Performance Evaluation of Python Tools to Capture Packets in Resource-Constrained Devices

Otávio O. Silva, Daniel M. Batista

[1]Department of Computer Science – Institute of Mathematics and Statistics
University of São Paulo (USP) – São Paulo, SP – Brazil

otavio.ols@usp.br, batista@ime.usp.br

Abstract. *Real-time traffic classifiers, such as Intrusion Detection Systems based on Online Learning, must be constantly fed with packets to classify the traffic by a specific deadline. When considering Python, there are several options for capturing packets. This paper evaluates the performance of three tools to capture packets in Python running on a Raspberry Pi 3 Model B. Exhaustive experiments conclude that Pypcap + dpkt is the best option over Pyshark and Scapy. For instance, in terms of average CPU usage, when capturing TCP traffic over cable, Pypcap + dpkt consumed 75.73% less than Pyshark and 4.24% less than Scapy. All the code used to run the experiments is shared as free and open-source software for reproducibility.*

1. Introduction

Despite being a packet-switched network, the modern Internet is powered by some mechanisms that require more information than a single packet to make the best decisions. For instance, an Intrusion Detection System may use the interval between several packets and the size of all these packets to classify traffic as benign or malicious [Kostas et al. 2025]. In this case, the packets must be captured and stored in memory for some time to allow good accuracy in classification. If the classification is made in real-time, for instance when online learning is employed, the capture must be efficient to avoid excessive delays, which will affect the performance of the mechanism [Oliveira et al. 2024].

Capturing the traffic in the default gateway of the network can be enough when making decisions related to the traffic going to or coming from the Internet. However, decisions that must consider the internal traffic of an organization would be negatively affected by this strategy. Moreover, capturing all the traffic in the default gateway could be demanding to the equipment, which will need to dedicate part of its resources to this end. An option would be to capture the traffic in the end nodes of the network. However, even considering that the traffic will be less intense than in the default gateway, the capacity of these devices can be low, mainly when considering a scenario of the Internet of Things with very simple devices like sensors and actuators.

With the advent of Artificial Intelligence techniques, such as Machine Learning, to help make decisions in computer networks, the Python language became a notable presence in new proposals that involve the processing of network datasets (basically CSVs generated from files captured by sniffers) [Afifi et al. 2024]. So, it is reasonable to consider the use of tools that can be imported in Python code to capture the traffic and send it directly to the part of the code responsible for training a model or making a specific

inference. Three tools that appear frequently in the literature for this purpose are Pyshark, Scapy, and Pypcap working together with dpkt.

Although there are proposals in the literature that make use of these tools [Dsouza et al. 2022] [Nazarov and Arslan 2022] [Sambath et al. 2024], the choice to do so is not based on a performance evaluation, or when it is, the conclusions presented by some of them are not observed in scenarios with resource-constrained devices or do not consider metrics other than time, such as CPU usage, memory usage, and packet loss in scenarios with varying configurations. This paper fills this gap by evaluating the performance of the three tools in scenarios with varying bit rates and packet lengths of traffic being captured in a Raspberry Pi 3 Model B. For instance, it was observed that in terms of average CPU usage, when capturing TCP traffic over cable, Pypcap + dpkt consumed 75.73% less than Pyshark and 4.24% less than Scapy (For the sake of reproducibility, all the software developed to conduct the experiments is available as free and open-source software at `https://github.com/otavioolsilva/wperformance-2025`). We expect that our findings will help those running data analytics in computer networks to select the best capturing tools to meet their demands.

The rest of this paper is organized as follows: Section 2 summarizes the three tools evaluated in this paper. Section 3 presents previous works from the literature that use the tools and highlight their gap, justifying the contribution of our work. Section 4 and Section 5 present the experiments carried out to evaluate the tools by varying the bit rates of the traffic and the packet lengths, respectively. The paper is finished with conclusions and future works in Section 5.

2. Tools for Packet Capture in Python

The experiments were conducted evaluating three different Python libraries capable of sniffing packets from the network interface: Pyshark[1], Scapy[2] and Pypcap[3] working together with dpkt[4]. These tools were chosen based on the literature and other related works in the area, as they are frequently used in Python context for sniffing and processing packets. Since some are more than libraries, the terms “tool” and “library” will be used interchangeably in this paper.

Pyshark is a wrapper for Tshark, the command-line version of Wireshark[5], one of the main tools used for packet sniffing. So, when using Pyshark, what is, in fact, being used is Tshark. A consequence of this is the fact that the Pyshark can take advantage of the various Wireshark dissectors[6], facilitating a deep analysis of the packets.

Scapy is a packet manipulation tool. It allows the creation/sending of packets and their decoding/capture, considering the particularities of diverse protocols. Besides being used as a Python library, Scapy can also be used interactively, which is helpful for prototype solutions before putting them into production.

[1] `https://github.com/KimiNewt/pyshark`
[2] `https://scapy.net/`
[3] `https://github.com/pynetwork/pypcap`
[4] `https://github.com/kbandla/dpkt`
[5] `https://www.wireshark.org/`
[6] `https://www.wireshark.org/docs/wsdg_html_chunked/ChapterDissection.html`

Pypcap is a wrapper for libpcap[7], a C/C++ library for network traffic capture and the basis for sniffers as tcpdump and Wireshark. Pypcap does not come with functionalities for packet parsing, making it necessary to use another independent tool for this purpose. For our work here, we chose to use the Python module dpkt, which is designed to be performant for this end.

3. Related Work

An anomaly-based IDS that can be powered by five different machine learning techniques is presented in [Dsouza et al. 2022]. Despite using the NSL-KDD dataset, which is based on a very dated dataset to train the IDS, the authors test the performance of classification of live traffic captured from the network interface of the Internet gateway, which is vital if considering putting the system in a real environment. For packet sniffing and feature extraction, the authors decided to use Pyshark, justifying that it has the best efficiency among other options, although they do not inform what these others were.

An in-network cache to reduce the impact of file retransmissions in case of transfer errors is proposed in [Nazarov and Arslan 2022]. The idea is to take advantage of the capabilities of programmable switches and resend the files to the hosts from a nearby location instead of the original file source. This proposal depends on efficient options to parse the captured traffic before reconstructing the files in the cache points of the network. To this end, the authors evaluated the same three options we assessed and concluded that dpkt was the best option. The only criterion used by the authors was the time to parse the captured traffic.

A network traffic analyzer is proposed in [Sambath et al. 2024]. The software is written in Python and the authors used two of the tools evaluated by us: dpkt and Scapy. However, the paper lacks a performance evaluation of them and a recommendation of when each one must be used.

Different from [Dsouza et al. 2022], we do not prematurely conclude that Pyshark is the best option and, as will be shown in the next sections, it was the least efficient. Different from [Nazarov and Arslan 2022], we evaluated the tools considering various metrics, not only time, and when capturing the traffic, not only when parsing it. Finally, different from [Sambath et al. 2024], we conduct an exhaustive performance evaluation to compare three options for capturing traffic from inside Python code.

4. Bit Rate Experiments

To understand what we can expect from an IoT device in terms of computing performance, which is usually more limited compared to a usual computer, in this first set of experiments we evaluate the tools under diverse bit rates of network traffic incoming to the device. The next subsections explain the design of experiments and present the results.

4.1. Design of Experiments

Each round of an experiment is conducted as follows, orchestrated by a Bash script:

1. Run the Python script to capture the traffic in background and wait 10 seconds;
2. Start a transmission of packets to simulate network live traffic for 10 seconds; *and*

[7] `https://www.tcpdump.org/`

3. After the artificial traffic is finished, wait 32 seconds and collect the results.

Regarding Step #1, the script is engineered to call each of the different tools separately. Each one of them is configured to sniff the network for 50 seconds. Regarding Step #2, iperf3[8] was used. It can transmit packets in the client-server model, allowing the adjustment of several parameters. Regarding Step #3, it is important to note that the Pypcap library does not implement a timeout mechanism that stops its execution after a given amount of time. To achieve this behavior and limit its execution to 50 seconds as designed, after the last 30 seconds (plus 2 seconds to have a safety margin) the Bash script performs a single ping to the server to generate a packet that will be out of the window of time defined manually in the code, so the loop breaks when the sniffer catches it.

Each combination of bit rate and tool was executed for five rounds, a sufficient number to observe a low standard deviation in the results. The average of the values collected is reported, except for the memory use metric, in which its peak was considered. To register each metric, we used:

- The psutil Python library to measure the CPU consumption. It can retrieve information about the system performance. For our purpose, we collected the percentage of CPU used by a process. Its operation is similar to the ps command in the shell: it can be used to obtain the CPU utilization as a percentage for a process since its last call using the total CPU time consumed. Note that this value can be greater than 100 if the process runs on more than one processor core.
- The "resource" Python module to measure the memory use. It can collect the maximum resident set size used by the process in KBytes.
- The "time" command to measure the total time utilized by the process, to ensure that the workflow defined above was being followed.
- The "number of packets counted" is a value reported by each Python script at the end of its execution. The different sniffers read the packets in the network and the only processing that is done is to verify if the transmission protocol used is UDP or TCP, to perform a minimal packet parsing, which in a real scenario would probably be more complex and deeper in the packet.

The scenarios considered for analysis are:

- Transmission of TCP packets without bit rate limit, with the Raspberry Pi being the client and the server at the same time in the localhost interface;
- Transmission of UDP packets at the bit rates of 1Mb/s, 2Mb/s, 3Mb/s, 4Mb/s, 5Mb/s and 10Mb/s with the Raspberry Pi being the client and the server at the same time in the localhost interface;
- Transmission of UDP packets without bit rate limit, with the Raspberry Pi being the client and the server at the same time in localhost; *and*
- All three previous scenarios, but with the Raspberry Pi being a client and a notebook being a server, on a direct Ethernet cable connection without Internet access.

To have an idea about the traffic transited in the experiments, Table 1 shows the values reported by iperf3 (second column) for each configuration (first column), with the localhost scenarios indicated as "LH" and the remote server ones as "RS". The values are

[8] `https://iperf.fr/`

Configuration	Report from iperf3
LH TCP (without limit)	3.53 Gb/s, 4.11 GBytes
LH UDP 1Mb/s	1.02 Mb/s, 1.22 MBytes in 39 packets
LH UDP 2Mb/s	2.02 Mb/s, 2.41 MBytes in 77 packets
LH UDP 3Mb/s	3.01 Mb/s, 3.59 MBytes in 115 packets
LH UDP 4Mb/s	4.01 Mb/s, 4.78 MBytes in 153 packets
LH UDP 5Mb/s	5.01 Mb/s, 5.97 MBytes in 191 packets
LH UDP 10Mb/s	10.00 Mb/s, 11.9 MBytes in 382 packets
LH UDP (without limit)	7.21 Gb/s, 8.4 GBytes in 275246 packets
RS TCP (without limit)	94.64 Mb/s, 113 MBytes
RS UDP 1Mb/s	1 Mb/s, 1.19 MBytes in 864 packets
RS UDP 2Mb/s	2.00 Mb/s, 2.38 MBytes in 1727 packets
RS UDP 3Mb/s	3.00 Mb/s, 3.58 MBytes in 2590 packets
RS UDP 4Mb/s	4.00 Mb/s, 4.77 MBytes in 3453 packets
RS UDP 5Mb/s	5.00 Mb/s, 5.96 MBytes in 4316 packets
RS UDP 10Mb/s	10.0 Mb/s, 11.9 MBytes in 8632 packets
RS UDP (without limit)	95.7 Mb/s, 114 MBytes in 82620 packets

Table 1. Parameters reported by iperf3 on each configuration for the Pyshark experiments.

the averages of the five runs. Since we are considering the capture in a device which can be used in IoT deployments, these bit rates are acceptable [Sivanathan et al. 2017]. Due to the page limit, we only show the measurements obtained when running Pyshark (in fact, the results of the scenarios with limited bit rates were the same for all the tools).

All metrics were collected in a Raspberry Pi 3 Model B, equipped with a Quad Core 1.2GHz Broadcom BCM2837 64-bit ARM CPU, 1GB RAM running Debian GNU/Linux 12 (bookworm), with a network interface of 100 Mb/s. An Acer Aspire 3 A315-41-R4RB notebook was used as the server in the experiments with the Ethernet connection, equipped with an AMD Ryzen 5 2500U 2.0GHz 64-bit x86 CPU, 12GB DDR4 2667 MHz RAM running Fedora Linux 41 (Silverblue), with a network interface of 1000 Mb/s.

4.2. Results

One first observation to evaluate the performance is the difference between the minimum and maximum values reached by each of the Python libraries in the CPU consumption over the configurations, as can be seen in Figure 1 (All the graphs in the paper have indications of the standard deviation as a metric of dispersion, represented as a black line on the top of the bars). At some point of the experiments, we can see that the libraries begin to present constant use of the processing power: for Pyshark this happens very fast, with about 4Mb/s in UDP localhost and 2Mb/s in UDP remote server, reaching approximately 80% of use; for Scapy this happens at 4Mb/s in UDP remote server, reaching around 20% of use; and for Pypcap together with dpkt this threshold is uncertain in the considered configurations, but we can see that in the unlimited bit rate cases the use of CPU is also close to 20%.

We can observe that the minimum use of CPU for Pyshark, in the cases of 1Mb/s

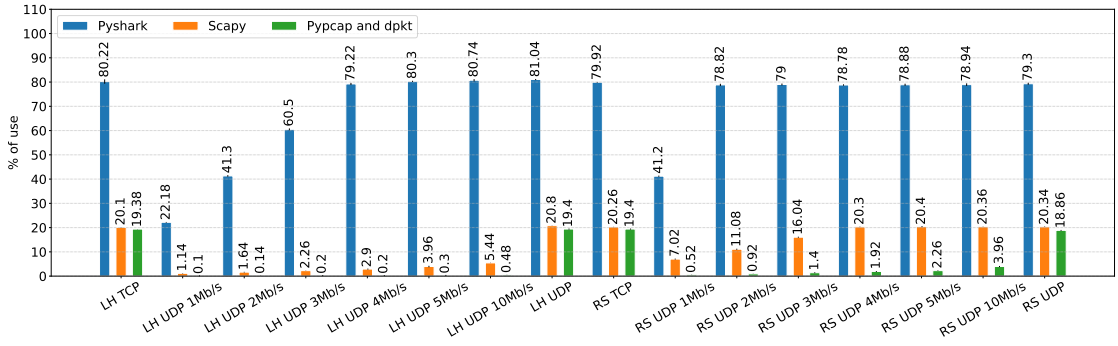

Figure 1. Average CPU usage as a function of protocol and bit rate.

bit rate, is greater than the maximum achieved by the other two libraries. Pyshark is a great library to extract processed information from the packets, however has demonstrated that it is not a good choice for handling high volumes of traffic, especially when processing power is a scarce resource.

Scapy had similar CPU usage compared with Pypcap and dpkt in the scenarios with no bit rate limits, but the key difference is that the pair of libraries reaches their peak more slowly. In a conventional network, we do not expect each device to be constantly dealing with high packet volumes, the usual is for traffic to be controlled. So, the difference in performance dealing with a low amount of traffic is interesting to observe, as this will probably be the main scenario that an IoT device will be working with. Also note that the growth of CPU use appears to be well-behaved: the standard deviation is not expressive over the different configurations, and comparing cases where the number of packets increases linearly (for example, from 2Mb/s to 4Mb/s), the CPU usage also increases in a nearly linear proportion (in RS, from 11.08% to 20.30%).

Increasing the bit rate of the traffic, and consequently the number of packets, beyond the observed thresholds does not lead to significant differences, indicating that the number of packets in the network is a very influential parameter to the performance of the libraries, but only to a certain point. After it, the libraries start struggling to deal with the volume of packets. However, this is not the only parameter that affects the observations: the size of the packets seems to be also an influential factor. In the localhost scenarios, the number of packets per configuration is lower compared with the remote server one, as can be seen in the Table 1, but the total number of bytes transmitted is similar, so the size of the packets in the localhost scenario is larger. Scapy and Pypcap + dpkt seem to present more difficulty when dealing with the traffic when the number of packets increases, but for Pyshark the size of packets is already relevant, as we can see in the localhost metrics. This parameter will be explored more in Section 5.

The memory use, in contrast with the CPU, does not present high variations between the configurations. The greatest difference can be observed when comparing localhost and remote server scenarios, as can be observed in Figure 2. This is justified: we are measuring the peak of memory and, as we discussed in the previous paragraph, the localhost scenarios have larger packets than the remote server ones, so the parsing of a single packet will take more memory in the first case. As the packets have a constant size in the same configuration, it is reasonable to observe constant values over the experiments. The

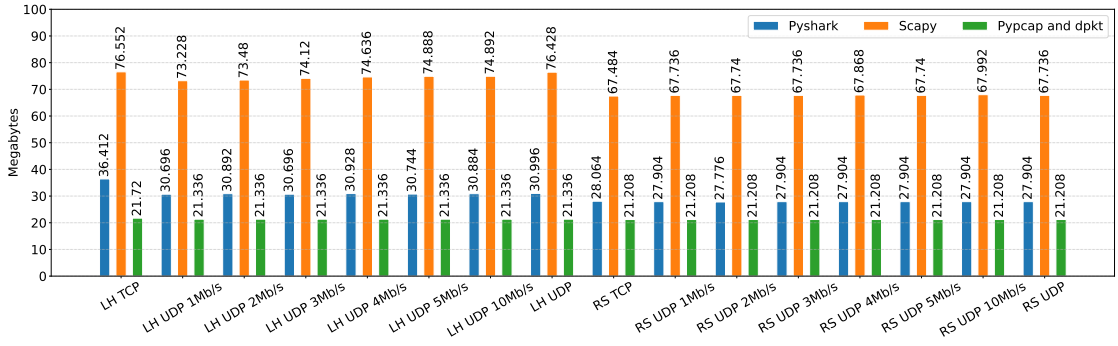

Figure 2. Peak of memory as a function of protocol and bit rate.

Scapy memory usage is considerably above the other libraries, using more than twice the amount required by Pyshark and even more when compared to Pypcap + dpkt. The absolute value of memory use is not very elevated, but considering the context of IoT devices, a difference of bytes can have a big impact.

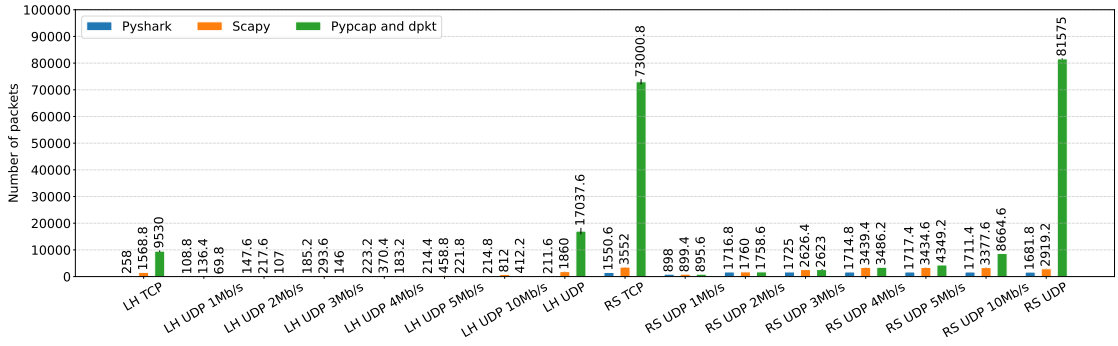

Figure 3. Average number of packets as a function of protocol and bit rate.

Considering the total number of packets processed by each library, the first observation in Figure 3 is the disparity between Pypcap + dpkt and the other ones in the scenarios without a bit rate limit. It is hard to compare them as the data of this pair of libraries is on another scale, achieving numbers much more significant than the other two. To reduce this disparity, Figure 4 represents only the scenarios with limited bit rate.

One may notice that the number of packets counted can be bigger than the number of packets reported in the reference Table 1. This happens because the numbers reported in the table are only referent to the UDP flow generated during the execution of iperf3. The application also establishes a control connection using TCP between the client and server to exchange the current parameters, so it is natural to have a greater final number of packets. However, note that in the majority of the cases, the values reported are lower than expected, especially in the unlimited bit rate cases, indicating a loss of packets.

The remote server results contain valuable insights: we can see that the bars from the three different libraries grow together, until Pyshark and Scapy stop improving. As the number of packets in the traffic grows, the performance of the libraries is degraded, and in the same way that happened with the CPU usage, they do not improve anymore after a threshold. The localhost scenarios presented unexpected behaviors on this metric,

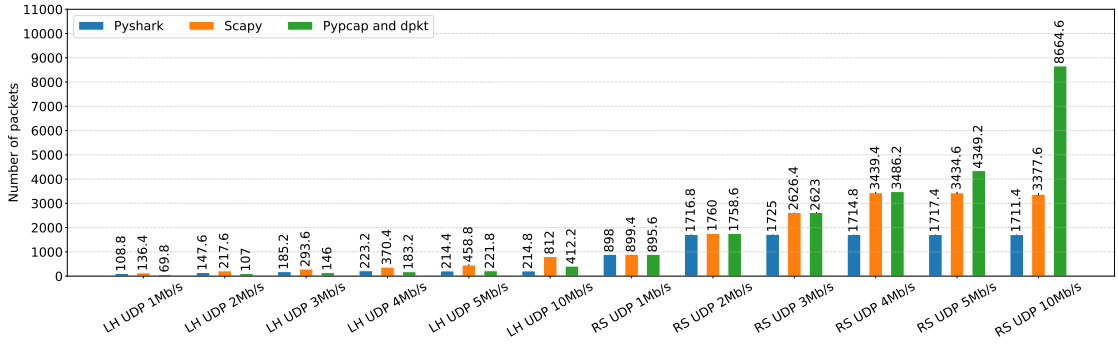

Figure 4. Average number of packets as a function of protocol and bit rate (only UDP scenarios with limited bit rates to improve visualization).

not being a good reference for conclusions:

- Scapy is counting all the packets twice, considering the number of packets counted by Pypcap + dpkt as reference. The loopback interface is very different from the normal ones, and Scapy's FAQ[9] indicates that unexpected behaviors can be observed using it.
- The number of packets registered by Pyshark is also above the number counted by Pypcap + dpkt, and half the number of Scapy. The reason for this is a Tshark plugin that searches for connections with Android devices on localhost before the sniff starts[10]. In this way, every run in localhost with Pyshark has about 40 packets more than the other libraries.

5. Packet Length Experiments

The second set of experiments evaluates the tools under flows with diverse packet lengths, to understand if this is an influential factor in the performance of the Raspberry Pi running the Python libraries that are being studied. The next subsections explain the design of experiments and present the results.

5.1. Design of Experiments

Different from the bit rate experiments (Subsection 4.1) in which the duration of the packet transmission was fixed, in the packet length experiments the constant over the configurations is only the bit rate at 2Mb/s. In this sense, the flow duration generated by iperf3 varies from one scenario to another. Moreover, the connectivity was done via Ethernet and the protocol was only UDP, without the localhost scenario. With this configuration, all the libraries can still perform well and also allow a considerable number of packets in a short time. The procedure orchestrated by a Bash script is as follows:

1. Run the Python script to capture the traffic and wait 10 seconds;
2. Start a transmission of packets in the local network to simulate live traffic; *and*
3. After 42 seconds, end the test and collect the results. Here, in the same way as in the last experiments, a ping is done by the Bash script to break the pypcap loop.

[9] https://scapy.readthedocs.io/en/latest/troubleshooting.html#i-can-t-ping-127-0-0-1-or-1-scapy-does-not-work-with-127-0-0-1-or-1-on-the-loopback-interface

[10] https://github.com/gcla/termshark/issues/98

This is the reason for the difference of 2 seconds between the total time spent by the Bash script and the Python script.

Similar to the last experiments, the sniffers are configured to run for 50 seconds. However, the total packet transmission duration is not defined here. The scenarios considered for analysis were previously tested to ensure they would be inside the time range of the sniffers and to permit a gap at the end of the iperf3 execution. They are:

- 1000 packets with 148, 724 and 1448 bytes of length;
- 4000 packets with 148, 724 and 1448 bytes.

The first configurations with 1000 packets were chosen to use a volume of packets that all the libraries would be capable of handling, considering the experiments of the previous section. For the configurations with 4000 packets, this number was chosen to understand whether a reduction in the length of the packets would improve the results we also observed in the sections above, where we saw that Pyshark and Scapy never reached a count above this. Similar to the bit rate experiments, five rounds were executed for each combination of number of packets, packet length and tool, usually taking the average value of the numbers observed, except for the memory use metric, in which the peak of use is considered. The same tools used there to register the metrics were used here.

The MTU (Maximum Transmission Unit) of the Raspberry Ethernet interface is 1500 bytes, being the TCP MSS (Maximum Segment Size) of 1448 bytes, the default length of a packet used by iperf3 for both UDP and TCP transmissions. Using this value as a basis, two others were chosen for the comparison: 148 bytes (close to 10% of 1448) and 724 bytes (50% of 1448). All the experiments were performed using the same equipment and Ethernet connection from the previous section.

5.2. Results

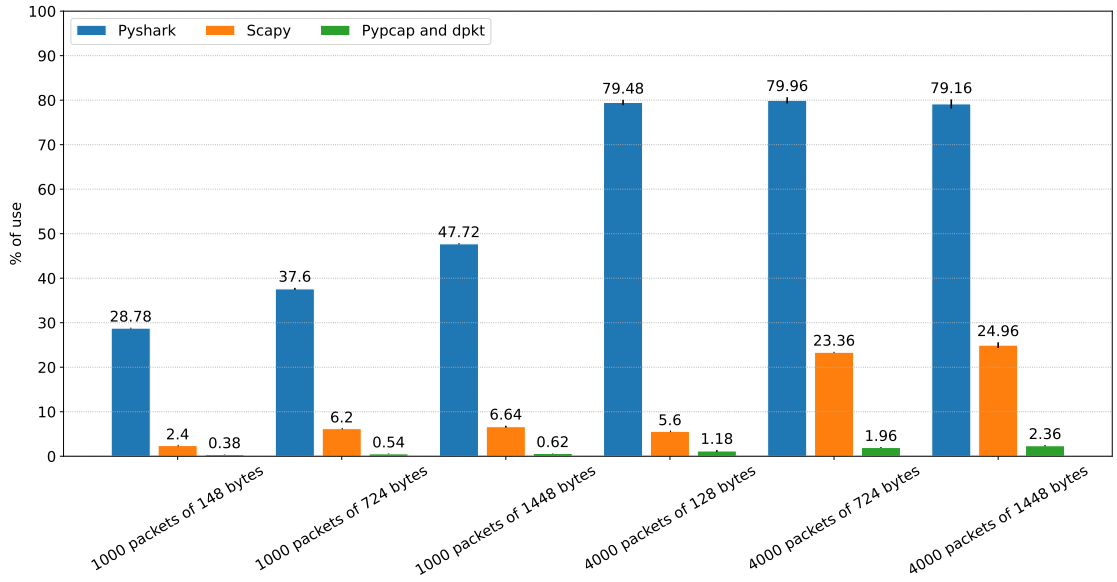

Figure 5. Average CPU usage as a function of number of packets and packet length.

Observing the CPU metric in Figure 5, it is evident that all the libraries have an increase in CPU usage when the packets are larger. Pyshark, being coherent with what we saw in the previous section, utilizes much more CPU than the other libraries. The Python script runs for a total of 50 seconds and the flow generated by iperf3 for the first scenario considered has only 0.59 seconds of duration, representing approximately 1.2% of the total time, while the usage of CPU time by Pyshark was of 28.78% on average, that is, this library had a much higher CPU time consumption than expected considering the duration of the experiment. In the configurations with 4000 packets, we can see that the library stays at the same level. This observation is interesting to be seen together with the number of packets counted in Figure 7.

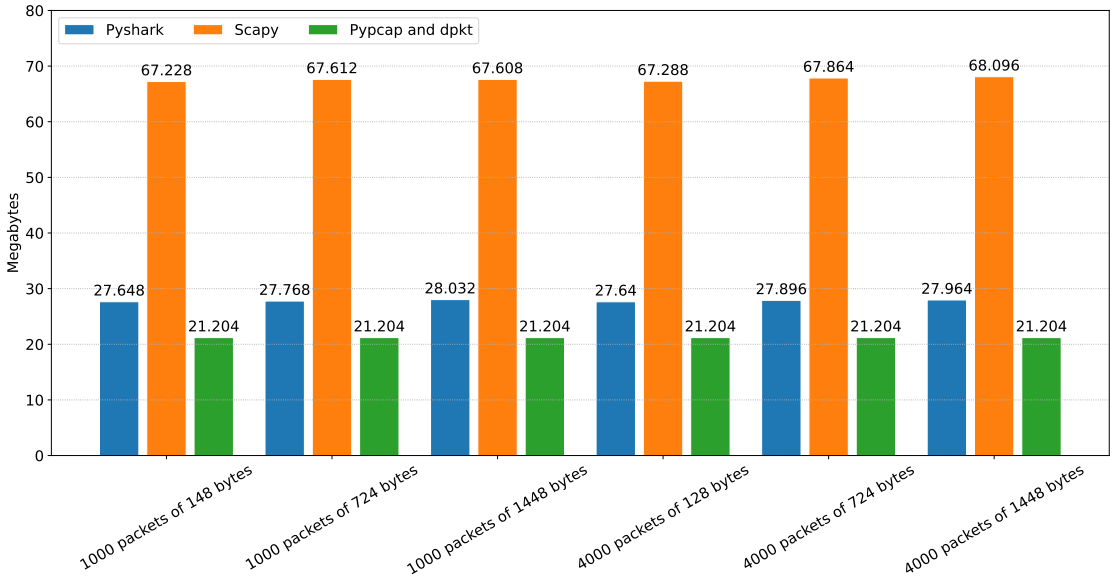

Figure 6. Peak of memory as a function of number of packets and packet length.

By analyzing Figure 6, the three libraries were coherent with what we saw in the bit rate experiments. What is not coherent is the variations between the configurations: neither of the libraries seems to respect the increase of packet length in their reports of memory consumption. Pyshark and Scapy had variations that are not proportional to the difference in length, and the duo Pypcap + dpkt was constant over all the configurations. The scale of the packet sizes considered and total memory use is different, but this is perhaps an indication that the structures used are not dynamic in terms of packet size.

In the 4000 packets scenarios, Pyshark counted fewer packets than the other libraries in most cases, as indicated in Figure 7, even with high CPU consumption. However, different from the plateau observed in Figure 5, it is clear that this library is being affected by the size of the packets: the CPU usage is the same over all the packet lengths considered, but the number counted decreases as there is an increase in the packet size. Even though Pyshark could not count the totality of data in the network traffic over the different packet lengths in the 4000 packets scenario, it struggled more in the ones with bigger packets. On the other hand, in the 1000 packets scenario, it counted all the packets.

Scapy, in contrast, even in the 1000 packets scenario, was not capable of counting all the packets, and note that the standard deviation is not very significant. This may

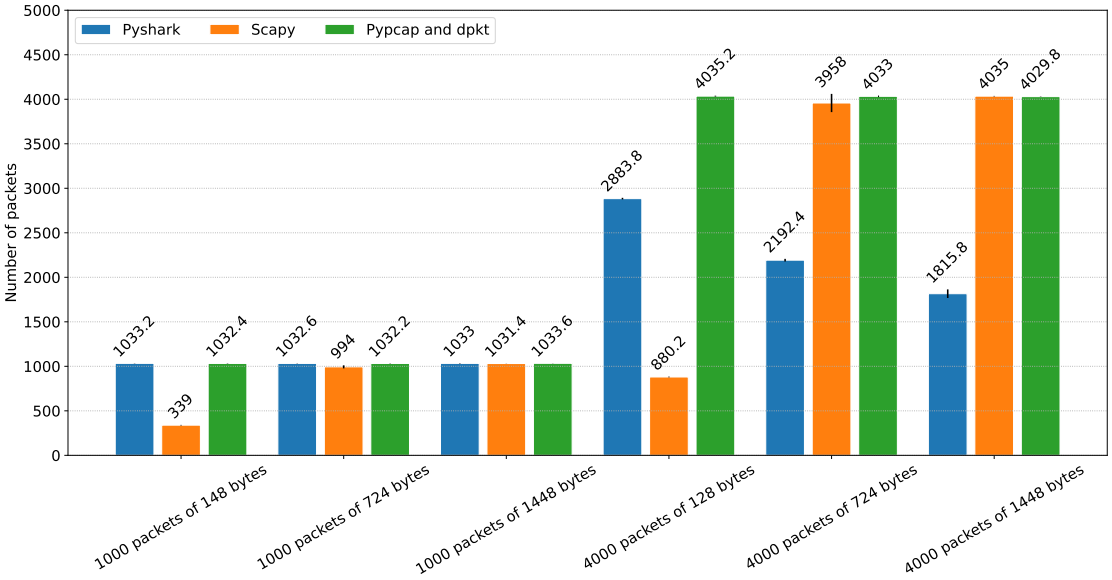

Figure 7. Average number of packets as a function of number of packets and packet length.

indicate that this library has a limitation in the number of packets it can read per unit of time (none of the libraries can read unlimited packets per second, however, we may have found this limit here). In the bit rate experiments we observed that the limit reached by this library in the number of packets counted was close to 3500 during the 10 seconds of flow, so it is interesting to see here that this library was able to count only 339 packets in the experiment with 0.59 seconds (1000 packets of 148 bytes) and 994 packets in the one with 2.89 seconds (1000 packets of 724 bytes). The numbers are not perfectly proportional, but are close: assuming that this library can read 350 packets per second, in 0.59 and 2.89 seconds we would expect 206.5 and 1011.5 packets, respectively. The bit rate being fixed leads to differences in the number of packets per second, which might have an influence here. Even more interesting to see is that in the experiments that lasted more than 10 seconds (4000 packets of 724 and 1448 bytes), Scapy was able to read more than the usual 3500 packets, counting 3958 packets on average in the one with 11.58 seconds (the standard deviation is more expressive here. During the five repetitions the number of packets varied between 3775 and 4006), in which we would expect 4053 packets considering 350 packets per second.

The CPU usage by Scapy also seems to reflect the behavior described in the previous paragraph: if we compare the consumption between the configurations with 1000 packets of 724 bytes and the one with 1448 bytes, we can see that the difference is small (6.20% and 6.64%, respectively), as is the difference in the counted number, but the duration of the experiments is much more significant (2.89 and 5.79 seconds). The use of CPU in the scenario with 4000 packets of 128 bytes, which lasted 2.37 seconds, also reinforces it: 5.60%, but the number jumps in the other 4000 packets experiments that have a longer duration. Therefore, it seems that the most influential factor here is not the packet length, but the number of packets per second.

Finally, Pypcap together with dpkt presented good results, with low CPU con-

sumption and counting all the packets. Note that the most stressful configuration (4000 packets of 1448 packets) had a flow with a duration of 23.16 seconds, which is approximately 46% of the total 50 seconds, but this duo of libraries presented only 2.36% on CPU consumption. In this way, as the results were excellent, it is not conclusive that the packet length had a significant influence on the performance of these libraries.

6. Conclusions and Future Works

The high number of proposals that keep emerging using Artificial Intelligence to classify network traffic has led to the increasing use of Python scripts to take care of all the training and testing phases of these proposals. This is due to the fact that Python has a good set of libraries to apply both traditional and recent learning techniques. In this context, capturing the traffic in the same script appears as a good strategy, mainly when dealing with real-time traffic classification. This paper evaluated three tools to capture and process traffic in Python: Pyshark, Scapy, and the duo Pypcap + dpkt. According to the evaluation, Pypcap acting together with dpkt is the best solution both in terms of CPU usage, memory usage and number of captured packets when running in a resource-constrained device. Ideas of future work include the evaluation of Pypcap + dpkt under volumetric attacks and acting as the first step of an Intrusion Detection System based on Online Learning.

Acknowledgments

This research is part of the STARLING project funded by FAPESP proc. 2021/06995-0. It is also part of the FAPESP proc. 2024/10240-3.

References

Afifi, H., Pochaba, S., Boltres, A., Laniewski, D., Haberer, J., Paeleke, L., Poorzare, R., Stolpmann, D., Wehner, N., Redder, A., Samikwa, E., and Seufert, M. (2024). Machine Learning With Computer Networks: Techniques, Datasets, and Models. *IEEE Access*, 12:54673–54720.

Dsouza, A., Lanjewar, V., Mahakal, A., and Khachane, S. (2022). Real Time Network Intrusion Detection using Machine Learning Technique. In *Proc. of the IEEE PuneCon*, pages 1–5.

Kostas, K., Just, M., and Lones, M. A. (2025). Individual Packet Features are a Risk to Model Generalization in ML-Based Intrusion Detection. *IEEE Networking Letters*, 7(1):66–70.

Nazarov, N. and Arslan, E. (2022). In-Network Caching Assisted Error Recovery For File Transfers. In *Proc. of the IEEE/ACM INDIS*, pages 20–24.

Oliveira, R., Pedrosa, T., Rufino, J., and Lopes, R. P. (2024). Parameterization and Performance Analysis of a Scalable, near Real-Time Packet Capturing Platform. *Systems*, 12(4).

Sambath, S, B. B., Mario, C., Maheswari, G., and Gunasekar (2024). Network Traffic Analyzer Using Python. In *Proc. of the 1st ICSCAI*, pages 1–7.

Sivanathan, A., Sherratt, D., Gharakheili, H. H., Radford, A., Wijenayake, C., Vishwanath, A., and Sivaraman, V. (2017). Characterizing and Classifying IoT Traffic in Smart Cities and Campuses. In *Proc. of the IEEE INFOCOM WKSHPS*, pages 559–564.