ensures detecting an existing L-character long ASCII run within the packet, different results may be obtained when applying the ASCII-percentage scheme. In the analytical case, we measure the FP rate. In particular, assuming independence between bytes in the payload, and considering that the probability of being ASCII is the same ($p \approx 0.4$), the problem

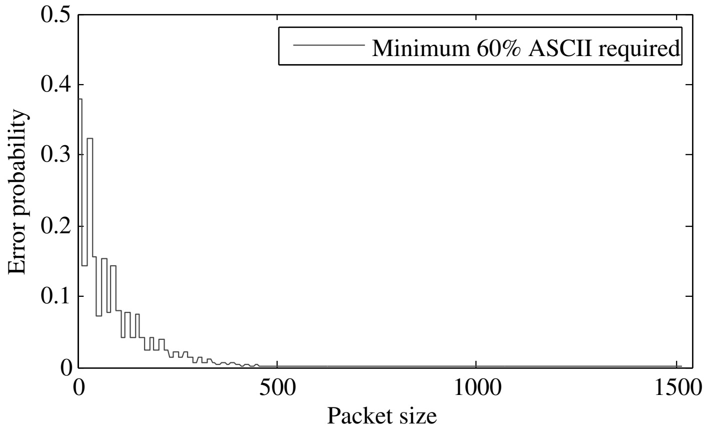


Figure 5: Error probability for a non-ASCII packet using the optimized algorithm ($L = 12$)

follows a binomial distribution as stated before. However, in this optimized case, the distribution parameters are $\lceil N/L \rceil$ and p, being N the length of the packet, L the run length and $f(x) = \lceil x \rceil$ the ceiling function.

The FP rate for a non-ASCII packet as a function of its length is shown in Fig. 5 for $L = 12$ and a minimum percentage of ASCII of 60%. Given that the minimum percentage of ASCII characters comes from a division, and the ceiling function is applied to the result, oscillations are present depending on the divisibility of N, L and the ASCII percentage threshold. For large packets, the FP rate is low. On the other hand, in applying the optimized algorithm increases the FP rates when analyzing small non-ASCII packets. According to this, small packets should be preferably processed with the vanilla algorithm.

Empirically, the error induced by the optimization can be illustrated by comparing the execution of both algorithms (vanilla and optimized). We identify the FPs, non-ASCII packets marked as ASCII by the optimized algorithm, and FNs, ASCII packets marked as non-ASCII by the optimized algorithm. FPs are not a significant problem, it will just lead to useless traffic being stored, increasing the storage space required. However, FNs mean that packets that should be stored are no longer accessible, resulting in loss of potentially relevant content.

As the FP and FN rates may vary significantly depending on the type of traffic and protocols analyzed, we conducted several experiments with real traffic traces that include a widespread group of the protocols introduced in Section 3.1. These are summarized in Table 1, and were captured from an international bank and insurance company (HTTP proxy) network. Note that some protocols, such as DNS, are not intended to be processed with the proposed algorithms, as they are known to be always relevant for subsequent analysis and are forwarded using NIC hardware (HW) filters. However, for the sake of completeness, these protocols were analyzed as representative examples.

Figure 6 shows the FP and FN rates from the traffic, where packets with less than 36 bytes of payload were processed using the vanilla algorithm as a successful tradeoff between accuracy, as shown, and performance, as will be shown. Indeed, rates are very low, particularly FNs, which are the most critical. As the axes are logarithmic and zero cannot be represented, FN rates of $2 \cdot 10^{-7}$ have been plotted for SAS, SSH, SMB, SSL, and SNA, where the calculated error rate was exactly zero. Other protocols not shown in the figure have both FP and FN rates equal to zero.

4.2 Cost Analysis

Let us now analyze the cost of both vanilla and optimized algorithms in terms of number of key comparisons, namely, the number of inspected bytes the algorithms need to make the decision, ASCII or non-ASCII, for a packet.

4.2.1 Vanilla Algorithm

To find the cost of the vanilla algorithm, we propose another Markov chain. While the one studied in Section 3.2 calculated the probability of finding an L-byte run throughout a packet, now we are interested in the probability of finding an L-byte run after inspecting i bytes being the run located at

Table 1: Protocols and traces used

Trace	**Protocol**	**Number of packets**	**Avg. packet size (Bytes)**
1	RTP	220,190	214.00
2	SIP	139,116	297.56
3	HTTP	5,850,632	383.51
4	DNS	6,058,087	175.82
5	LDAP	5,826,266	227.17
6	Netflow	1,077,883	453.09
7	Radius	173,536	349.20
8	SAS	1,968,157	1,026.70
9	SAMBA	9,292,127	175.79
10	SNA	10,000,000	138.04
11	UPnP	44,647	483.95
12	SSL	44,263	573.67
13	TNS	2,992,060	195.79
14	P2P	3,287,599	449.32
15	SMTP	386,075	436.99
16	SSH	203,308	1,134.80

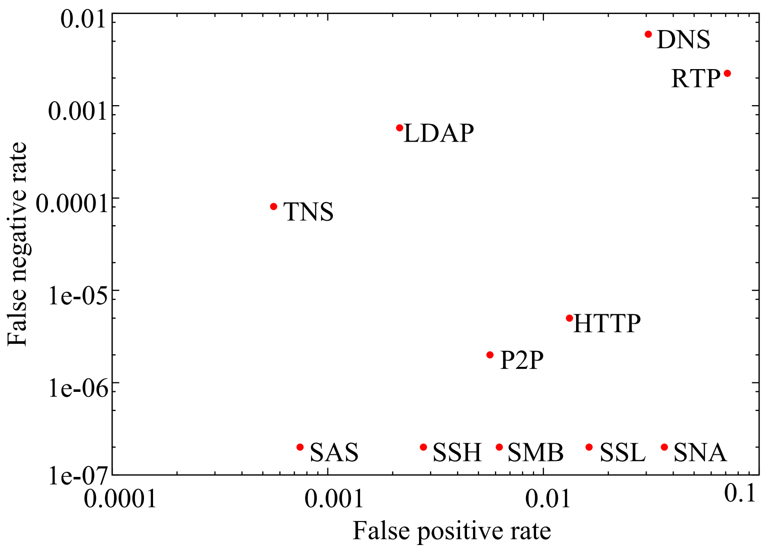


Figure 6: FP and FN rates for optimized algorithm

the last L positions. In this way, the average cost comes from the addition of the number of inspected bytes multiplied by such probability.

We separate the probability of the absorbent state in the previous Markov chain into detecting a run in less than i bytes and exactly after i bytes inspected. To do so, we add a new transient state (L) which corresponds to having detected an L-byte long run in the last L bytes of i bytes inspected. Hence, the absorbent state ($L+1$) in this case represents the probability of having found an L-byte run after inspecting less than i bytes.

Formally, we denote by $\mathcal{Y} = \{Y_i \in \{0, \ldots, L+1\}, i \in \{0, \ldots, N\}\}$ the discrete-time finite Markov chain that represents the number of consecutive ASCII characters in a run of L bytes with the exception of state $L+1$, which represents that an L-byte run was already found. Where N is the length of the packet under examination and $N \geq L$.

As it turns out, the event that exactly L consecutive ASCII characters are found just after i inspections is equivalent to the event that the chain $\mathcal{Y}$ visits state L, and visiting state $L+1$ is equivalent to have already found an L-byte run. Consequently, the stochastic matrix for this Markov chain is:

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

where:

- ${M^i}_{(1,k)}$ $0 \leq k < L$ is the probability of being at the state k after i inspections.
- ${M^i}_{(1,L)}$ is the probability of finding an L-byte run after exactly i inspections.
- ${M^i}_{(1,L+1)}$ is the probability of having found an L-byte run previously (i.e., with less than i inspections).

This gives the following equation for the cost, $C_v(N,p,L)$, for the vanilla algorithm:

$$C_v(N,p,L) = N \cdot (1 - {M^{N+1}}_{(1,L+1)}) + \sum_{i=L}^{N} i \cdot {M^i}_{(1,L)} \tag{1}$$

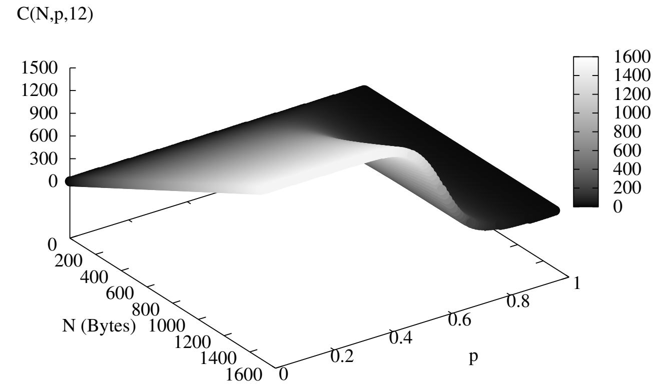
where the rightmost term represents the average number of bytes inspected when a run of length L has been found and the left term represents the average number of inspections when no runs have been found.

4.2.2 Optimized algorithm

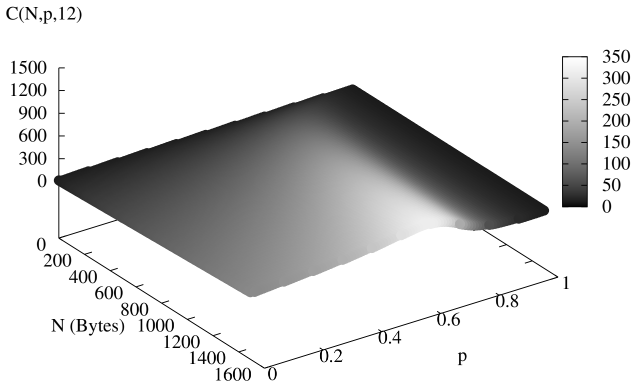
The reduction in cost for the optimized algorithm ranges from a factor of $1/L$ inspections for pure non-ASCII packets to no gain for those packets whose L first bytes are ASCII. In paying attention in between these figures, we calculate the average cost for the optimized algorithm ($C_o(N,p,L)$) following a recurrent approach described by the next equation:

$$C_o(N,p,L) = \begin{cases} N, & if \quad N \leq L \\ L \cdot p^L + \sum_{i=0}^{L-1} (1-p) \cdot p^i \cdot (i + 1 + C(N-(L-i),p,L)), & if \quad N > L \end{cases} \tag{2}$$

where $L \cdot p^L$ represents the number of byte inspections when a run of length L is found. Otherwise, the number of backward inspections carried out until a non-ASCII byte is found is accounted along with the average cost for the rest of the packet (the recursive call). Note that i represents the number of backward inspections whose probability is $(1-p) \cdot p^i$.



((a)) Vanilla



((b)) Optimized

Figure 7: Number of inspected bytes for algorithms for different packet sizes and ASCII-occurrence probabilities ($L = 12$)

4.2.3 Discussion

To compare the cost of both algorithms, we have numerically evaluated equations 1 and 2. In this way, Fig. 7 shows the average cost varying the packet length (N) and the ASCII-occurrence probability (p). It becomes apparent that the optimized algorithm outperforms the vanilla one. As expected, for pure ASCII packets ($p = 1$), there is no gain while for pure non-ASCII ($p = 0$) the gain peaks at 12 (namely, the best case). The optimized algorithm's shape ranges between narrow intervals (up to 300 inspections) for all possible points of operation. Similarly, the vanilla algorithm also presents a flat-like behavior but only for $p > 0.7$ which are the less relevant values (0.4 for ASCII and similar values for other codification schemes). On the other hand, for values below 0.7 the number of comparisons increment progressively with the number of bytes up to 1500 comparisons. In this way, from a practical point of view, the most common cases (e.g. $p = 0.4$ and $N = 600$) are located closer to the best-case gain.

In paying attention to different run lengths, several numerical evaluations were carried out sweeping values from $L = 2$ to $L = 18$. Figure 8 depicts the average cost for 1500-byte packets varying the detection run length and the probability p. The average costs in all cases are below some 520 comparisons even using minimum values of L. Long run thresholds benefit from low ASCII-occurrence probabilities given that the packet is traversed in long jumps. In contrast, short run thresholds benefit progressively from the increment of p as less backward inspections are carried out as well as from the higher possibility of occurrence of a run.

5 Selective capping sniffer architecture

The implementation of the selective capping sniffer is based on a high performance capture engine, HPCAP [22], and follows a three-step architecture, as shown in Fig. 9.

1) Incoming traffic is split into two categories using hardware filters such as Intel flow director [10]: well-known (in the protocol/port sense) traffic that network managers do want to keep, and other traffic that managers want to cap to transport headers or keep complete if detected as ASCII. That is, each traffic set is redirected to a different receive side scaling (RSS) driver queue.

2) The selective capping algorithm is applied to detect non-ASCII packets and truncate them. As a first approach, the network driver layer is entrusted with this task [21], Fig. 9 (a). The motivation is to apply the selective capping as soon as possible in the network stack, avoiding useless bytes traversing the stack, and, importantly, provide users with thinned traffic in a total transparent way. That is, any application that requests traffic will receive the payloads capped without requiring different operation from a vanilla driver.

This implies a careful low level hardware-software interaction, although this also provides some advantages. For example, our implementation exploits the advanced packet descriptor features that

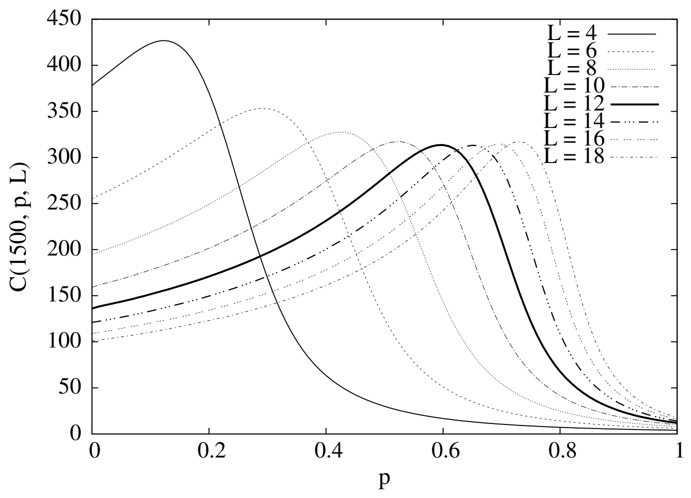


Figure 8: Optimized algorithm's cost (C) for different run lengths (L) and ASCII-occurrence probabilities (p), ($N = 1500$)

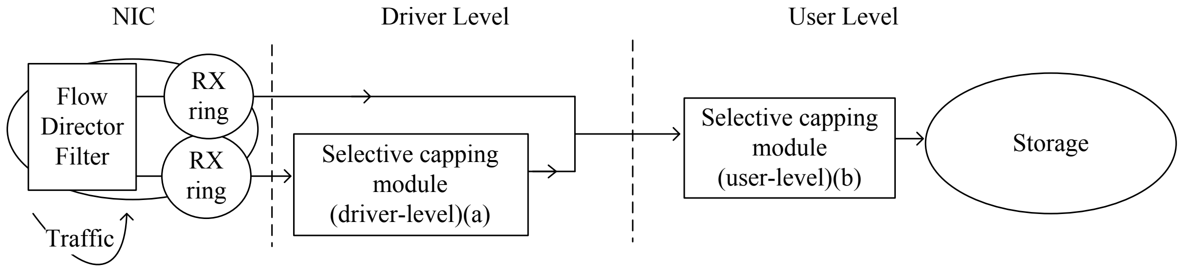

NIC
Driver Level
User Level
Flow Director Filter
RX ring
RX ring
Traffic
Selective capping module (driver-level)(a)
Selective capping module (user-level)(b)
Storage

Figure 9: Selective capping traffic sniffer architecture

modern NICs offer. Such descriptors provide the protocol stack and header lengths coded in NIC hardware, and so packets can be capped to the protocol headers without any extra processing.

Alternatively, the filtering process may be executed at user level, as illustrated in Fig. 9 (b). Rather than directly filtering the data received from the driver, a buffer is inserted between the driver and storage process to absorb traffic peaks. The driver writes all the captured traffic to the buffer, then, a user level thread reads from the buffer, applies the ASCII detection method, and writes capped packets to the storage buffer. Once the storage buffer is full, data moves to the next stage. At this point, both options converge.

On the upside, this architecture provides more capacity to process traffic. In the first option, both capture and filter processes share execution on the same CPU, while the latter option allows moving the filtering task to another CPU, possibly more than one CPU could be employed where tasks such as flow analysis or DPI could be applied if desired. If the addition of the capture and filter processes exceeded the CPU capacity, the first option would result in losses, whereas the user level approach allows us to exploit parallelizing techniques.

Both driver modifications and software module developed for each approach, respectively, are available under an open-source license at [23].

3) Capped and uncapped packets are stored at user level along with a PCAP like header on a high-performance storage solution for subsequent access, e.g. [22]. Traffic can also be forwarded to an on-the-fly analysis system [24], or a general purpose compression system, e.g. gzip, which is particularly effective for ASCII content.

6 Performance Evaluation

We evaluated the performance of the proposed system using a commodity server that contained 96 GB of DDR3 memory, two 6 core Intel Xeon E52630 processors running at 2.30 GHz, on a Supermicro X9DR3 motherboard. The server also had an Intel RS25DB080 RAID controller with a RAID-0 volume composed of 12 high end mechanical hard disks, Hitachi HUA723030ALA640, that allowed

Table 2: Traffic shares in different environments

Protocol	% in Enterprise	% in ISP	% in Domestic
RTP	5	4	5
SIP	2	0.8	1
HTTP	35	42	50
DNS	3	6	5
LDAP	2	3	0
Netflow	3	5	0
Radius	0	1	0
SAS	10	0	0
SAMBA	2	0.5	1
SNA	3	4	0
UPnP	0	0	1
SSL	12	14.5	17
TNS	5	0.2	0
P2P	0	15	19
SMTP	8	4	1
SSH	10	0	0

effectively traffic storage at 10 Gb/s. The server network interface was an Intel 10 Gb/s NIC based on the 82599 chip, in a PCIe 3.0 slot, and connected to an FPGA based traffic generator [25] able to replay PCAP traces at variable rates up to 10 Gb/s regardless of packet size. This system allowed two classes of experiments: memory-replayed, which tested the algorithm by reading traces previously loaded in memory; and FPGA-replayed, which represents a real scenario where traffic from a link is received by the NIC, forwarded to the capping module, and the resulting traffic is stored on a RAID-0 volume.

We used three traces to evaluate the performance as representative scenarios of domestic, enterprise, and ISP environments. These traces were synthetically constructed by merging the subtraces discussed in Section 4. The protocol shares for each environment were based on our experience in monitoring these types of networks. Table 2 details the distribution, in volume, of each protocol in each trace. The three traces have a total size of 2 GB, and all protocols are uniformly distributed across time. The maximum throughput that can be achieved on a 10 GbE link depends on the trace average packet size due to the minimum inter-frame gap of the standard. Thus, the maximum achievable throughput is 9.23, 9.11, and 8.26 Gb/s respectively. These were our targets for the FPGA-replayed experiments. For each experiment, the traffic traces were replayed for 30 minutes to avoid cache effects.

The achieved throughput rates results are shown in Table 3 and compression rates in Table 4. Processing throughputs for the FPGA-replayed scenario at driver level are 9.22, 9.00, and 8.25 Gb/s for the domestic, ISP, and enterprise environments respectively, with mean packet loss rate 0.3%. These suggest that after receiving each packet, the driver has minimal time for performing additional processing tasks, i.e., the driver level approach is very close to the processing limit with these traces.

Using FPGA-replayed traffic applying the filtering algorithm at user level, the achieved throughputs are 9.23, 9.11, and 8.26 Gb/s for the domestic, ISP and enterprise environments, respectively, and no packet losses were observed. This supports the hypothesis of minimal free processing capability with the driver level application, whereas the user level option is only limited by the capture process and the filtering task is not a relevant challenge for the CPU. Thus, for more demanding scenarios (e.g. 40 Gb/s NIC), the user level option is likely to be the best candidate. The in-memory experiments, where no 10 Gb/s limitation is present, confirmed the hypothesis, as the obtained throughput was higher than 10 Gb/s. Note that the maximum achieved throughput is highly dependent on the CPU frequency. The CPU's in this system are not high end and the figures may be surpassed if more powerful processors were used.

Regarding compression, space saving between two and three times is achieved by capping nearly 60% of the packets for the three traces (Table 4).

To evaluate the best and worst cases for the proposed algorithm and how different parameters and traffic characteristics affect its performance, the in-memory test was performed over each protocol described in Table 1 independently. The throughput rates achieved are shown in Table 5, and the

Table 3: Performance and packet loss for different traces

	Trace		
	Domestic	**ISP**	**Enterprise**
Avg. throughput (Gb/s) driver-level	9.22	9.00	8.25
Avg. Packet Loss (%) driver-level	0.31	0.30	0.32
Avg. throughput (Gb/s) user-level	9.23	9.11	8.26
Avg. Packet Loss (%) user-level	0.00	0.00	0.00
Avg. throughput (Gb/s) Memory	10.49	11.71	13.38

Table 4: Compression for different traces

Trace	**Compression rate**	**% of capped packets**
Domestic	3.31	64.37
ISP	3.08	62.36
Enterprise	2.32	63.91

Table 5: Performance for different protocols (in memory)

Protocol	**Avg. throughput (Gb/s) ± Std. Dev.**	**Avg. packet rate (kpps) ± Std. Dev.**
RTP	9.26 ± 0.02	$7,237.90 \pm 15.76$
SIP	51.95 ± 0.40	$26,666.35 \pm 209.75$
HTTP	12.77 ± 0.65	$4,847.15 \pm 249.19$
DNS	11.36 ± 0.02	$11,656.66 \pm 20.76$
LDAP	15.61 ± 0.11	$11,268.51 \pm 86.23$
Netflow	30.63 ± 0.02	$9,593.67 \pm 8.54$
Radius	37.92 ± 0.21	$16,059.85 \pm 92.33$
SAS	59.60 ± 2.90	$7,659.51 \pm 372.74$
SAMBA	14.66 ± 0.08	$15,046.49 \pm 87.97$
SNA	15.90 ± 0.02	$23,654.85 \pm 42.28$
UPnP	9.63 ± 0.02	$2,775.15 \pm 6.60$
SSL	9.73 ± 0.09	$2,340.87 \pm 22.50$
TNS	20.40 ± 0.37	$17,987.15 \pm 332.09$
P2P	7.90 ± 1.12	$1,636.27 \pm 232.11$
SMTP	60.34 ± 0.21	$19,695.01 \pm 70.58$
SSH	9.07 ± 1.38	$1,049.43 \pm 160.45$

Table 6: Compression for different protocols

Protocol	**Compression rate**	**% of capped packets**
RTP	3.89	74.35
SIP	1.12	54.06
HTTP	2.86	59.37
DNS	1.25	17.42
LDAP	1.13	30.08
Netflow	5,263.16	100.00
Radius	1.20	38.23
SAS	1.02	29.08
SAMBA	5.60	93.11
SNA	5.10	94.88
UPnP	70.44	97.94
SSL	9.67	93.08
TNS	1.23	54.58
P2P	45.96	62.33
SMTP	1.00	0.05
SSH	135.77	99.34

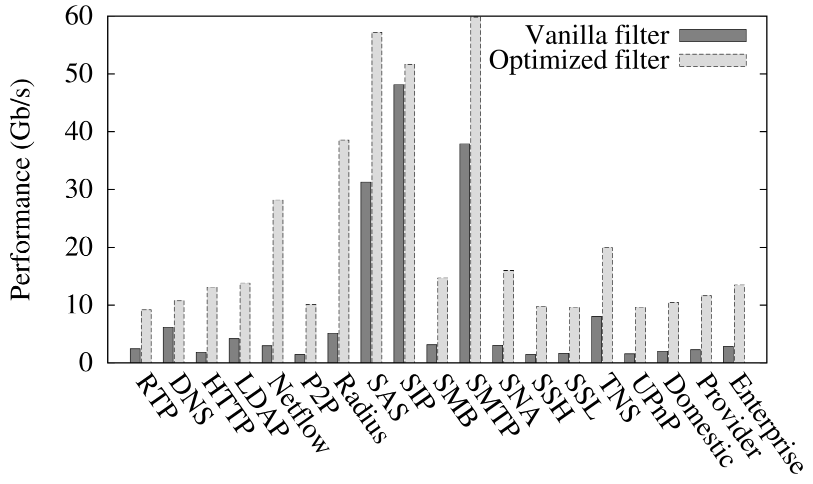


Figure 10: Performance of each algorithm for different traces (in memory)

compression rates in Table 6. The highest throughput is achieved with pure ASCII protocols that present a large average packet size (such as SAS), since only the first bytes are inspected. The worst performance is achieved for protocols that show high compression rates, as they present binary payloads and the proposed algorithm must inspect all payload bytes, in the worst case scenario, searching for ASCII runs.

Figure 10 compares the proposed vanilla and optimized algorithms for each of the protocols and traces discussed above, executing both algorithms in memory, otherwise throughput would be capped at 10 Gb/s due to the link speed. As expected, overall gain from the optimized algorithm is high, particularly for binary protocols. Interestingly, the figure shows the potential gain that could be obtained using the proposed optimized algorithm in a production environment with a 40 Gb/s link, showing that it can manage throughput five times larger than the experimental cases.

7 Conclusions

We presented an effective and novel solution for performing selective packet capping based on retaining the payload of packets with useful content for interpretation by network managers and analysts. This subset of packets is composed of ASCII protocols as well as well-known binary protocols (e.g. DNS), and are stored unchanged, whereas the rest of the packets are truncated to their header.

To detect ASCII-based protocols, we constructed a taxonomy of how ASCII content is carried, and found significant diversity. This led us to develop two mechanisms to identify packets as ASCII based on ASCII run detection, and the total percentage of ASCII within a payload. Unfortunately, the vanilla implementation was unable to achieve multi-Gb/s rates, and further optimization was required, inspecting a carefully chosen subset of bytes of the payload rather than all the bytes. Indeed, the analytical study of the optimized algorithm's cost showed a significant reduction in the number of inspections, while still keeping the FP rate below 10% and the FN rate below 1% with respect to the vanilla algorithm. Moreover, experiments showed that for real traffic traces, space saving was achievable of 2–3 times when capping packets.

We implemented the proposed algorithm as a real system, comprising two tasks, to capture and then cap traffic. The algorithm was implemented both as a modification of the default Intel driver and as a module at the application layer. While the former provides a more transparent mechanism without user level interaction, the latter allows exploiting CPU parallelism and, consequently, enhanced performance.

The driver based implementation achieved slightly less than 10 Gb/s with packet loss of 0.3% using real traffic traces, whereas the user level implementation did not show any packet loss for the same cases. Replaying traces in memory, processing throughput ranged 11–14 Gb/s for real traffic traces, and peaked over 40 Gb/s for text based protocols, such as SAS or SMTP. We conclude that the driver based implementation is very close to the processing limit, i.e., the combination of capture and the proposed algorithm on the same CPU achieved maximum rates close to, but lower than the maximum defined in the 10 GbE standard. On the other hand, the user level approach achieved such figure without losses and peaks over 40 Gb/s in memory.

To address 100 GbE standard, future work should be conducted to improve the capture process by devising suitable low level hardware-software interactions, further parallelism paradigms, or support from FPGA or GPU devices.

Acknowledgements

The authors would like to thank the anonymous reviewers for their useful and enriching comments. This research was carried out with the support of the European Union's project Fed4FIRE (FP7-ICT-318389).

References

[1] V. Moreno, J. Ramos, P. Santiago del Rio, J. Garcia-Dorado, F. Gomez-Arribas, and J. Aracil, "Commodity packet capture engines: tutorial, cookbook and applicability," *IEEE Communications Surveys Tutorials* , vol. 17, no. 3, pp. 1364–1390, 2015.

[2] M. Forconesi, G. Sutter, S. López-Buedo, J. E. López de Vergara, and J. Aracil, "Bridging the gap between hardware and software open-source network developments," *IEEE Network*, vol. 28, no. 5, pp. 13–19, 2014.

[3] A. N. Mahmood, J. Hu, Z. Tari, and C. Leckie, "Critical infrastructure protection: Resource efficient sampling to improve detection of less frequent patterns in network traffic," *Journal of Network and Computer Applications*, vol. 33, no. 4, pp. 491–502, 2010.

[4] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider, "Enriching network security analysis with time travel," in *ACM SIGCOMM*, 2008, pp. 183–194.

[5] Y.-D. Lin, C.-N. Lu, Y.-C. Lai, and Z. Eliezer, "Bug traces: identifying and downsizing packet traces with failures triggered in networking devices," *IEEE Communications Magazine*, vol. 52, no. 4, pp. 112–119, 2014.

[6] J. van der Merwe, R. Cáceres, Y.-H. Chu, and C. Sreenan, "Mmdump: A tool for monitoring internet multimedia traffic," *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 5, pp. 48–59, 2000.

[7] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *ACM SIGCOMM*, 2000, pp. 87–95.

[8] D. Hoplaros, Z. Tari, and I. Khalil, "Data summarization for network traffic monitoring," *Journal of Network and Computer Applications*, vol. 37, pp. 194–205, 2014.

[9] CAIDA, "Caida data - overview of datasets, monitors, and reports," http://www.caida.org/data/overview/ [20 Jan 2016].

[10] Intel, "82599 10 Gbe controller datasheet," 2012, http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html, [20 Jan 2016].

[11] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better NetFlow," in *ACM SIGCOMM*, 2004, pp. 245–256.

[12] J. L. García-Dorado, J. Aracil, J. A. Hernández, and J. E. López de Vergara, "A queueing equivalent thresholding method for thinning traffic captures," in *IEEE/IFIP Network Operations and Management Symposium*, 2008, pp. 176–183.

[13] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer, "Building a time machine for efficient recording and retrieval of high-volume network traffic," in *ACM Conference on Internet Measurement*, 2005, pp. 267–272.

[14] Y.-D. Lin, P.-C. Lin, T.-H. Cheng, I.-W. Chen, and Y.-C. Lai, "Low-storage capture and loss recovery selective replay of real flows," *IEEE Communications Magazine*, vol. 50, no. 4, pp. 114–121, 2012.

[15] A. Papadogiannakis, M. Polychronakis, and E. P. Markatos, "Stream-oriented network traffic capture and analysis for high-speed networks," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 10, pp. 1849–1863, 2014.

[16] T. Taylor, S. E. Coull, F. Monrose, and J. McHugh, "Toward efficient querying of compressed network payloads," in *USENIX Annual Technical Conference*, 2012, pp. 113–124.

[17] F. Fusco, M. Vlachos, and X. Dimitropoulos, "Rasterzip: Compressing streaming network monitoring data with support for partial decompression," in *ACM Internet Measurement Conference*, 2012, pp. 51–64.

[18] V. Jacobson, C. Leres, and S. McCanne, "libpcap," *Lawrence Berkeley Laboratory, Berkeley, CA*, 1994.

[19] naudit, "Detect-pro," 2013, http://www.naudit.es/, [20 Jan 2016].

[20] F. Schneider, B. Ager, G. Maier, A. Feldmann, and S. Uhlig, "Pitfalls in HTTP traffic measurements and analysis," in *Passive and Active Measurement Conference*, 2012, pp. 242–251.

[21] V. Uceda, M. Rodríguez, J. Ramos, J. L. García-Dorado, and J. Aracil, "Selective capping of packet payloads for network analysis and management," in *Workshop on Traffic Monitoring and Analysis*, 2015, pp. 3–16.

[22] V. Moreno, P. M. Santiago del Río, J. Ramos, J. L. García-Dorado, I. Gonzalez, F. J. Gómez-Arribas, and J. Aracil, "Testing the capacity of off-the-shelf systems to store 10Gbe traffic," *IEEE Communications Magazine*, vol. 53, no. 9, pp. 118–125, 2015.

[23] CapAs, "ASCII-based traffic capping solution," 2015, https://github.com/hpcn-uam/CapAs, [20 Jan 2016].

[24] A. Finamore, M. Mellia, M. Meo, M. Munafò, and D. Rossi, "Experiences of Internet traffic monitoring with Tstat," *IEEE Network*, vol. 25, no. 3, pp. 8–14, 2011.

[25] J. Zazo, M. Forconesi, S. Lopez-Buedo, G. Sutter, and J. Aracil, "TNT10G: A high-accuracy 10 GbE traffic player and recorder for multi-Terabyte traces," in *Conference on ReConFigurable Computing and FPGAs*, 2014, pp. 1–6.