Performance evaluation of open-source software for traffic traces manipulation and analysis

German Retamosa and Javier Aracil

Abstract—**The paper presents a performance evaluation of commonly-used open-source software for manipulation and off-line analysis of traffic traces. In traffic analysis, there is a trade-off between either implementing ad-hoc, low-level software that is optimized for performance or using off-the-shelf open-source software written in high-level languages such as Python. Clearly, the former approach has a penalty in development time. While using high-level languages is easier in terms of development, the size of traffic traces is increasing and so is the processing time. We conclude that the use of high-level languages provides similar processing times in comparison with the low-level languages counterpart, provided that a pre-filtering of the traffic can be performed (by means of tcpdump).**

Index Terms—**Traffic analysis, Performance Analysis and Design Aids, Python.**

I. INTRODUCTION

NOWADAYS, network infrastructures represent a significant share of the budget in large companies or institutions, thus requiring an accurate capacity planning. Furthermore, the quality of service provided by the network has to be carefully monitored, because an increasing part of customer relationship is carried out through the network. Finally, there are many possible security threads, that can eventually lead to service outages, with severe impact in the organization.

Traffic analysis comes as a fundamental tool for capacity planning, quality of service assessment and reinforcement of security. By traffic analysis we understand the in-depth study of network traffic to extract conclusions about network and application behavior. To this end, we distinguish the following three phases in traffic analysis.

First, there is the traffic capture, which consists of passive traffic sniffing by means of a tap or SPAN port into a traffic probe. Such probe is normally equipped either by specialized hardware for traffic capturing or by commodity hardware. Recently, the use of commodity hardware for traffic sniffing at 10 Gbps has attracted considerable attention, because of the reduced cost and good performance. Actually, the authors in [1] report 10 Gbps packet capturing capability with commodity hardware and no packet loss at minimum packet size. Note that the number of packets per second in the monitored link matter, because the DMA interrupts from the network interface card happen at packet batches. Thus, for the same rate in bits per second, the larger the packet size the smaller the DMA interrupt rate.

Second, there is a preprocessing phase which is a ”thinning” of the traffic for storage and subsequent analysis. As the network bandwidth increases to 10 Gbps and beyond it becomes impractical to store raw traffic traces. As an alternative, either the packet payload may be chopped to, say, 100 bytes or the packets may converted into *flows*. The latter is straightforward and it has reduced impact on traffic analysis because the application layer control information (for example the URL) usually appears in the first 100 bytes of the packet. The former requires more computing power in real time because packets are grouped into flows (same source and destination IP address, port and protocol) by means of a hash table in real time. Then, the flow record is dumped to permanent storage. Such flow record contains basic flow data such as flow size, duration, etc, and it may also contain the first bytes of the flow payload in order to drill down in the traffic content. Note that flows can be unidirectional (UDP) or bidirectional (TCP) and the sorting of packets at high speed into flows poses a major performance bottleneck.

Third, there is the processing of the network traffic. While part of the processing can be performed on-line (flow records), specially for alarm reporting, there is a large off-line analysis on the traffic trace. Such off-line analysis of the traffic trace normally involves the in-depth dissection of network protocols and applications in order to assess possible performance bottlenecks. Actually, this falls within the Application aware network performance monitoring (AA-NPM) area, which has been recently identified by Gartner as one of the top strategic areas in information technologies [17].

This paper provides a performance evaluation of commonly-used open-source software for the off-line manipulation of data traffic. Such performance evaluation is interesting for researchers and practitioners in the field because it provides an assessment of development cost versus flexibility and execution time of high-level languages such as *Python* [2]. Thanks to these kind of scripting languages, code prototyping for traffic analysis is a simple task. However, this comes at the expense of a larger execution time, which may deem the traffic analysis programs useless for high-speed environments. As an alternative, ad-hoc traffic analysis programs can be used that make optimized use of the memory and CPU, possibly performing parallelization among the CPU cores. However, this comes at the expense of a higher development cost. This tradeoff is investigated in the paper, which reviews the most popular *DPKT* [4], *Pcapy* [6], *Impacket* [6] approaches for early-prototyping of traffic analysis programs.

Manuscript submitted August 31, revised September 20, 2012. This work was supported by the IEEE.

G. Retamosa is with Naudit High Performance Computing and Networking, Faraday, 7, 28049 Madrid SPAIN (phone: +(34) 91 116 99 40; email: german.retamosa@naudit.es).

J. Aracil is with Universidad Autonoma de Madrid, Francisco Tomas y Valiente, 11, 28049 Madrid SPAIN (phone: +(34) 497 22 72; email: javier.aracil@uam.es).

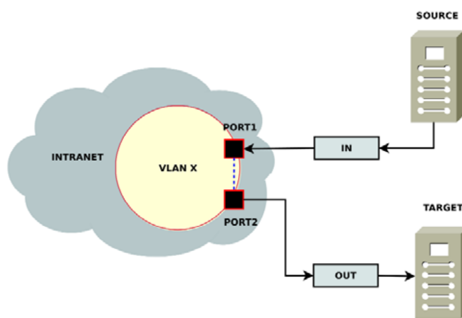


Fig. 1. Traffic duplication with SPAN port over VLAN Architecture

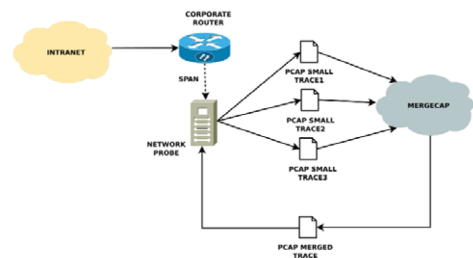


Fig. 2. *Mergecap* Architecture

Section II provides an overview of the off-line traffic analysis process, namely what are the common steps for large traffic trace handling and processing. Section III is devoted to protocol dissectors and, finally, section IV presents a use case, followed by conclusions about the assessment work carried out.

II. Software for traffic traces manipulation

Offline traffic analysis involves the following consecutive steps:

1 Merging: This is the procedure to merge several short traces into a larger one. In most cases, the traffic sniffers produce relatively small traces (1 GB worth of traffic with approximately 2,4 millions of packets), which have to be merged in order not to miss flows that have the starting part in one file and the ending part in another subsequent file.

2 Traffic deduplication: The figure 1 shows an span port on a switch or router VLAN. As it turns out, packets come out of the SPAN twice because the same packet inbound to the VLAN is transmitted eventually outbound of the VLAN. Consequently, there is a packet copy. However, chances are that the packet is not a complete duplicate but the same packet with TTL field decremented by one. This is case when there is an intermediate router in the VLAN under analysis and the packet goes in and out of such intermediate router.

3 Flow analysis: Once the traffic trace has been totally deduplicated the traffic flows have to be identified at different levels: MAC address, IP address, unidirectional flows and bidirectional flows. Normally, UDP flows are unidirection and TCP flows are bidirectional. For both, a flow record is formed that includes statistics such as origin and destination port, number of SYNs being interchanged, TCP segments with the RST flag, etc.

4 Protocol dissection: For the most common services, application performance monitoring is usually performed, such as for instance in HTTP or LDAP services. The procedure involves the extraction of the most common control fields of the protocol, per packet. Then, packets are paired (for instance GET and reply to GET in HTTP) and further statistical treatment is performed, which results in calculation of the service response time, for example (from GET to response to GET).

In what follows we describe and evaluate the software for merging, deduplication and flow analysis.

A. Merging

Merging is the first procedure that all networking software for traffic traces manipulation has to deal with. As we see in 2, traffic merging applications are a very important part in traffic monitoring because they provide a single unified traffic trace to protocol dissectors, from the many different files that are provided by the sniffer.

Traffic sniffers usually dump traffic into small files, say 1 GB worth of traffic with approximately 2,4 millions of packets. That way, the sniffer is continuously dumping traffic from memory to files, instead of keeping the whole traffic capture in memory and dumping it to file afterwards. Note that the latter would be impractical for large traffic capture periods.

In this section we analyze the architecture of merging applications. More specifically, we consider the strengths and drawbacks of the commonly used *mergecap* [11], which combines multiple saved capture files into a single PCAP output.

One of most important problems in merging applications, and specially in applications like *mergecap* [11], is the possible misordering of the packet timestamps. Actually, the packet sequence may be re-ordered when several traffic traces are merged together. Figure 3 left shows a time series of bytes per time interval with this common misordered timestamping problem in the merged traffic trace. As shown, there are noticeable temporal jumps. However, the right graph does not contain this kind of errors.

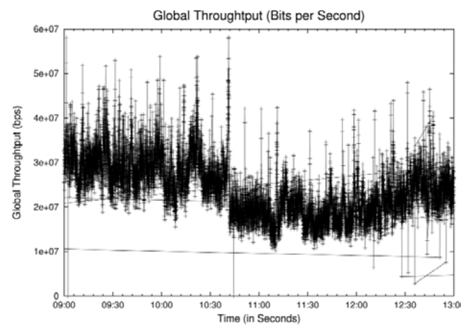
To cope with this issue it is very important to comply with the following guidelines.

1 Before calling *mergecap*, sort chronologically the PCAP input files in order to append them in chronological order and to reduce the computational load of *mergecap* [11], which employs quadratic algorithms to sort input files.

2 Do not use the exclusive append option, i.e. the *-a* option in *mergecap*, because it disables sorting algorithms in the application.

3 Sort the merged data by timestamp before processing it with traffic dissectors. Our extensive *mergecap* experimentation shows that this final step is necessary, even though the application has already sorted the packets in the merged file.

After the merging has been performed, traffic deduplication follows.



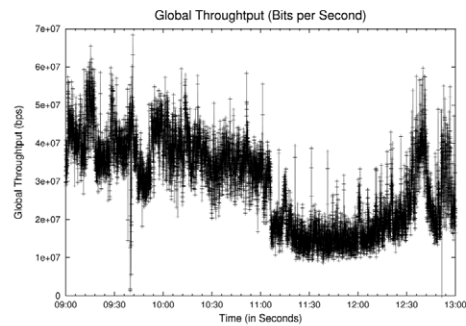


Fig. 3. Comparison of time series (unsorted left, sorted right)

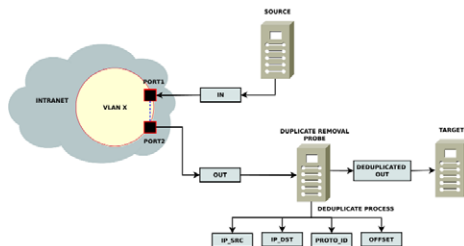


Fig. 4. Custom Duplicate Removal Architecture

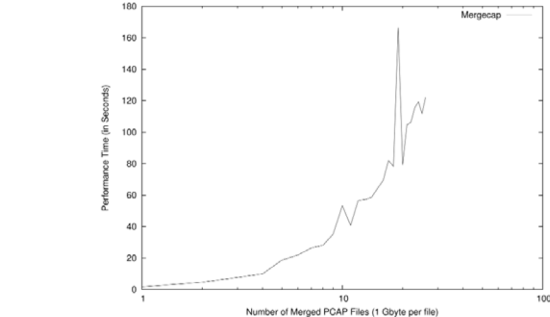


Fig. 5. *Mergecap* Performance

B. Deduplication

Typically, packet duplicates appear when sniffing traffic directly from a VLAN with input and output ports. They can also be due to routing errors. Depending on the architecture, and depending on its elements, duplicates may appear at any TCP/IP stack level.

Using a SPAN port is a very common procedure to capture traffic. The SPAN port is a special port which is configured in the router and provides a copy of the traffic, packet by packet, of either another port or a complete VLAN. For the latter, note that a packet that enters the VLAN from one port will normally leave the VLAN by another different port. Since both ports are part of the monitored VLAN it turns out that the same packet is copied twice to the SPAN.

As a result, we get a duplicate that has to be removed from the traffic trace. Other duplicates may arise when a packet is routed out of a port being monitored, then it goes through another router and then it is injected back into the same port. However, in that case the TTL field is decremented by one. Broadly speaking, we have the following types of duplicates:

1 Duplicates with the same payload but with a slight difference in the header, i.e. sequence number, TTL value and IP/Ethernet values.
2 Duplicates with the same payload and header structure.

First, duplicates with the same payload but with some differences in header fields are mainly due to routing issues or active equipment (firewalls) that duplicate traffic. For this kind of duplicates, the only way to remove them is with a custom application, whose architecture can be shown in figure 4, which evaluates whether the packet payload and IP and TCP/UDP relevant fields (address, port, etc) are indeed the same.

Second, *Editcap* [13] is a popular application to remove duplicates with the same payload and header. It is part of the *Wireshark* [14] framework.

C. Performance evaluation of merging and deduplication

In this section we evaluate the performance of the merging and deduplication procedures.

1) Merging: On the one hand, figure 5 shows the time it takes to merge several 1 Gbyte size files. The X-axis shows the number of files to be merged.

The *Mergecap* execution time increases exponentially with the size of the files and the number of traces to be merged. As it turns out, the packet-reordering process is a potential performance bottleneck, specially when the number of files to be merged is very large. In any case, the execution times are manageable and it is not worthwhile to code ad-hoc applications with performance optimizations.

2) Deduplication: On the other hand, figures 6 and 7 show the execution time of an ad-hoc application and *editcap*. The X-axis shows the size (in GBytes) of the traffic trace to be deduplicated.

The execution time for traffic deduplication, using either ad-hoc applications or editcap, increases exponentially with the number of duplicates that are present in the original PCAP traces. Furthermore, both applications have structural differences because editcap [13] is intended to remove identical packets and our custom application for duplicate removal eliminates packets that have the same payload, source and destination IP address and source and destination TCP/UDP ports.

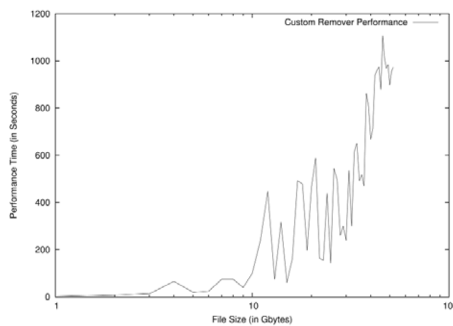


Fig. 6. Custom duplicate removal performance

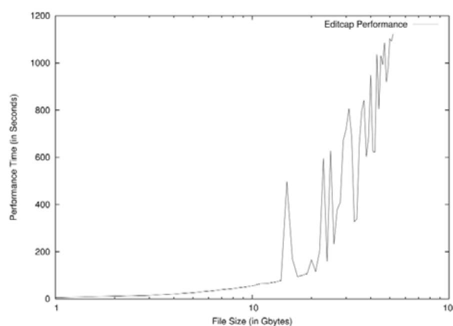


Fig. 7. *Editcap* performance

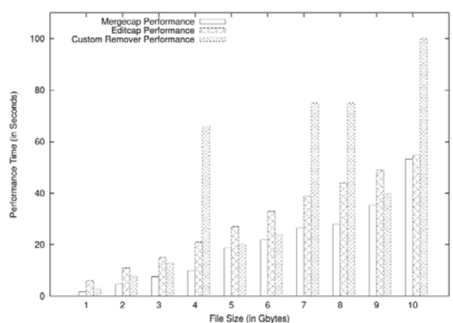


Fig. 8. Performance Comparison

Another potential performance bottleneck is the window size for deduplication, namely the number of packets that are compared with the deduplication process. This means that larger window size involves matching a larger batch of packets in a single pass and thus, impacting in the granularity of the algorithm and a better performance due to the number of packet matches to be carried out during the deduplication phase.

As a conclusion, figure 8 and specially with file sizes of 4, 7, 8 and 10 Gbytes, shows that when the number of duplicates, either identical packets or with some differences in header fields, is very high the execution time of deduplicating PCAP traces is higher than merging them due to the internal processing of PCAP packets with DPI libraries. In all other cases, their execution times and performance are close related between them.

Finally, as we see in figure 8, we conclude that the execution times of merging PCAP traces with *mergecap* is lower than deduplicating them with custom applications or *editcap* in all cases evaluated.

III. Software for traffic dissection

Developing a software application for traffic dissection is one of the most important issues in Application aware network performance monitoring (AA-NPM) but it is not a simple task. On the one hand, there is a set of communication protocols that must be selected according to user needs and studied to obtain the most important parameters. On the other hand, we have to take into account the prototyping time, analysis depth and code maintenance among others, because they are relevant aspects for the choice of the programming language and also for the overall performance. In this section, we evaluate some of the main traffic dissection programming techniques in terms of performance and timing.

A. Python *Libraries*

Python is a general-purpose and high-level programming language whose design philosophy emphasizes features such as code readability, fast prototyping and simple maintenance and as a result, It is becoming increasingly popular in recent times. Furthermore, due to its strong developer community, it is well supported and there are new versions and patches that are continuously posted.

In terms of networking and third-party libraries for traffic dissection, the most important ones are *DPKT* [4], *PCAPY* [6] and *SCAPY* [9]. *Python DPKT* [4] is a third-party library hosted into *Google Project Hosting* [3], which is maintained periodically. The following code snippet is a HTTP traffic dissection software written in *Python* [2] and *DPKT* [4] library and it is divided into three tasks: load libraries, parse PCAP files and analyze network traffic.

```
import dpkt

f = open('test.pcap')
if f:
    pcap = dpkt.pcap.Reader(f)

    for ts, buf in pcap:
        eth = dpkt.ethernet.Ethernet(buf)

    ip = eth.data
    tcp = ip.data

    if tcp.dport == 80 and len(tcp.data) > 0:
        http = dpkt.http.Request(tcp.data)
        print http.uri

    f.close()
```

Code Snippet 1. *Python DPKT* sample code

First of all, all *Python* [2] scripts must define or import all third-party dependencies (*DPKT* [4] in this case). Thus, the *Python* [2] application can use all methods inherited from those libraries.

Once the *DPKT* [4] module is loaded, the application opens the offline PCAP file (after the merging) and parses it with *DPKT* routines for networking traffic dissection.

```
f = open('test.pcap')
if f:
    pcap = dpkt.pcap.Reader(f)
```

Code Snippet 2. *Python DPKT* file reading

Finally, code snippet 3 shows the analysis process for each flow record. As shown, it is quite simple with this library and supports a large number of protocols such as LDAP, DNS or HTTP.

```
for ts, buf in pcap:
    eth = dpkt.ethernet.Ethernet(buf)

    ip = eth.data
    tcp = ip.data

        if tcp.dport == 80 and len(tcp.data) > 0:
        http = dpkt.http.Request(tcp.data)
        print http.uri
```

Code Snippet 3. *Python DPKT* file parsing

From the above code examples, it turns out that prototypes can be coded in a very short time and that maintenance is good but unfortunately, performance and analysis depth are the most important drawbacks for this kind of libraries.

The second networking library to assess is *Scapy* [9], a powerful interactive packet manipulation program written in *Python* in August 2007. As in the previous library, we show a sample code of a HTTP traffic dissection software written in *Python* and *Scapy* library, which also shows the same three steps: loading of the libraries, parsing the PCAP files and analyzing packets.

```
from scapy.all import *

pkts = rdpcap('test.pcap')
for pkt in pkts:
    if pkt.haslayer(TCP):
        if pkt.getlayer(TCP).dport == 80:
            if pkt.haslayer(Raw):
                print pkt
```

Code Snippet 4. *Python Scapy* sample code

Code snippet 4 shows a source code even shorter than other alternatives but slightly different. First of all and like the *DPKT* sample, third-party dependencies are needed to instantiate the *Scapy* methods.

Once the libraries have been loaded, code snippet 5 shows how *Scapy* [9], with *rdpcap* instruction, tries to read, parse and store in-memory at the same time. We will see later the multiple problems that this entails.

```
pkts = rdpcap('test.pcap')
```

Code Snippet 5. *Python Scapy* PCAP reading

Finally and as we see in code snippet 6, packets are filtered to the particular protocol being analysed. The method that *Scapy* [9] used to filter packets according to a specific protocol, like HTTP, is by checking protocol (TCP) and port (80).

```
if pkt.haslayer(TCP):
    if pkt.getlayer(TCP).dport == 80:
        if pkt.haslayer(Raw):
            print pkt
```

Code Snippet 6. *Python Scapy* PCAP filtering

The third library that we assess in this section is the combination of two, *Pcapy* [6] and *Impacket* [6]. *Pcapy* is a module extension that interfaces with the *libpcap* packet capture library and *Impacket* is a collection of *Python* classes focused on providing access to network packets.

As we see in Code Snippet 7, the mixed solution with *Pcapy* and *Impacket* has a code very similar to that of the *DPKT* library with the difference that *Pcapy* has an approach based into n-ary trees, where each child contains the information of each TCP/IP stack layer.

```
from pcapy import open_offline
from impacket.ImpactDecoder import EthDecoder
from impacket.ImpactPacket import IP, TCP, UDP, ICMP

decoder = EthDecoder()

def callback(jdr, data):
    packet = decoder.decode(data)
    child = packet.child()
    if isinstance(child, IP):
        child = child.child()
    if isinstance(child, TCP):
        if child.get_th_dport() == 80:
            print 'HTTP'

pcap = open_offline('test.pcap')
pcap.loop(0, callback)
```

Code Snippet 7. *Python Pcapy+Impacket* sample code

Also, the structure of the code is very similar to *DPKT* starting with reading and parsing the PCAP file (code snippet 8)

```
pcap = open_offline('test.pcap')
pcap.loop(0, callback)
```

Code Snippet 8. *Python Pcapy* PCAP reading

and then filtering and analyzing the network protocol (code snippet 9):

```
decoder = EthDecoder()

def callback(jdr, data):
    packet = decoder.decode(data)
    child = packet.child()
    if isinstance(child, IP):
        child = child.child()
    if isinstance(child, TCP):
        if child.get_th_dport() == 80:
            print 'HTTP'
```

Code Snippet 9. *Python Pcapy* PCAP filtering

As we discussed at the beginning of the section, *Python* is one of the most promising languages of the moment due to its easy integration with web interfaces, fast deployment and prototyping. Furthermore, the growing availability of new

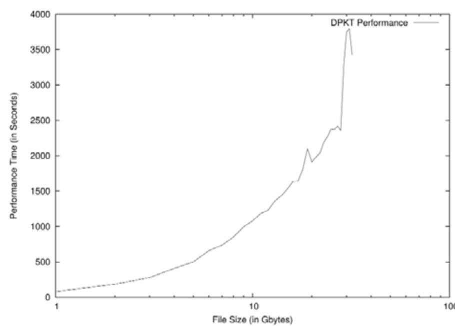


Fig. 9. *Python DPKT* Performance

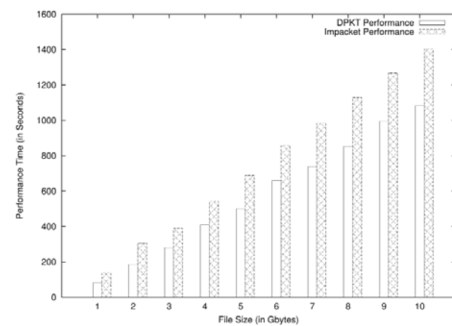


Fig. 11. Performance Comparison

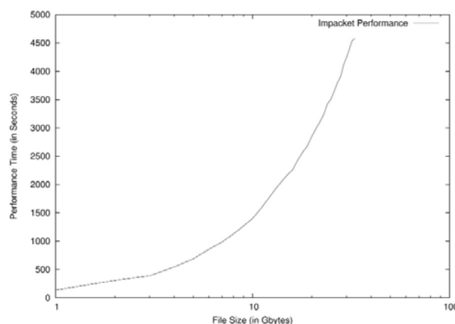


Fig. 10. *Impacket + Pcapy* Performance

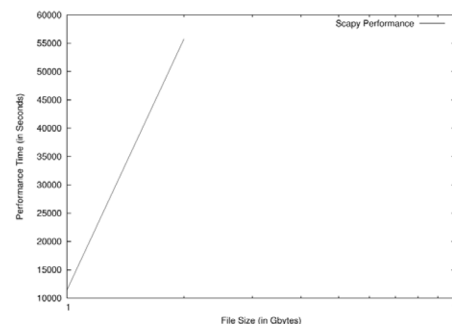


Fig. 12. Scapy Performance

third-party libraries such as *pylibpcap* [7] and *pycap* [8] represents a significant advantage, albeit there is very few documentation.

B. Performance evaluation of Python libraries

Once the most relevant Python libraries for traffic analysis have been presented in the previous section, we will assess them in order to analyze their performance and potential bottlenecks.

This assessment process consisted of measuring the execution time of a trace containing one-day worth of traffic with approximately 300 Gbytes (600 million packets) size with protocols as varied as DNS, HTTP, LDAP, TNS and so on. Furthermore, the testing architecture is composed with the following features: Intel Xeon CPU E5645 with 24 cores of 2.40GHz, 28 Terabytes of HDD, 24 GBytes of RAM and 1 interface Gigabit Ethernet.

As we can see in figures 9 and 10, the execution time increases exponentially with the PCAP file size, represented on X-axis. Furthermore, both libraries are significantly slower than the low-level application counterpart and, as we see in figure 11, their differences are slightly nonexistent.

However, the figure 12 show much worse performance figures. This is due to the internal logic of the application that individually evaluates each packet on a larger protocol list, thus impacting on search times.

As a conclusion, the main drawback of these libraries is the performance compared to low-level applications written in C with *libpcap* [10] packet capture library. Thus, deploying mixed applications combining *Tcpdump* [10] and *Python* [2] appear as the preferred solution in terms of performance.

C. Tcpdump

As we have seen in the previous subsection, *Python* libraries are a good solution for rapid prototyping of traffic dissectors and without an advanced networking and programming skills needed thanks to the abstraction layers that they provide. However, the assessment procedure carried out over these libraries shows that the bottleneck is the execution time against other solutions written in low-level languages like C.

At this point, we propose the best compromise solution between complexity and execution time is the usage of *Tcpdump* [10] in combination with *Python*. As we will show in this section, this technique serves to accelerate the *Python* performance.

Thanks to the profiling and auto-inspection methods that some *Python* packages provides, we have concluded that the *Python* application waste a significant amount of time dissecting each packet. Therefore, a compromised solution is to filter the original trace and to extract the packets subject of analysis (for instance DNS) and, then, using the *Python* application. The filtering procedure is performed with *Tcpdump* as follows

```
tcpdump −r 'input.pcap' −w 'output.pcap' "port 53"
```

Code Snippet 10. *Tcpdump* command example

which results in the extraction of the UDP DNS packets (port 53) from the original trace file *input.pcap* to the output file *output.pcap*. Figure 13 shows a performance comparison between a ad-hoc Python application and a combined solution with Tcpdump and Python. Note that the excution time decreases significantly thanks to the *tcpdump* pre-filtering.

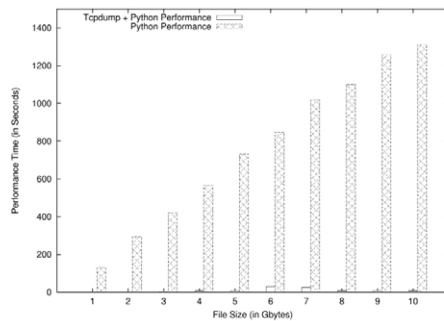


Fig. 13. *Tcpdump* Performance

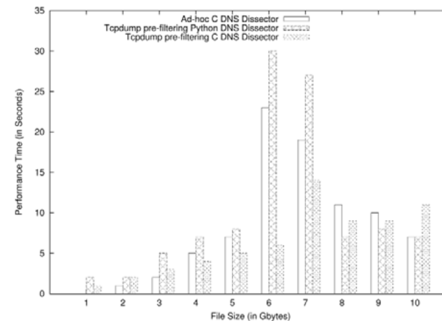


Fig. 14. DNS Traffic Dissector Performance Comparison.

The following section shows, by a simple use case, a practical and consolidated view of the previous assumption.

IV. USE CASE

In this section we show a use case for the DNS protocol. This protocol, explained with more detail into RFC 1034 [15] and RFC 1035 [16], is one of the most well-known communication protocol which provides a hierarchical distributed naming system for computers, services or any other resource connected to the Internet.

By creating a simple DNS dissector with *Python* and *Tcpdump*, it is possible obtain valuable information such as for example IP clients and servers with a high number of connections, IP clients and servers with a high number of errors or temporal series of response times and identification of clients and servers.

Before using the dissector, it is necessary to obtain UDP packets from the original PCAP trace file and filter according with the selected protocol, i.e. 53 port for DNS. As we discussed into the previous section, the usage of tools like *Tcpdump* aims to improve the performance of *Python* dissectors and reduce the workload of Python libraries on the filtered trace.

```
tcpdump -r original.pcap -w filtered.pcap "port 53"
```

Code Snippet 11. *Tcpdump* [10] DNS filtering

Secondly, we will use the *DPKT* library because after evaluating all *Python* libraries, *DPKT* has demonstrated to be the fastest and most complete library for analyzing PCAP traces in *Python*.

As we see in 12, the abstraction layer created by *DPKT* library provides a high level abstraction layer to parse PCAP files with little effort. Furthermore, *Python* [2] provides a useful implementation of lists, dictionaries and tuples that help to reduce the development time significantly.

```
import socket
import dpkt
import sys
import numpy as np

pcapReader = dpkt.pcap.Reader(open('input.pcap'))

pkts_no_ethernet = 0
pkts_IP = 0
pkts_lost = 0

for ts, data in pcapReader:
  try:
    ether = dpkt.ethernet.Ethernet(data)
    if ether.type == dpkt.ethernet.ETH_TYPE_IP:
      ip = ether.data
    if ip.p == 17:
      udp = ip.data
      pkts_IP += 1
      if (udp.dport == 53) and ip.off & 0x00FF == 0:
        dns = dpkt.dns.DNS(udp.data)
      elif (udp.sport == 53) and ip.off & 0x00FF == 0:
        dns = dpkt.dns.DNS(udp.data)
      else:
        pkts_lost+=1
    else:
      pkts_no_ethernet+=1

  except Exception as ex:
    continue
```

Code Snippet 12. *Tcpdump* DNS filtering

As conclusion of the use case, figure 14 shows the results of a performance assessment that compares a traffic dissector written in C language with *libpcap* [10] library, *tcpdump* pre-filtering with *Python DPKT* [4] script and *tcpdump* pre-filtering with the same ad-hoc C language traffic dissector.

At first glance, we note that for some file sizes the performance of tcpdump pre-filtering and Python is worse compared to the ad-hoc. As it turns out, the performance depends on the number of DNS packets rather than the file size. Furthermore, the obtained results are close in most cases. However, in those cases where the number of DNS packets is too high, namely with file sizes of 6 and 7 Gbytes, the performance of python pre-filtered solution is significantly worse than its homologous in C and even worse than the ad-hoc C dissector.

The main reason for this loss of performance is due to, as we discussed in III-B, all *Python* libraries evaluated waste large

amounts of time with respect to C applications, to classify each packet individually within their internal lists of protocols. Therefore, the number of DNS packets into PCAP traces will affect the number of packet classifications and search times within protocol lists and thus, the performance variations respect to C dissectors.

V. Conclusions

In conclusion, the usage of third-party libraries written in *Python* have some advantages in terms of readability and fast prototyping time with respect to low-level solutions written in C and *libpcap*. However, the main drawback for these solutions is performance and the best choice are pre-filtered solutions with *Tcpdump* or similar pre-filtering tools to obtain good performance results. As a future work, we plan to extend the assessment procedure to newer libraries and follow the evolution that the well-established libraries which have been studied in this paper.

References

[1] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *ACM SIGCOMM, 2010*.
[2] Python Website. http://python.org
[3] Google Project Hostings. http://code.google.com
[4] DPKT Website. http://code.google.com/p/dpkt/
[5] PCAPY Website. http://oss.coresecurity.com/projects/pcapy.html
[6] Impacket Website. http://oss.coresecurity.com/projects/impacket.html
[7] pylibpcap Website. http://pylibpcap.sourceforge.net/
[8] pycap Website. http://pycap.sourceforge.net/
[9] SCAPY Website. http://www.secdev.org/projects/scapy/
[10] Tcpdump and libpcap Website. http://www.tcpdump.org
[11] Mergecap Website. http://www.ethereal.com/docs/man-pages/mergecap.1.html
[12] Ethereal Website. http://www.ethereal.com
[13] Editcap Website. http://www.wireshark.org/docs/man-pages/editcap.html
[14] Wireshark Website. http://www.wireshark.org
[15] P. Mockapetris. RFC 1034. http://www.ietf.org/rfc/rfc1034.txt
[16] P. Mockapetris. RFC 1035. http://www.ietf.org/rfc/rfc1035.txt
[17] J. Kowall, D. Curtis. Vendor Landscape for Application-Aware Network Performance Monitoring and Network Packet Brokers. http://www.gartner.com/resId=1987715

Javier Aracil received the M.Sc. and Ph.D. degrees (Honors) from Technical University of Madrid in 1993 and 1995, both in Telecommunications Engineering. In 1995 he was awarded with a Fulbright scholarship and was appointed as a Postdoctoral Researcher of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. In 1998 he was a research scholar at the Center for Advanced Telecommunications, Systems and Services of The University of Texas at Dallas. He has been an associate professor for University of Cantabria and Public University of Navarra and he is currently a full professor at Universidad Autnoma de Madrid, Madrid, Spain. His research interest are in optical networks and performance evaluation of communication networks. He has authored more than 100 papers in international conferences and journals.

German Retamosa received the M.Sc. from Universidad Autonoma of Madrid in 2009 in Computer Science and Telecommunications Engineering. As a serial entrepreneur, he combined his computing science and telecommunications studies with R+D jobs at IBM Global Services, Movistar and as a cofounder and CEO of his own startup, Tink Security. He is currently R+D senior developer at Naudit High Performance Computing and a Ph.D. student at Universidad Autnoma de Madrid, Madrid, Spain. His research interest are in high performance computing systems on communication networks and information security threads evaluation.