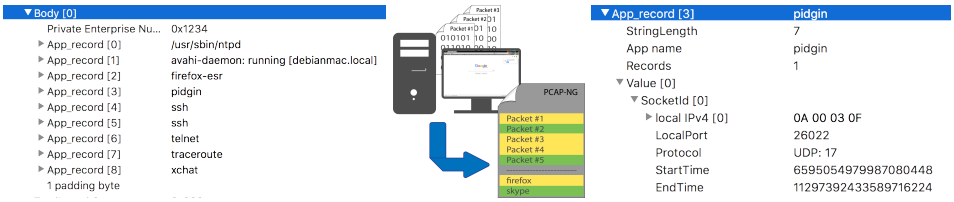
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2017
http://excel.fit.vutbr.cz

Network Traffic Capture with Application Tags

Jozef Zuzelka*



Abstract

Network traffic capture and analysis are useful in case we are looking for problems in our network, or when we want to know more about applications and their network communication. This paper aims on the process of network applications identification that runs on the local host and associates them with captured packets. The goal of this project is to design a multi-platform application that captures network traffic and extends the capture file with application tags. Operations that can be done independently are parallelized to speed up packet processing and reduce packet loss. An application is being determined for every (both incoming and outgoing) packet. All identified applications are stored in an application cache with information about its sockets to save time and not to search for already known applications. It's important to update the cache periodically because an application in the cache may close a connection at any time. Finally, gathered information is saved to the end of pcap-ng file in special structure as the separate pcap-ng block.

Keywords: Network Traffic Capture — Network sniffing — Network Application Identification

Supplementary Material: Example output file — Hexinator grammar

*xzuzel00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

An ordinary capture file usually contains a communication of multiple network applications. This project started as a solution to a difficult analysis of network communication of just one selected application. Stored information about applications and their communication can help if we are interested just in single application, so we do not have to process every packet in the capture file. Another usage of this information is to compute statistics which applications transferred most of the data. Further, based on an application information and servers it communicates with, we can specify only one application, we are interested in and capture just its packets in real-time. This information can also be used in firewalls, e.g. to deny a network connection for a particular application, although this is not the goal of this project. We can also identify a malicious software which resides in our computer using either its name or servers it communicates with. Furthermore, the stored communication can be used in network forensics analysis.

The main part of the tool is an identification of application which captured packet was destined for or which application generated it. The application is identified using netflow information and information provided by the operating system. The goal is to make the application multi-platform and usable on 1 Gb/s networks with minimal packet loss. In order to save captured traffic and extend it with custom information in the same file, pcap-ng file format is used. This format is open-source, well documented and widely supported.

Existing tools either are not multi-platform or they

have just part of required functionality. Ntopng[1] is a network traffic probe that does high-speed web-based traffic analysis and flow collection but doesn't support export of the captured traffic to a file. This product is also licensed per system and distributed only in binary[1].

Next tool, which shows applications and their connections is `lsof`. It is a command-line utility on unix-like systems which lists open files. It can print open UDP and TCP sockets and applications which opened them. But it prints this information just once after its run and it doesn't capture traffic.

Popular network sniffers like *Wireshark*[2] and *tcpdump*[3] offer network capture, but neither extends captured traffic with information about communicating applications.

The most similar application with the functionality we want is *Microsoft Network Monitor*. It shows captured traffic per application and it can save results for later processing. Two main limitations of this tool are its dependency on Windows platform and that it uses its capture file format which is undocumented and not so supported.

In our solution, after capture starts, network traffic is saved continuously into the output pcap-ng file, and when capture is stopped, this file is extended with a custom block containing communicating applications and identification of their sockets. Writing to the output file can be done independently as we don't need the whole packet for later processing. Needed information is saved in a structure which is later saved in the cache and used to identify source application. To handle sudden peaks in network traffic, the tool uses ring buffers between the main thread and both thread for writing to the output file and thread which searches in the cache. When traffic capture is stopped, cache records are appended to the output file.

The tool is aimed to be multi-platform and to be able to process 1 Gbps links with minimal packet loss. Thanks to the use of pcap-ng file format, applications that support this format will be, after small modification, able to process also our block.

2. Network traffic capture

The most common application programming interface used for writing programs which use the Internet protocols are *sockets* (also called Berkeley sockets). The *socket* is one endpoint in two-way communication link between two programs on the network. Network traffic can be captured using RAW sockets, which works on the L3 layer of ISO/OSI model. This type of *sockets* is not suitable for network sniffers as there is removed ethernet header from the packet and on BSD, neither TCP nor UDP packets are received using this *socket* [2].

The second option is to capture on the link layer of the ISO/OSI model. In this case, the whole packet is passed to the application. Access to the link layer is available with most current operating systems. This allows programs such as `tcpdump` to be run on normal computer systems and capture packets without special hardware. In combination with an interface in promiscuous mode, this allows an application to capture all the packets received on the local interface despite the fact they weren't destined for this host. Different platforms use different methods how to access link layer, which can be unified using a *libpcap* library. Using this library, we can write portable code and use single API on different platforms to capture on link layer. Unfortunately, although the *libpcap* is quite efficient, it is not fast enough for 1 Gbps links. The bottleneck is kernel's TCP/IP stack which can handle only about 1 million packets per second [3]. To achieve higher throughput kernel bypass techniques, such as netmap[4] and PF_RING[5] can be used.

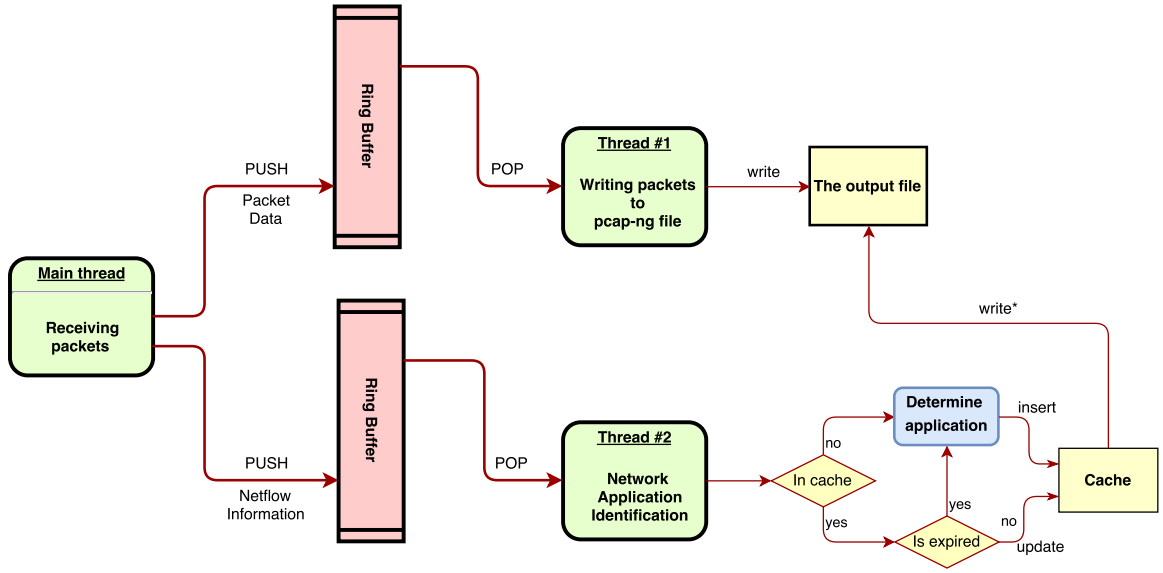
2.1 Capture speed up

`PF_RING` is a replacement for `PF_PACKET` that not only uses memory mapping instead of processing expensive buffer copies from kernel space to userspace, but it also uses ring buffers. It comprises of a kernel patch and a modified libpcap. This modified libpcap provides exactly the same API to the user but underneath it is using the ring buffers provided by the kernel patch to read packets. The patch copies the packet into the ring straight from the driver. `PF_RING` supports up to 10 Gbps packet capture with Intel cards by using a kernel module and modified NIC drivers [4]. Performance impact to the capturing speed is shown in Figure 7. Main disadvantage of `PF_RING` is that it supports only Linux.

2.2 Packet processing speed up

To be able to process more packets per second the application works in three threads. Their functionality is described in Figure 1. The main thread receives a packet, pushes it into ring buffers of the other two

[1] http://www.ntop.org/products/traffic-analysis/ntop/
[2] https://www.wireshark.org
[3] http://www.tcpdump.org
[4] http://info.iet.unipi.it/~luigi/netmap/
[5] http://www.ntop.org/products/packet-capture/pf_ring/



*Recognized applications and their records are saved to the output pcap-ng file as a separate part right after network traffic capture is stopped.

Figure 1. Threads and their roles

threads and notifies them about new packet in their buffer. Then it can receive another packet.

The second thread writes all packets from the ring buffer right into the output file in *Enhanced Packet Block*[6] format and then waits for notification about a new packet.

The third thread determines source application for every packet. It searches in the application cache which is shown in Figure 2. If an application is already in the cache and the record is not expired (will be explained later in this section), it updates time of the last packet in packet's netflow which belongs to the application. Otherwise, it tries to find an application, and in the case of success, the new application is inserted into the cache. Records in the cache are valid for a specific time period because an application can terminate its connection at any time. When a packet which belongs to an expired netflow is received, original application is determined again. The time period of records directly impacts effectivity of the application. A lower value means that network applications are identified with higher accuracy, but the application has to be determined more often which can increase packet loss. On the other hand, when records are valid for a long time, we can handle more packets per second, but cache can hold expired records.

[6]Type of PCAP-NG block that can be used to store a packet.

2.3 Application identification speed up

An application cache is used to speed up identification of network applications. After an application is successfully determined, it is inserted into the cache. As the application can close its socket at any time, it is important to update the cache regularly. This is realized using validity time, which is stored in every *TEntry* cache record. The validity time and operations with the cache are described in section 2.2.

When capture stops, the application fetches records from the cache and writes them into the output pcap-ng file. The cache improves the tool performance as the source application does not have to be determined for every packet when it is not needed.

The application cache consists of three levels – local port level, local IP address level and transport protocol level. Figure 2 shows an example of a cache structure. Search in the first level is based on a local port because it is least likely that two applications will have got the same local port. If two applications have the same local port, either a local IP address or a transport layer protocol must differ [5].

Mostly just one network interface is used to communicate over a network, so the second level compares transport layer protocol. Currently supported protocols are TCP, UDP and UDPLite.

If both the local port and the transport layer protocol are same, the local IP address is compared. This can occur if the host has more IP addresses set on one network interface. It could also happen if the host had more network interfaces but packets destined for other

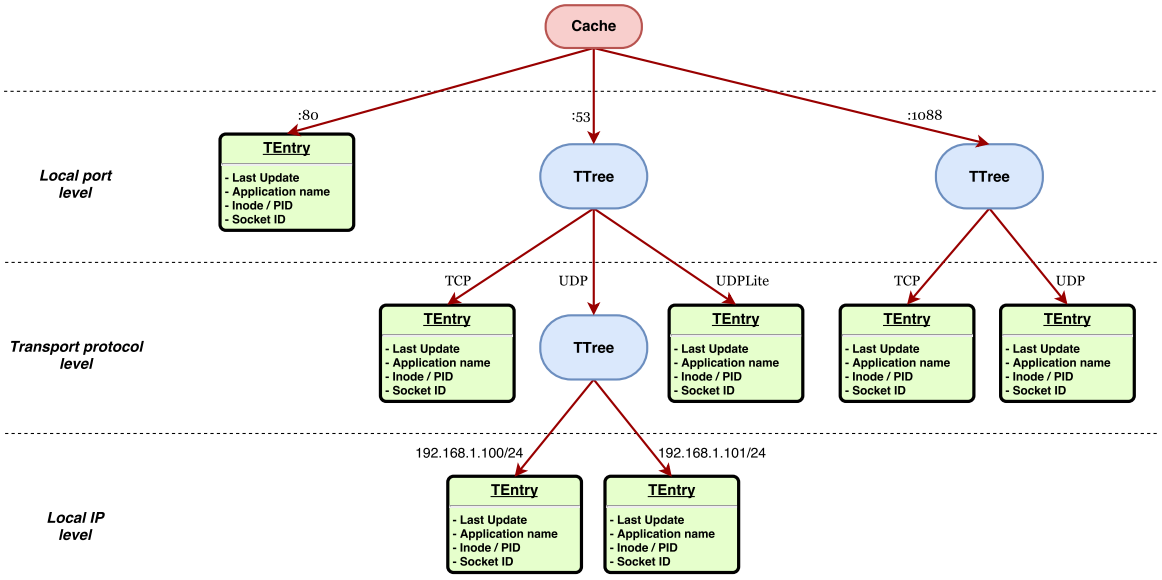


Figure 2. Application cache structure example

interfaces are not captured because capturing is not done in promiscuous mode.

3. Network applications identification

The aim of this project is to associate an application with its network traffic. Unfortunately, there is not a portable way how to implement it. Linux uses *procfs* file system and all important information is stored in this virtual file system. In Windows we can use *IP Helper API*[7] to retrieve information about connections and their PIDs, and then get process's command line from *Windows Management Instrumentation (WMI)*[8]. Following section describes in more detail just Linux platform, as other platforms haven't been fully explored yet.

3.1 GNU/Linux

Linux uses a virtual file-system called *procfs*. It is usually mounted in `/proc` and allows the kernel to export internal information to user-space in the form of files. The files don't actually exist on disk, but they can be read like other normal files. The default kernel that comes with most Linux distributions includes support for *procfs*.

Most networking features register one or more files in `/proc`. When a user reads the file, it causes the kernel to indirectly run a set of kernel functions that return some kind of output. The files registered by the networking code are located in `/proc/net` [6].

[7] https://msdn.microsoft.com/en-us/library/windows/desktop/aa366073(v=vs.85).aspx

[8] https://msdn.microsoft.com/en-us/library/aa384642(v=vs.85).aspx

3.1.1 List open sockets per PID

Information about sockets can be retrieved from `/proc/net` directory. It contains various net pseudo-files, all of which give the status of some part of the networking layer. These files contain ASCII structures, so they are easily readable. They contain information about open sockets and also their `inode` numbers. Once we have retrieved socket's `inode` number, we have to scan through all the processes to determine which process has an open file descriptor that points to the socket with this `inode` number. Open file descriptors per PID are stored in `/proc/[pid]/fd` folder.

3.1.2 Associate PID with the process name

The file `/proc/[pid]/cmdline` holds complete command line for the process. The command line arguments appear in this file as a set of strings separated by null bytes. The first argument is always the name of the application [7].

4. Output

Captured traffic is continuously saved into the pcap-ng file. When traffic capture is stopped, a special custom block is inserted to the end of the file as is shown in Figure 3.

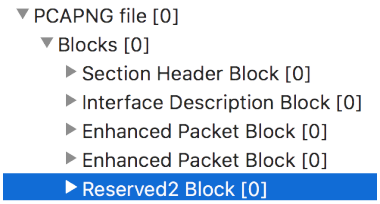


Figure 3. Structure of the pcap-ng file

A structure of the custom block with information

about recognized application is in Figure 4. This block contains application tags, which contain recognized application and identification of packets which belongs to the application.

▼ Reserved2 Block [0]	
BlockType	0x40000BAD
TotalLength	0xBDC
▼ Body [0]	
Private Enterprise Number	0x1234
▶ App_record [0]	firefox-esr
▶ App_record [1]	traceroute
3 padding bytes	
TotalLength2	0xBDC

Figure 4. Custom block structure

Specifically it consists of application records which contain application name, number of records for this application and records themselves (Figure 5).

▼ App_record [1]	traceroute
StringLength	19
App name	traceroute
Records	10
▼ Value [0]	
▶ SocketId [0]	
▶ SocketId [1]	
▶ SocketId [2]	
▶ SocketId [3]	
▶ SocketId [4]	
▶ SocketId [5]	
▶ SocketId [6]	
▶ SocketId [7]	
▶ SocketId [8]	
▶ SocketId [9]	

Figure 5. Application record structure

Each record identifies a group of packets which was sent using the same socket, thus from one application. The record consists of local IP address, local port, used transport protocol and time of the first and the last packet in the group. Based on these values we can uniquely identify socket in the capture file [5]. Exact structure is shown in Figure 6.

▼ App_record [1]	traceroute
StringLength	19
App name	traceroute
Records	10
▼ Value [0]	
▼ SocketId [0]	
▼ local IPv4 [0]	0A 00 03 0F
IPVersion	4
in_addr	0A 00 03 0F
LocalPort	8373
Protocol	UDP: 17
StartTime	7202951251095453696
EndTime	7202951251095453696
▶ SocketId [1]	
▶ SocketId [2]	

Figure 6. Socket identification structure

Other applications can still work with the pcap-ng file even if it contains our custom block. Applications that don't support our block will just ignore it [8].

Vendor of the custom block and its structure is identified using *Private Enterprise Number (PEN)*. Applications can recognize various types of custom blocks using this number. *Section Header Block* contains a name of user application which created the pcap-ng file and using this value, other applications will know whether the pcap-ng file contains inserted application records at the end of the file or not.

5. Tests

Implemented application was tested on MacBook Pro 13" (Early 2015) running Ubuntu 16.04 TLS. Figure 7 shows an amount of data which can be processed in real-time using standard `libpcap` and using `libpcap` with `PF_RING`. It is important to note that in this test, only one network application communicated. With more communicating applications, the tool has to actualize more cache records and it can make big difference in its performance.

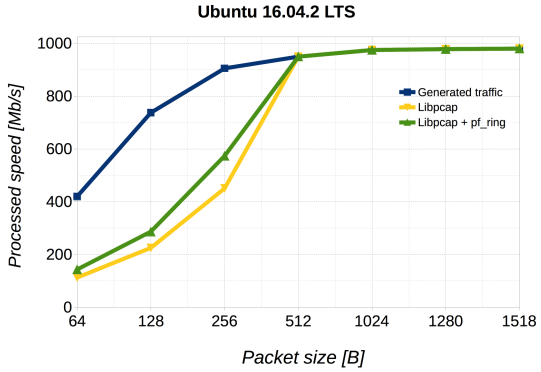


Figure 7. Network application identification on Linux

6. Conclusions

The paper describes the process of network application identification on Linux and methods how to speed up this process. Particular tasks which can be done independently are executed in threads. Gathered information is stored in memory, and after capture is stopped, it is appended to the end of the output pcap-ng file. The tool uses a cache to store determined network applications, so it has more time to identify newly opened sockets. Final pcap-ng block with results contains applications and identification of a group of packets which belongs to each application.

The only application with desired functionality is *Microsoft Network Monitor*, but it is only for Windows and uses undocumented Netmon capture file format. Our solution uses widely supported and open-source pcap-ng file format. It will be, unlike *Microsoft Network Monitor*, multi-platform.

Currently the application works on Windows and Linux for both IPv4 and IPv6 connections. Platform-dependent code is located in separated files and the right file is included during compilation, so it is easy to implement functionality for new platforms. A current limitation of the application is, it can handle traffic only around 100 Mb/s for in case of 64B packets. Although the main core of the application can process up to 800 Mb/s, searching in *procfs* on Linux takes too long. The tool also faces the problem that sometimes when it receives a packet and opens the *procfs* file to find application's socket, the socket is already closed, thus the application can't be determined.

In the future, the application will be implemented on other platforms, and new ways how to speed up application identification process on Linux will be explored.

Acknowledgements

I would like to thank my supervisor Ing. Jan Pluskal for many valuable suggestions and feedback.

References

[1] ntop. What is your software licensing model? Web page, October 2012. http://www.ntop.org/support/faq/what-is-your-software-licensing-model/.

[2] ithilgore. SOCK_RAW Demystified. http://sock-raw.org/papers/sock_raw. [Online; visited on 01/28/2017].

[3] Marek Majkowski. Kernel bypass. blogpost, September 2015. https://blog.cloudflare.com/kernel-bypass/.

[4] J. Gasparakis and J. Chapman. Improvement of libpcap for lossless packet capturing in linux using pf_ring kernel patch. Web page, October 2009. http://www.embedded.com/print/4008809.

[5] Mecki (http://stackoverflow.com/users/15809/mecki). Stackoverflow, January 2017. http://stackoverflow.com/a/14388707 (version: 2017-04-09).

[6] C. Benvenuti. *Understanding Linux Network Internals: Guided Tour to Networking on Linux*. O'Reilly Media, 2005.

[7] *proc(5) Linux Programmer's Manual*, September 2014.

[8] F. Risso, J. Bongertz, G. Combs, and G. Harris. Pcap next generation (pcapng) capture file format, April 2017. http://xml2rfc.tools.ietf.org/cgi-bin/xml2rfc.cgi?url=https://raw.githubusercontent.com/pcapng/pcapng/master/draft-tuexen-opsawg-pcapng.xml&modeAsFormat=html/ascii&type=ascii.