BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

NETWORK TRAFFIC CAPTURING WITH APPLICATION TAGS

NÁSTROJ PRO ZACHYCENÍ SÍŤOVÉHO PROVOZU S APLIKAČNÍM TAGEM

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **JOZEF ZUZELKA**
AUTOR PRÁCE

SUPERVISOR **Ing. JAN PLUSKAL**
VEDOUCÍ PRÁCE

BRNO 2017

Zadání bakalářské práce/20013/2016/xzuzel00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Zuzelka Jozef**
Obor: Informační technologie
Téma: **Nástroj pro zachycení síťového provozu s aplikačním tagem**
Network Traffic Capturing With Application Tags
Kategorie: Počítačové sítě

Pokyny:

1. Analyzujte možnosti zachycení sítového provozu na vybraných platformách po konzultaci s vedoucím.
2. Nastudujte formát Pcap-ng a navrhněte vhodné řešení pro jeho obohacení o aplikační tagy.
3. Implementujte nástroj zachycující síťovou komunikaci rozšířenou o apliakční tagy pro vybrané platformy.
4. Otestujte výkonost zachycení provozu z pohledu ztrátovosti na 1Gbit síti.

Literatura:

- Deri, L. (2005, May). nCap: Wire-speed packet capture and transmission. In *Workshop on End-to-End Monitoring Techniques and Services, 2005.* (pp. 47-55). IEEE.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Pluskal Jan, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

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 run on the local host and their associating 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. Records of 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 as a separate pcap-ng block.

Abstrakt

Zachytávanie sieťovej prevádzky a jej následná analýza sú užitočné v prípade, že hľadáme problémy v sieti alebo sa chceme dozvedieť viac o aplikáciach a ich sieťovej komunikácii. Táto práca sa zameriava na proces identifikácie sieťových aplikácií, ktoré sú spustené na lokálnom počítači a ich asociáci so zachytenými paketmi. Cieľom projektu je vytvorenie multi-platformového nástroja, ktorý zachytí sieťovú komunikáciu do súboru a pridá k nej aplikačné tagy, čo sú rozpoznané aplikácie a identifikácia ich paketov. Operácie, ktoré môžu byť vykonávané samostatne sú paralelizované pre zrýchlenie spracovania paketov, a teda aj zníženie ich strátovosti. Zdrojová aplikácia je zisťovaná pre všetky (prichádzajúce aj odchádzajúce) pakety. Všetky identifikované aplikácie sú uložené v aplikačnej cache spolu s informáciami o jej soketoch pre ušetrenie času nevyhľadávaním už zistených aplikácií. Je dôležité túto cache pravidelne aktualizovať, pretože komunikujúca aplikácia môže zatvoriť soket v ľubovoľnom čase. Nakoniec sú získané informácie vložené na koniec pcap-ng súboru ako samostatný pcap-ng blok.

Keywords

Network Traffic Capture, Network sniffing, Network Application Identification, Network traffic, Network monitoring

Klíčová slova

Zachytávanie sieťovej komunikácie, Sledovanie sieťovej prevádzky, Identifikácia sieťových aplikácií, Sieťová komunikácia, Monitorovanie siete

Reference

ZUZELKA, Jozef. *Network traffic capturing with application tags.* Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jan Pluskal

Rozšířený abstrakt

Tato bakalářská práce se zaměřuje na zachycení síťové komunikace, identifikování komunikujících aplikací a jejich asociaci se zachycenými pakety. Cílem práce bylo navrhnout a implementovat nástroj, který je multi-platformní a zvládne pracovat na 1 Gbps sítích. Na základě přidané informace o síťových aplikacích se mohou síťoví administrátoři zaměřit pouze na pakety vybraných aplikací. To umožní rychlejší a efektivnější práci se zachycenou komunikací například při zkoumání chování konkrétní aplikace v síti. Zjednoduší to práci také vývojářům s laděním jejich aplikací. Mezi názvy aplikací se může navíc vyskytovat známý nežádoucí software čím je snadno identifikovatelný.

Práce analyzuje způsoby zachycování na různých platformách a popisuje univerzální způsob jak zachytit síťovou komunikaci. Dále popisuje knihovny, které mohou být použity k urychlení zachycování paketů a principy, které jsou v těchto knihovnách využívány. Popsané knihovny avšak nemusí být použity aplikacemi pouze k urychlení zachycování, ale také k urychlení obyčejné síťové komunikace. Z důvodu, že se mi `PFQ` framework nepodařilo přeložit a s `netmap` API se seznámit v dostatečné míře, byla otestovaná pouze knihovna `PF_RING`.

Pro uložení zachycené komunikace může být použito více formátů, avšak musí být schopny uchovat i přidanou informaci, co ne všechny z formátů splňují. Vybraný byl `pcap-ng` formát pro jeho jednoduchost a rozšiřitelnost. Získané informace o komunikujících aplikacích jsou uložené v speciálním bloku na konci souboru se zachycenou komunikaci ve formě aplikačních tagů. Každý tag reprezentuje jednu aplikaci a obsahuje záznam pro každý její rozpoznán socket a jeho asociaci se zachycenými pakety.

Výsledkem práce je nástroj, který pracuje na platformě Linux pro IPv4 a IPv6 spojení a na platformě Windows pro IPv4 spojení. Úspěšnost nástroje závisí na počtu komunikujících aplikací a frekvenci jejich posílání zpráv. Důležitá je také doba, po kterou je otevřený komunikující socket, aby ho nástroj stihl rozeznat. Na platformě Linux zvládá nástroj pracovat u jedné komunikující aplikace rychlostí 1 Gbps pro pakety větší než 200 B a při šestnácti komunikujících aplikacích pro pakety větší než 1280 B. Výkonnostní testy byly na Windows ovlivněny chybnými ovladači síťové karty, které nebyly schopné využít celý její potenciál a od určité velikosti paketů byly schopny odesílat pouze rychlostí 100 Mbps. Nástroj používá různé způsoby pro snížení trvání zpracování paketu jako cache s již rozpoznanými aplikacemi, kruhový buffer pro zvládnutí náhlých výkyvů síťového provozu či rozdělení práce na podúlohy do vláken. Nástroj je implementován v jazyce C++ a zdokumentovaný nástrojem `Doxygen`. Komunikující aplikace jsou identifikovány pomocí informací o otevřených spojeních, které jsou poskytovány operačním systémem, proto musí být zachytávání prováděno na koncové stanici.

Práce má potenciál v oblasti bezpečnosti jako je např. detekce škodlivého softwaru v síti. Může také sloužit jako účinný nástroj síťovým administrátorům, kterým dovolí odhalit komunikující aplikace či jejich nevhodné chování.

Network traffic capturing with application tags

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Jan Pluskal from the Brno University of Technology, Faculty of Information Technology. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Jozef Zuzelka
May 16, 2017

Acknowledgements

I would like to thank Ing. Jan Pluskal for his help and professional advices. I also deeply appreciate customized bachelor's thesis assignment which fit my interests.

Contents

Chapter 1

Introduction

The aim of this research is to design and implement a tool which will capture network traffic and save it extended by an application information. An ordinary capture file usually contains a communication of multiple network applications. Stored information about applications and their communication can help if we are interested just in a 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[1].

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 Gbps networks with minimal packet loss. In order to save captured traffic and extend it with custom information in the same file, a pcap-ng file format is used. This format is open-source, well documented and widely supported.

Existing tools either do not work in Linux and FreeBSD or they have just part of required functionality. Ntopng[2] 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 [16].

Next tools, which show applications and their connections are `lsof` and `netstat`. They are command-line utilities which list open files by processes, and network connections, respectively. They can print open UDP and TCP sockets and applications which opened them. However, this information is printed just once after they are run and they do not capture traffic.

Popular network sniffers like *Wireshark*[3] and *tcpdump*[4] offer network capture, but neither extends captured traffic with information about communicating applications.

[1] https://en.wikipedia.org/wiki/Network_forensics
[2] http://www.ntop.org/products/traffic-analysis/ntop/
[3] https://www.wireshark.org
[4] http://www.tcpdump.org

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 chapter 2 is presented how to capture network traffic. This can be done on different network layers with different results. Captured traffic is needed to save to an output file. *Pcap* and *pcap-ng* are the most widely used capture file formats. These formats are described and compared in chapter 3. Chapter 4 presents methods that can be used to obtain a list of network connections and communicating processes on different platforms, while implementation details and a tool design are described in chapter 5. Chapter 6 describes structure of data block appended to the end of the output file. Packet loss level with various packet sizes is shown in chapter 7. Finally, chapter 8 summarizes the results.

Chapter 2

Network Traffic Capture

There are two popular application programming interfaces used for writing programs which use the Internet protocols. The first is *TLI* (Transport Layer Interface) introduced with AT&T's System V, Release 3 in 1968[1]. Sometimes it is also called *XTI* (X/Open Transport Interface), but it is developed by an international group of computer vendors who produce their own set of standards and it is a superset of *TLI* [32]. *TLI* was modeled after the ISO[2] Transport Service Definition and provides an API between OSI[3] transport and session layers. *TLI/XTI* implementation is a part of the standard programming interfaces on modern Unix System V operating systems where it is implemented using STREAMS [20].

Sockets are the second application programming interface. They are also called Berkeley sockets, since it was widely released with the BSD system. *Sockets* are more common then TLI/XTI and they are used on the target platforms, so *TLI/XTI* will not be further discussed. The sockets interface was later ported to many non-BSD Unix systems and also to many non-Unix systems. Figure 2.1 shows that sockets API is located in the communication model between the application layer and the transport layer. The sockets API is not a layer in the communication model.

The *socket* is one endpoint in a two-way communication link between two programs on the network. A *socket* is created with `socket()` system call. By its arguments, an appropriate socket is created and file descriptor, also called handle, is returned. An application can work with this file descriptor and can call `read` and `write` operations like to any other block device. The *socket* itself is a data structure stored in operating memory containing connection information. Its structure depends on a socket type. *Sockets* can work with layer 3 or layer 2 of ISO/OSI reference model, which is illustrated using the arrows in Figure 2.1 [32].

Common applications communicating on the network usually use sockets from family `AF_INET` (`PF_INET`). These sockets work with the transport layer of ISO/OSI reference model. When PC receives a packet, NIC's (Network Interface Controller) device driver sets the L3 protocol identifier and the packet type. In case the L2 destination address of the frame is different from the address of the receiving interface, the packet type is set to `PACKET_OTHERHOST`, otherwise it tells if the packet is broadcast, multicast or unicast. If the NIC has been put into the promiscuous mode it passes the frame up to higher layers otherwise it is discarded. The kernel routes traffic at every level to the proper protocol by invoking the handler function registered in the kernel by that protocol. This is called

[1] https://docs.oracle.com/cd/E19455-01/806-1017/6jab5di2n/index.html
[2] International Standards Organization
[3] Open System Interconnection
[4] http://www.ibm.com/support/knowledgecenter/en/ssw_i5_54/rzab6/howdosockets.htm

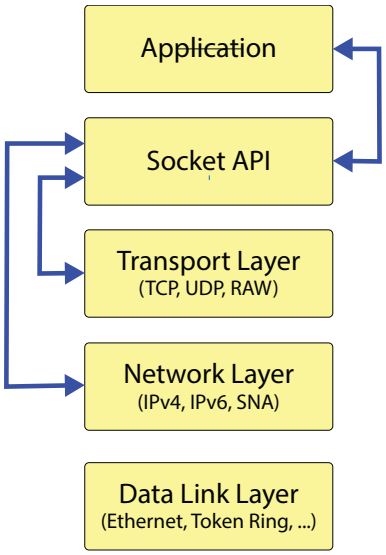


Figure 2.1: Communication model[4]

network stack or TCP/IP stack. When the packet is passed to the L3 layer, the IP header is checked, and checksums are computed. Then it is decided whether the packet has to be locally delivered or forwarded, and IP options are processed. If the packet's destination corresponds to the local host, it must always be defragmented and passed as a whole for local delivery. The packet is then passed to the L4 layer and later to an appropriate socket/application [1].

Following sections contain a description of traffic capture using sockets on different network layers, but also other methods like using kernel modules which bypass TCP/IP stack. Most packet capturing methods make copies of the packets, and do not remove them from the system's network stack, but there are also frameworks which take over the whole network card, not allowing any traffic on that NIC to reach the kernel.

2.1 Capturing on Network Layer

Not all the L4 protocols are implemented in kernel space. An application can use *raw* sockets to bypass L4 in kernel space. When using *raw* sockets, the application receives the whole IP packet with all necessary L4 information (see section 2.3). This makes it possible to implement new L4 protocols in user space. By default, only the superuser, or someone who have `CAP_NET_RAW` capability on, can open/create a *raw* socket.

If the process creates a *raw* socket, also referred as `SOCK_RAW`, with a nonzero protocol value (the third argument to socket() function), and if that value does not match `IPPROTO_ICMP`, `IPPROTO_IGMP` and `IPPROTO_RAW`, then the wildcard entry with a protocol value of 0 will be used. A protocol of `IPPROTO_RAW` can send any IP protocol that is specified in the passed header, but only incoming packets destined for that protocol are received. This behavior is actually OS dependent. Linux, after the IP layer processes a new incoming IP datagram, invokes a registered transport protocol handler depending on the protocol field of the IP header. However before it delivers the datagram to the handler, it checks every time if an application has created a raw socket with the same protocol number. If there are one or more such applications, it makes a copy of the datagram and delivers it

to them as well[5]. BSD (Berkeley Software Distribution) systems take another approach. It never passes TCP or UDP packets to *raw* sockets. Such packets need to be read directly at the datalink layer by using PF_PACKET, BPF, etc. BSD passes to a raw socket every IP datagram with a protocol field that is not registered in a kernel and all IGMP and ICMP packets after kernel finishes processing them, while Linux may pass all IP protocols to the *raw* socket [11].

Raw sockets are not suitable for sniffers as they do not receive all the packets and also they are very slow in comparison with other methods, because of extra system calls and lack of using buffers. This will be discussed in later sections.

2.2 Capturing on Datalink Layer

Access to the datalink 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 on the local interface. Different platforms use different methods how to access datalink layer, e.g., *Tru64* uses `packetfilter`[6], *IRIX* uses `snoop`[7], *SunOS* uses `DLPI` (formerly `NIT`)[8], *Solaris* and *HP-UX* use `DLPI`[9], *BSD* uses `BPF`[10] and *Linux* uses `PF_PACKET`[11]. Following sections describe some of these methods.

2.2.1 PF_PACKET

Current Linux kernels use the protocol family `PF_PACKET` to allow direct access to all packets on the network. Older kernels had a special, now obsolete, socket type `SOCK_PACKET` to implement this feature.

```
int s = socket(PF_PACKET, type, protocol);
```

Listing 2.1: Linux - opening socket with access to datalink layer

`PF_PACKET` completely bypass the kernel networking stack and directly receives and sends the packet to the link layer. This protocol family can be used with `SOCK_DGRAM` or `SOCK_RAW` socket types, depending on the application requirement. If `SOCK_DGRAM` is used, the application receives the packet without Ethernet header (also called "cooked" packets). If `SOCK_RAW` is used, application receives the complete frame including link layer header. This header has to be restored because it has been removed either by the network card or by its driver. Restoring the link layer header is not possible if it was removed at the hardware level, since that information never gets to the system main memory and is invisible outside the network device [10]. Protocol argument (from Listing 2.1) is used to filter only specific types of packet, like if the protocol is `ETH_P_IP`, then only IP packets are received. To receive all protocol packets, protocol `ETH_P_ALL` should be used. Older `SOCK_PACKET` sockets always return the complete link layer frame, and it cannot be linked to a device, so it receives frames from all devices (Ethernet, PPP links, SLIP links, ...). `PF_PACKET` does not do any buffering (there is a normal socket receive buffer, but multiple

[5] https://linux.die.net/man/7/raw
[6] http://nixdoc.net/man-pages/Tru64/man7/packetfilter.7.html
[7] http://nixdoc.net/man-pages/IRIX/man7/snoop.7.html
[8] http://docs.oracle.com/cd/E19455-01/805-6331/6j5vgg6a5/index.html
[9] http://nixdoc.net/man-pages/HP-UX/man7/dlpi.7.html
[10] http://nixdoc.net/man-pages/FreeBSD/man4/bpf.4.html
[11] http://nixdoc.net/man-pages/Linux/man7/packet.7.html

frames cannot be suffered together and passed to the application with a single `read`) and has no kernel filtering[12], although it can be achieved in combination with BPF filters and PACKET_MMAP [27].

2.2.2 Berkeley Packet Filter (BPF)

The *Berkeley Packet Filter* (BPF) provides direct access to datalink layer on *BSD systems. Each datalink driver calls *BPF* right before a packet is transmitted and right after a packet is received. At the time when *BPF* was introduced, it was up to 20 times faster than the original Unix packet filter and up to 100 times faster than Sun's NIT running on the same hardware. *BPF* is protocol independent, which means that the kernel has not to be modified to add new protocol support [13]. Linux version derived from the *BPF*, called *Linux Socket Filtering* (LSF), uses the very same mechanism of filtering in the Linux kernel[13]. However, *LSF* is just a filter.

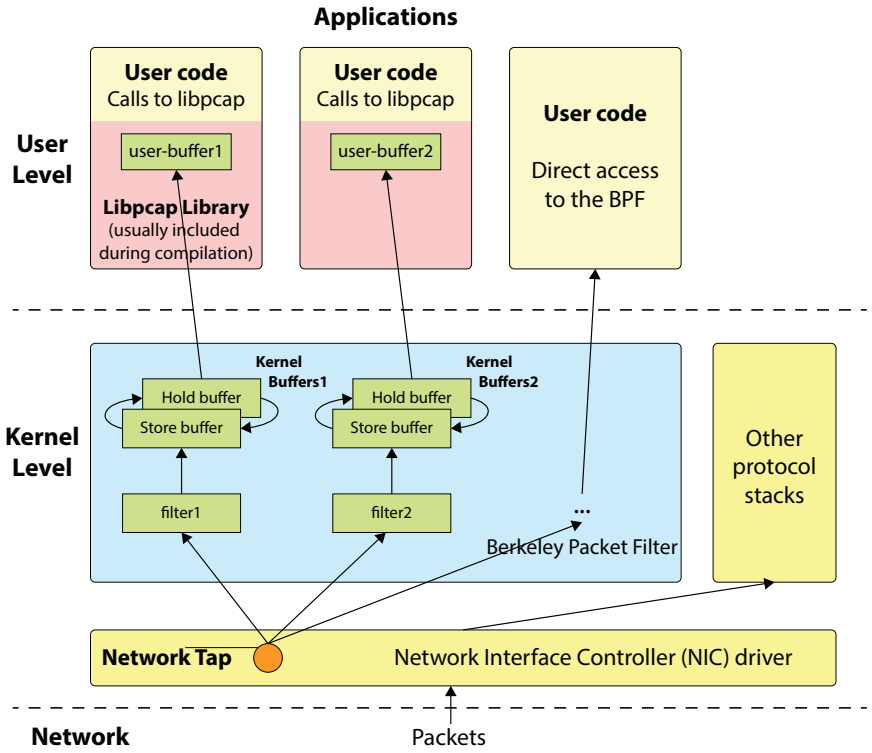


Figure 2.2: BPF Capturing Components[14]

BPF implements a register-based filter machine that applies application-specific filters to each received packet. We can use filters written in the machine language of this pseudomachine, or we can compile ASCII strings into this machine language[15]. Since a process might want to look at every packet on the network and the time between packets can be only a few microseconds, it is not possible to do a read system call per packet and *BPF*

[12] http://faqs.cs.uu.nl/na-dir/internet/tcp-ip/raw-ip-faq.html
[13] https://www.kernel.org/doc/Documentation/networking/filter.txt
[14] https://www.winpcap.org/docs/iscc01-wpcap.pdf
[15] https://sharkfest.wireshark.org/sharkfest.11/presentations/McCanne-Sharkfest'11_Keynote_Address.pdf

must collect the data from several packets and return it as a unit when the monitoring application does a read. To achieve this, *BPF* maintains two buffers for each application and fills on one while the other is being copied to the application. When the buffer is full, *BPF* swaps buffers. Capturing components are shown in Figure 2.2. In 2007, the zero-copy support (see section 2.3) was added[16].
To access *BPF*, we must `open` a BPF device that is not currently open. For example, we could try `/dev/bpf0`, and if the `EBUSY` error is returned, then we could try `/dev/bpf1`, and so on. Once a device is opened, `read` and `write` operations can be used to communicate on the network [27, 13].

In 2004, the *Extended BPF* (eBPF) was introduced. This version adds a number of capabilities and performance improvements like additional registers, BPF maps and more[17]. Recently, it is called just *BPF*, and it can be used, except communication on the network, to trace application calls like `DTrace`[18] does [8].

2.2.3 Libpcap

Libpcap provides a single API that interfaces with different OS dependent packet capturing APIs (described in previous sections). *Libpcap* was developed by `tcpdump`[19] developers in 1994. Originally it supported packet capture only but later, in 2005, support for sending packets was added[20]. This library is used in applications like Wireshark, tcpdump, snort[21], nmap[22], suricata[23], scapy[24] and others. Although most packet capture interfaces support in-kernel filtering, *libpcap* utilizes in-kernel filtering only for the BPF interface. On systems that don't have BPF, all packets are read into user-space and the BPF filters are evaluated in the *libpcap* library[25].

In my project, pcap-ng file format will be used. Although *libpcap* got limited support for reading these files in version 1.1.0, no enhancements for pcap-ng support were made from then. Newer versions of macOS have functions in *libpcap* to write pcap-ng files. That version is open-source so that we could use the code. However apple's implementation of *libpcap* is licensed under the APSL[26], so if it were incorporated into the standard *libpcap*, it would put *libpcap* under the APSL, which has patent clauses that some OSes that ship libpcap might find objectionable, so that code won't be incorporated into libpcap unless Apple relicenses it. *Libpcap* currently supports SunOS 3.x and 4.x (NIT and BPF), Solaris (DLPI), HP-UX (DLPI), IRIX (snoop), Linux (PF_PACKET), ULTRIX and Tru64 Unix (packetfilter) and BSD including macOS (BPF) [18].

[16] http://www.watson.org/~robert/freebsd/2007asiabsdcon/20070309-devsummit-zerocopybpf.pdf
[17] https://videos.cdn.redhat.com/summit2015/presentations/13737_an-overview-of-linux-networking-subsystem-extended-bpf.pdf
[18] http://dtrace.org/blogs/about/
[19] http://www.tcpdump.org
[20] http://www.tcpdump.org/libpcap-changes.txt
[21] https://www.snort.org/
[22] https://nmap.org/
[23] https://suricata-ids.org/
[24] http://www.secdev.org/projects/scapy/
[25] https://github.com/the-tcpdump-group/libpcap
[26] https://opensource.apple.com/apsl

2.2.4 WinPcap

WinPcap is the Windows version of the *libpcap* library. *WinPcap* includes an optimized kernel-mode driver, called Netgroup Packet Filter (NPF), and a set of user-level libraries that are libpcap-compatible. *WinPcap* retains the most important modules from BPF shown in Figure 2.2: a filtering machine, two buffers (kernel and user) and a couple of libraries at a user level. *WinPcap* components are shown in Figure 2.3.

WinPcap has the same copy overhead of the original libpcap/BPF and packets are copied two times (from network driver to the kernel, then to user space). The filtering is performed when the packet is still into the network driver's memory, thus before buffering.

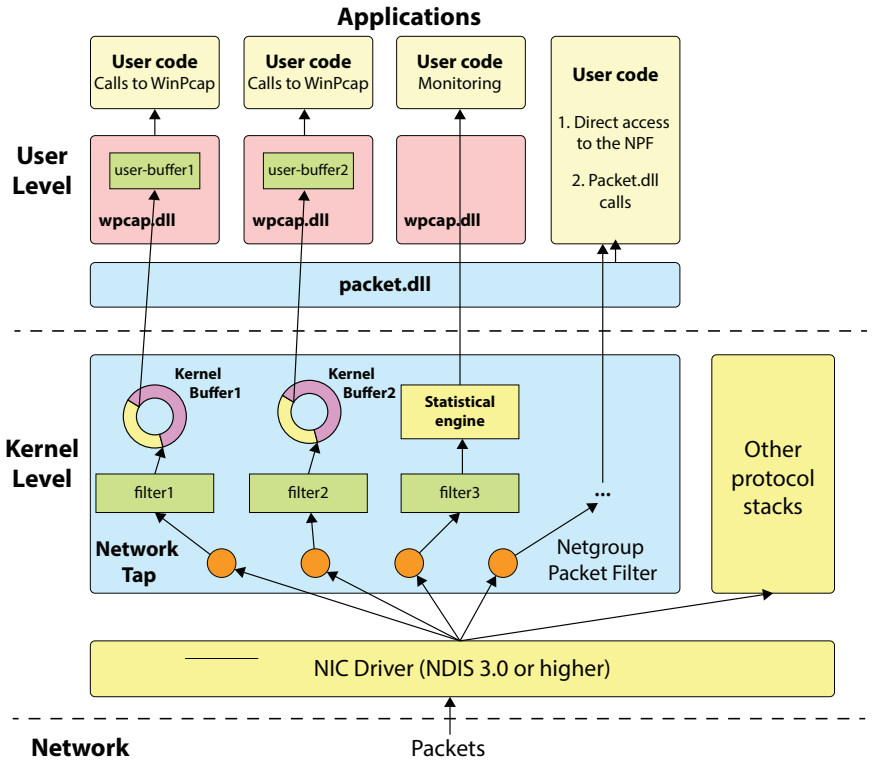


Figure 2.3: WinPcap and NPF[27]

An important architectural difference between BPF and NPF is the choice of a circular buffer as kernel buffer. The entire kernel buffer is usually copied using a single `read`, thus decreasing the number of system calls and therefore the number of context switches between user and kernel mode. Another difference is that NPF is a part of the protocol stack and it interacts with the OS like any other network protocol (Windows does not allow modifying the OS and the NIC drivers in order to add a feature) while BPF interacts with the system through a single callback function [6].

Windows networking architecture is based on NDIS (Network Driver Interface Specification) standard, the lowest level networking portion of the Windows kernels. NDIS is a specification for building network interface (NIC) drivers and protocol drivers and to handle interaction among them. Intuitively, the core of the capture process must run at the kernel level and it must be able to access the packets before the protocol stack processes them [21].

[27] https://www.winpcap.org/docs/iscc01-wpcap.pdf

WinPcap is implemented in the NDIS 5.x driver model, so it might not work in some builds of Windows 10 correctly. Additionally, *WinPcap* cannot capture 802.1Q tagged VLAN headers received by NDIS 6.x NICs in Windows 7, 8 and 10. To solve these problems, Daiyuu Nobori wrote *Win10Pcap*[28] library. *Win10Pcap* is compatible with NDIS 6.x driver model to work stably with Windows 10 and also supports capturing IEEE802.1Q VLAN tags. *Win10Pcap* has the binary-compatibility with the original *WinPcap* Dynamic Link Libraries (DLLs).

Npcap is the Nmap Project's packet sniffing library for Windows. It is based on *WinPcap* libraries, but with improved speed, portability, security and efficiency[29]. *Npcap* provides support for the latest libpcap API (actually 1.8.0) by accepting *libpcap* as a Git submodule while original *WinPcap* was shipped with the deprecated *libpcap* version 1.0.0. *Npcap* is unlike *WinPcap*, which has its latest version from March 2013, still maintained. If *Npcap* is not installed in the compatibility mode, both original *WinPcap* and *Npcap* drivers can be installed on the same system. Otherwise, *Npcap* will use standard *WinPcap* directories and software build with *WinPcap* will transparently use *Npcap* instead.

2.3 Capture Problems

In section 2.1 is written that using *raw* sockets the application receives the whole IP packet. This is not truth for IPv6 communication. When an application opens *raw* socket from AF_INET6 family, it receives L4 header, but IPv6 header is not passed to it. However, almost all fields in an IPv6 header and all extension headers are available to the application through socket options or ancillary data [27]. Another problem on BSD platform is that the application never receives TCP or UDP packets on `SOCK_RAW` socket. Solution to problems mentioned above is to use PF_PACKET, BPF or other approaches to capture on the L2 layer. The main problem which we can face is packet loss. Next follows a description of common principles to speed up capture process and reduce packet loss.

Data buffer The kernel buffers up to X bytes of packet data and pass the packets one by one at the user's request. If the amount exceeds a certain limit (the limit can be set), the packets are dropped. The purpose of the buffering is to reduce the number of system calls. Some implementations use two buffers; one is user-level buffer and second kernel-level. The size of the user buffer is very important because it determines the maximum amount of data that can be copied from kernel space to user space within a single system call. Important is also the minimum amount of data that can be copied in a single call (low number of system calls versus immediate access to received packet). Buffering is done in, e.g., BPF, DLPI, NIT, WinPcap, etc.

Filtering Sometimes, we don't want to capture all packets. Packet filters can be used to specify which packets we are interested in. This way we can limit the scope of the packets read by an application and reduce packet loss. Typically, we have set of opcodes for routines to perform on the packet. The opcodes usually perform very simple operations[30]. The important thing is if the filter is applied before or after buffering. If it is applied before buffering (BPF) it reduces an amount of data which are copied from the kernel to the

[28] http://www.win10pcap.org/
[29] https://nmap.org/npcap/
[30] https://leonerds-code.blogspot.cz/2010/05/pfpacket-linux-socket-filters-and-ipv6.html

process and decreases the number of context switches between user and kernel level, which is expensive. In the latter case (NIT) a copy of each packet is always made, and many CPU cycles will be wasted copying unwanted packets [13].

Working with a portion of packet Sniffer applications usually need only the packet headers, not the packet data. In this case, we can pass to the application only a part of each packet. This is called the `snaplen`. This reduces the amount of data copied to the application and reduces the overhead. This is by default used in `tcpdump`.

Zero-copy During normal packet processing, minimally two copies per packet are made. One from NIC buffer to the kernel buffer and then the second copy from kernel buffer to user memory. This causes a significant performance overhead. The solution is to use shared memory buffer between user process and kernel, which is used by BPF[31] and PACKET_MMAP. This eliminates copy to user space but not in-kernel so there is still one copy. In order to have no copies, the special NIC driver is needed, which has direct access to the kernel-user memory space [12].

Packet fanout Multiple opened sockets can be bound to a cluster and every socket can process just a portion of the packets. Fanout supports multiple algorithms to divide traffic between sockets. Packets can be aggregated using hash, so packets from the same netflow are sent to one socket, they can be spread using round robin algoritm or randomly. More fanout policies and more details can be found in [17].

2.4 Speeding Up Capture Process

On fast networks, the application can be affected by packet loss. Follows description of libraries which use methods mentioned in section 2.3. Just the most widely known were described and specialized ones like OpenOnload[32], PacketShader[33] or Sniffer10G[34] were skipped.

PACKET_MMAP

Packet_mmap is a Linux API for fast packet sniffing. *Packet_mmap* consists of a ring buffer mapped in shared memory that both kernel and a user program can directly access. To use *packet_mmap*, kernel has to be compiled with `CONFIG_PACKET` and `CONFIG_PACKET_MMAP` flags. *Packet_mmap* provides a size configurable circular buffer mapped in user space that can be used to either send or receive packets. This eliminates copy to user space but not to the kernel so there is still one copy. *PF_PACKET* (see section 2.2.1) works by registering new protocol handler on target network devices. The NIC is unaware of the *packet_mmap* framework and the kernel does not abstract `sk_buff`[35] memory allocation enough that the driver can DMA[36] data directly into the `PACKET_RX_RING` (receive ring buffer). The NIC just allocates a random `sk_buff` and then passes the `sk_buff` to the

[31] https://www.freebsd.org/cgi/man.cgi?query=bpf
[32] http://www.openonload.org
[33] http://shader.kaist.edu/packetshader/
[34] http://www.cspi.com/ethernet-products/software/sniffer10g/
[35] http://www.makelinux.net/ldd3/chp-17-sect-10
[36] Direct Memory Access

protocol handlers [33]. In order to achieve true zero-copy using *PF_PACKET*, the NIC driver would need to be able to allocate `sk_buff` structure where the data segment was part of the `PACKET_RX_RING` buffer[37]. *Packet_mmap* is implemented in Linux kernel and it is compatible with `libtrace`[38] and `libpcap`[39] libraries. If *packet_mmap* is supported by the kernel `Libpcap` will use it[40].

Socket creation is the same as in Listing 2.1. The important action is allocation of RX ring buffer and TX ring buffer. To do this we need to use the `setsockopt` call as is show in Listing 2.2.

The most important argument in the calls is the request structure `req`, which defines the ring buffer parameters. It specifies minimal size of contiguous block, number of blocks, size of frame and total number of frames. In this structure, the circular buffer, which is unswappable memory in the kernel, properties can be set. As this buffers are mapped to user space, the applications can directly read the packet and read the meta information like timestamp without using any system call. Next, it is needed to map the allocated buffer to the user process using `mmap()` and then to wait for incoming packets using `poll()` [5].

```
1 /* capture process */
2 setsockopt(fd, SOL_PACKET, PACKET_RX_RING, (void*)&req, sizeof(req));
3 /* transmission process */
4 setsockopt(fd, SOL_PACKET, PACKET_TX_RING, (void*)&req, sizeof(req));
```

Listing 2.2: PACKET_MMAP - Allocation of the circular buffer

PF_RING

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 [26].

There are two versions of *PF_RING*: a "Vanilla" open-source version[41], and a *ZC* ("Zero Copy") version[42]. *PF_RING ZC* allows the application to completely bypass the host operating system's network processing, resulting in much better performance, but it is licensed per mac address[43]. It can be considered as the successor of DNA and LibZero[44], which were parts of the *PF_RING* library.

The *PF_RING* can work in three modes. In the first – default mode, packets are sent to *PF_RING* via the standard kernel mechanisms. In this, setup the packets are both sent to *PF_RING* but to all other kernel components. All NIC drivers support this mode.

In the second mode, packets are sent directly by the NIC driver to *PF_RING*, packets are still propagated to other kernel components. In this mode packet capture is accelerated

[37] https://yusufonlinux.blogspot.cz/2010/11/data-link-access-and-zero-copy.html
[38] https://github.com/wanduow/libtrace
[39] https://github.com/the-tcpdump-group/libpcap/blob/master/pcap-linux.c#L27
[40] https://github.com/the-tcpdump-group/libpcap/blob/master/pcap-linux.c#L1516
[41] https://github.com/ntop/PF_RING/blob/dev/doc/README.vanilla.md
[42] http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/
[43] https://shop.ntop.org/cart.php
[44] http://www.ntop.org/pf_ring/introducing-pf_ring-dna-direct-nic-access/

because packets are copied by the NIC driver without passing through the usual kernel path. In this case, a NIC driver with *PF_RING* support must be used.

When *PF_RING* works in DNA (Direct NIC Access) mode, packets are not propagated to other kernel components as this slows down packet capture. *PF_RING* maintains a circular buffer in the kernel. The ring buffer has two interfaces, one for the card to write data packets and one for the application to read data. When the external data packet arrives, the NIC receive data and copy it into the ring, and then the data packet itself is discarded, no into kernel protocol stack of Linux. The application gets read pointer of the ring buffer by opening the socket descriptor of *PF_RING* types, and then access the ring buffer by mmap [25]. *PF_RING* can be used in combination with NAPI[45] to improve the performance[46] and using libpcap-over-PF_RING we can take any pcap-based legacy application, and run all those applications without any code change [15].

Netmap

Netmap is community-driven software framework that started as a research at the University of Pisa. Like *PF_RING*, it uses a kernel module and modified NIC drivers in an attempt to speed up packet capture. The kernel module implements a ring buffer. This buffer is a pre-allocated memory that, once filled, is over-written from the beginning. *Netmap* has its own API for network applications to pull data from the ring buffer, and also includes a modified version of libpcap for supporting legacy applications. 10G support through *Netmap* is only available for Intel NICs. *Netmap* is available for FreeBSD, Linux and also Windows [24]. Compared with *PF_RING* Vanilla, *Netmap* has a lower rate of packet loss [26].
In *netmap*, a NIC can work in regular mode (where NIC exchanges data with host stack), and in netmap mode, where the NIC rings are disconnected from the host network stack, and exchange packets through the netmap API. Other applications cannot use this interface to communicate over network anymore. Two additional netmap rings let the application talk to the host stack [23]. Network applications like `tcpdump` which use *libpcap* can use *netmap's* modified version of libpcap and interface with *netmap* prefix and postfix[47]. The libpcap emulation library adds a significant overhead about 80-100 ns per packet [22].

Data Plane Development Kit (DPDK)

DPDK is a set of libraries and drivers for fast packet processing. It runs mostly in Linux userland, but a FreeBSD port is available for a subset of DPDK features. *DPDK* framework is written in C and it was created especially for Intel chips. It works by taking over a network card, and implements a hardware driver in userspace. It is done on a PCI device level with a form of userspace IO (UIO)[48] [9].
Using Flow Bifurcation[49], the incoming data traffic can be split to user space applications (for example DPDK application) and kernel space programs (such as the Linux kernel stack) and using Kernel NIC Interface (KNI)[50] userspace applications can interface with the kernel network stack.

[45] https://wiki.linuxfoundation.org/networking/napi
[46] http://www.ntop.org/pf_ring/packet-capture-performance-at-10-gbit-pf_ring-vs-tnapi/
[47] https://www.freebsd.org/cgi/man.cgi?query=netmap&sektion=4#IOCTLS
[48] https://www.osadl.org/fileadmin/dam/rtlws/12/Koch.pdf
[49] http://dpdk.org/doc/guides/howto/flow_bifurcation.html
[50] http://dpdk.org/doc/guides/prog_guide/kernel_nic_interface.html#kni

Snabb

Snabb (formerly Snabb Switch) is a networking framework written in LUA mostly geared towards writing L2 applications. It is similar to *DPDK* since it is a full framework and relies on UIO[51]. *Snabb* compiles into a stand-alone executable called `snabb`[52] which calls a C function compiled to a shared library. Similarly to *DPDK*, it takes over the whole network card, not allowing any traffic on that NIC to reach the kernel, which is not what we want in a network sniffer.

nCap

nCap is a high-speed packet capture and transmission library. It is a pure userland library that is based on a special kernel driver. *nCap* consists of the accelerated kernel driver that provides low-level support and ethernet device programming and user space SDK that can be used directly or through an enhanced `libpcap`. Currently, the accelerated *nCap* driver is provided only for Intel 1 Gbit (copper, SX and LX) and 10 Gbit Ethernet cards. To run *nCap*, a precompiled Linux kernel with all needed kernel modules is needed. Once the kernel is installed we need to modify boot loader in order to let the system access the new kernel [7]. When the device is open using nCap, it is completely controlled by userland application and only one application at time can use the NIC[53].

PFQ

PFQ[54] is a functional networking framework designed for the Linux that allows efficient packet capture/transmission. It is highly optimized for multi-core architecture. *PFQ* comes with modified and accelerated `libpcap` library and works with any network device driver. It does not require any modifications to network device drivers and exposes programming interfaces to multi-threaded applications natively designed to run on top of it, as well as to legacy monitoring tools using the pcap library. The package consists of a Linux kernel module and a C++ user space library [3]. More information about *PFQ* can be found in [2] and [4].

2.5 Summary

There are several options where we can capture traffic on its way from wire to an application. When we forget about special hardware and special device drivers, one possibility remains and that is to make a copy of the packet while it is being processed in the network stack. Raw sockets can be used to capture traffic on the network layer, but we can face the problem of not receiving all packets on BSD platform. Another problem is, that packets are being processed in the network stack, which takes some time and resources. Network traffic analyzer usually wants the whole packet and it does not need the packet headers to be parsed, so kernel brings another vain overhead. Because of this, raw sockets are not applicable in network sniffers.

[51] https://blog.cloudflare.com/kernel-bypass/

[52] https://github.com/snabbco/snabb

[53] https://repository.ellak.gr/ellak/bitstream/11087/1537/1/5-deri-high-speed-network-analysis.pdf

[54] http://www.pfq.io

Another option is to capture traffic on datalink layer. In this case, *raw* packets are passed to an application and depending on the platform it is possible to use additional features like buffers and filters to speed up capture process and reduce packet loss. Different platforms use different methods of network capture on datalink layer and that is where libpcap come in. Using this library, we can write portable code and use single API on different platforms to capture on datalink layer.

Unfortunately, although the *libpcap* is quite efficient, it is not fast enough for 1 Gbps links. To achieve higher throughput, we can use frameworks in which captured packets are passed right to an application and completely bypass the kernel network stack. This can radically reduce packet loss as it reduces amount of data copying, context switching and so on. These frameworks usually use some form of custom drivers in combination with buffers and filters. Some of the applications are interested in the headers of the captured packet only, and not in the captured data itself, so we can ignore the data part and again reduce performance overhead.

In our case, to reduce time needed to assign packet to an application we can also use an application cache which would contain already known applications. This is described in section 5.1.

Chapter 3

Capture File Formats

Captured data have to be stored for their later processing like computing amount of transferred data per application or just showing communication of selected applications. Various software uses different file formats. For example, Microsoft Network Monitor uses Netmon (`.cap`)[1] file format, Oracle uses for their captures `snoop`[2] format, Novell LANalyzer uses `.TR1` file format and libpcap uses `pcap`[3] file format. Many formats are either closed source or poorly documented, if at all. `Pcap` and `pcap-ng`[4] are fully documented and they are widely supported across various network capture software. Both file formats and differences between them are described in following sections.

3.1 PCAP

Pcap is basic file format used to store captured traffic. In unix-like[5] systems, *pcap* is implemented in the `libpcap` library. The library and its ports are described in section 2.2. Although, network tools start using newer *pcap-ng* file format, tcpdump still uses *pcap* as its default output capture file format, because `libpcap` does not support writing to the *pcap-ng* file format, so far.

Global Header	Packet Header1	Packet Data1	Packet Header2	Packet Data2	...

Figure 3.1: PCAP capture format structure[6]

Figure 3.1 shows a structure of *pcap* file. The blue parts are added by `libpcap` or any other capture software, while the parts in red are original captured data. *Pcap* format consists of global header and pairs of headers and data of captured packets [29]. The file format structure with example values is shown in Figure B.1.

Global Header is inserted only once in the file. It has fixed size of 24 bytes and it is located at the start of the file. A content of the header is listed in Figure 3.2. The first 4 bytes constitute the `magic_number`. This number is used to detect the file format itself

[1] https://msdn.microsoft.com/en-us/library/windows/desktop/ee817717(v=vs.85).aspx
[2] https://tools.ietf.org/html/rfc1761
[3] https://wiki.wireshark.org/Development/LibpcapFileFormat
[4] 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
[5] http://www.linfo.org/unix-like.html
[6] http://www.kroosec.com/2012/10/a-look-at-pcap-file-format.htm

and the byte ordering. Next are `version_major` and `version_minor` numbers that specify the file format version. After that follow three fields namely `thiszone` (correction time in seconds between GMT (UTC) time and time zone of timestamps in packet headers), `sigfigs` (accuracy of the timestamps) and `snaplen` (maximal length of captured packets). The last part of the global header, field `network`, stores link-layer header type of captured packets (FDDI, PPP, USB, RAW, ...)[7]. The global header is followed by packet header-data pairs.

```
1 typedef struct pcap_hdr_s {
2     guint32 magic_number;
3     guint16 version_major;
4     guint16 version_minor;
5     gint32  thiszone;
6     guint32 sigfigs;
7     guint32 snaplen;
8     guint32 network;
9 } pcap_hdr_t;
```

Figure 3.2: Pcap - Global Header[29]

```
1 typedef struct pcaprec_hdr_s {
2     guint32 ts_sec;
3     guint32 ts_usec;
4     guint32 incl_len;
5     guint32 orig_len;
6 } pcaprec_hdr_t;
```

Figure 3.3: Pcap - Packet Header[29]

Packet Header follows immediately after Global Header. This header (also like the first one) is added by capture software and contains information about captured packet. This header has also fixed structure, which is listed in Figure 3.3. The header consists of `ts_sec`, `ts_usec`, `incl_len` and `orig_len`. The first, `ts_sec`, is the number of seconds since the Unix epoch[8]. The `ts_usec` contains either microseconds or nanoseconds when this packet was captured, as an offset to `ts_sec`. The next one is `incl_len`. This field stores number of bytes of packet captured in the file. The last 4 bytes contain the length of the packet as it appeared on the wire. It can differ from `incl_len` because of `snaplen` limitation. Each captured packet starts with its own packet header. Data follows after the header. Appendix B contains example of the *pcap* file.

3.2 PCAP Next Generation

PCAP Next Generation (*pcap-ng*) capture file format is used to record captured packets to a file. This file format is unlike previous one extensible. This is very important for us, as we want to extend the packet with new information. *Pcap-ng* is used by *Wireshark*[9] and *TShark*[10] as their default file format for saving capture files. Tools that use `libpcap` for network capturing, like *tcpdump*, does not support writing *pcap-ng* files so far[11].

The new capture format brings features as captures from more interfaces in one file, statistical data, comments can be added and saved to each frame, files can be tagged with metadata about what OS, hardware and application captured the traffic (this can be seen in Figure B.2) and more. Although there is no support for writing *pcap-ng* files in the official `libpcap` library, some tools can write these files using their own implementation[12].

[7] http://www.tcpdump.org/linktypes.html
[8] https://en.wikipedia.org/wiki/Unix_time
[9] https://www.wireshark.org/
[10] https://www.wireshark.org/docs/man-pages/tshark.html
[11] https://github.com/pcapng/pcapng/wiki/Implementations
[12] https://github.com/pcapng/pcapng/wiki/Implementations

3.2.1 General Block Structure

The capture format is organized in blocks, which are appended one to another. Basic block structure is shown in Figure 3.4.

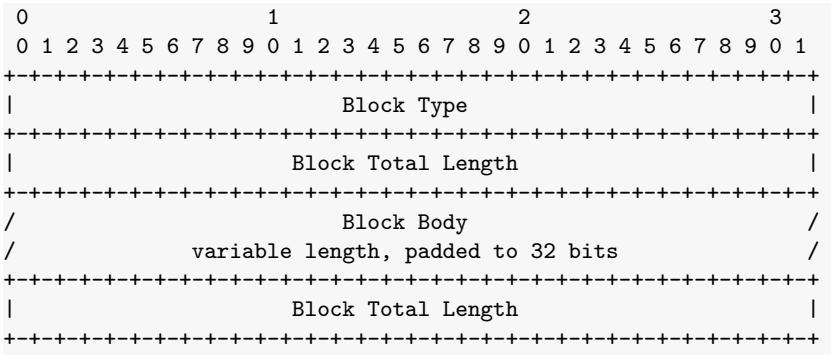


Figure 3.4: Pcap-ng - Basic block structure

Every block has its own unique identifier. This 32-bit number is stored in `Block Type` field. `Block Total Length` contains a length of the block. If the block has no body, length is 12 octets - Block Type and two times Block Total Length. The length is 4B value, and it has to be multiple of 4. A content of the block is stored in `Block Body` part. The last field is a copy of the `Block Total Length` - total size of the block, in octets [28].

3.2.2 Block Types

The *pcap-ng* file format is organized in blocks. These blocks are grouped into four categories. The first group contains blocks which are `mandatory`, so they *must* appear at least once in a file. There is just one block in this group, and that is Section Header Block. The next are `optional` blocks. Blocks in this group, Interface Description Block, Simple Packet Block, Enhanced Packet Block, Interface Statistics Block, Name Resolution Block and Custom block *may* appear in a file. `Obsolete` blocks *should not* appear in new files. In this group is just one block, so far and it is Packet Block. This block was superseded by the Enhanced Packet Block. `Experimental` blocks are in the last group. These blocks "are considered interesting but the authors believe that they deserve more in-depth discussion before being defined" [28]. Follows description of three *pcap-ng* blocks that are used in the output capture file. Their description and block formats are taken from [28]. Appendix B contains example of blocks located in real file.

Section Header Block

This block is important because it is in the `mandatory` group, so it has to be in a file and in order to create a valid *pcap-ng* file it is good to know what this block contains. A structure of the block with an example values is in Figure B.2. This block identifies a beginning of a section of the capture file.
The block starts with a type identification. Section Header Block is identified by sequence "\r\n\n\r" in the `Block Type` field, at the start of the block. This is followed by `Block Total Length`, which copy is also located at the end of the block. There is block body between them which consist of `Byte-Order Magic`, `Major Version`, `Minor Version`, `Section`

`Length`, and `Options`. `Options` are a list of option fields which is terminated with an option of `End Of Options` type.

Interface Description Block

An Interface Description Block (IDB) contains a description of an interface which was used to capture data. Tools that write capture file associate an incrementing 32-bit number to each IDB, called `Interface ID`. The network traffic capture must contain this block because Enhanced Packet Blocks contain a reference to this block. `Interface ID` must be unique within each section, so two sections can have different interfaces with the same ID. The format of the block is shown in Figure B.3. Mandatory fields are (except block's type and length) type of the link layer of the interface, maximum number of octets captured from each packet and reserved field which must be filled with zero. Optionally Options, such as interface name, description, IP addresses, speed and more can be present.

Enhanced Packet Block

An Enhanced Packet Block (EPB) is the standard container for storing packets. This block is optional, packets can also be stored in the Simple Packet Block (SPB). The Enhanced Packet Block is improved, now obsolete, Packet Block. The format of this block is shown in Figure B.4.
The `Block Type` is identified by number 6. Next value is, like in every other block, `Block Total Length`. A body of the block contains `Interface ID`, 64-bit `Timestamp`, `Captured Packet Length`, `Original Packet Length`, `Packet Data` (padded to 32 bits) and `Options`. Some of the valid options are the number of lost packets, a hash of the packet and custom option which is described in section 3.2.3. If a file contains this block, it will also contain an Interface Description Block with the description of an interface identified by the `Interface ID` field.
Captured packet can also be stored in Simple Packet Block format (SPB), but this block does not contain timestamp field, and it would be harder to identify a group of packets in the file, so Enhance Packet Block format is used.

3.2.3 Format Extension

There are two ways how to save additional information about the packet. Either we can insert `Custom Option` field into a list of options to extend some existing block, or we can add a new `Custom Block` to the file.

Custom Options

Each block may embed optional fields. Custom Options are used to store vendor-specific data related to the block they are in. It can be in any block that can have options and can be repeated multiple times. All share a common format (TLV[13]), where an option field starts with an identifier and then length, both two bytes long. Data are located after the `Option Length` field. The format of a Custom Option is shown in Figure 3.5. Some optional fields can be located in a block multiple times, like comments or interface IP addresses, and some options may appear at most once in a given block.
A mandatory part of the Custom Option must contain `Private Enterprise Number` in

[13]Type, Length, Value

addition to option code and its length. This number is described later in this section. The options list is terminated by an option which uses special `End of Option` code.

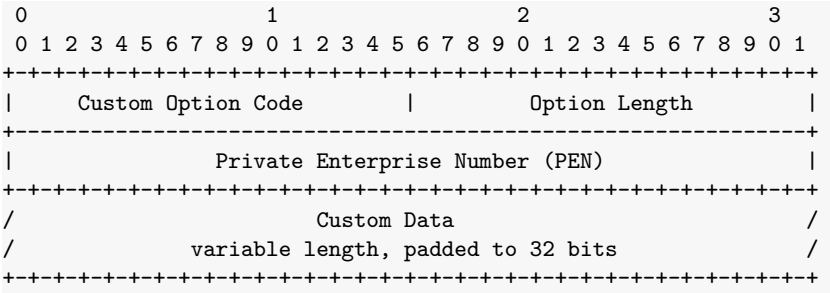


Figure 3.5: Pcap-ng - Custom Options Format[28]

Custom Block

Custom Block is a container for storing custom data that is not part of another block. This block can be repeated any number of times and can appear anywhere in the file, except before the first Section Header Block. The format of a Custom Block is shown in Figure 3.6. Type of the block is defined by codes 0x00000BAD and 0x40000BAD. In case, that the Custom Block depends on some other block or option of another block and its data would be invalid without this block/option, the 0x40000BAD code should be used. Some blocks or options can be removed during copying and because of this, our Custom Block could be invalid. This behavior is indicated by the second code and the Custom Block will not be copied into a new file. This block contains its type and length like every other block. Its body consists of `Private Enterprise Number`, `Custom Data` padded to 32 bits and optional `Options` [28]. The exact structure of the Custom Block used by the tool is described in chapter 6. Appendix B then shows example values in the output file.

Private Enterprise Number

Field with a Private Enterprise Number (PEN) is in both Custom Options and Custom Block. There are two supported use-cases for custom extensions: local and portable. `Local` use means that the extension will be read on one machine, and the same application. It is chosen by a vendor, so there can be a collision with another extension. In order to avoid collisions vendors should use a `portable` method. In this case, the number is assigned by Internet Assigned Numbers Authority (IANA)[14]. It can be shared across applications, organizations and so on. Anyone can register their own PEN number; it does not matter if you are a manufacturer, association, particular user or group of users, everyone can register it for free. To register a PEN number, the complete application[15] has to be submitted. After that, it can take up to 30 days to process the application. The PEN number must be encoded using the same endianness as the Section Header Block to which it belongs [28].

[14] https://www.iana.org/
[15] http://pen.iana.org/pen/PenApplication.page

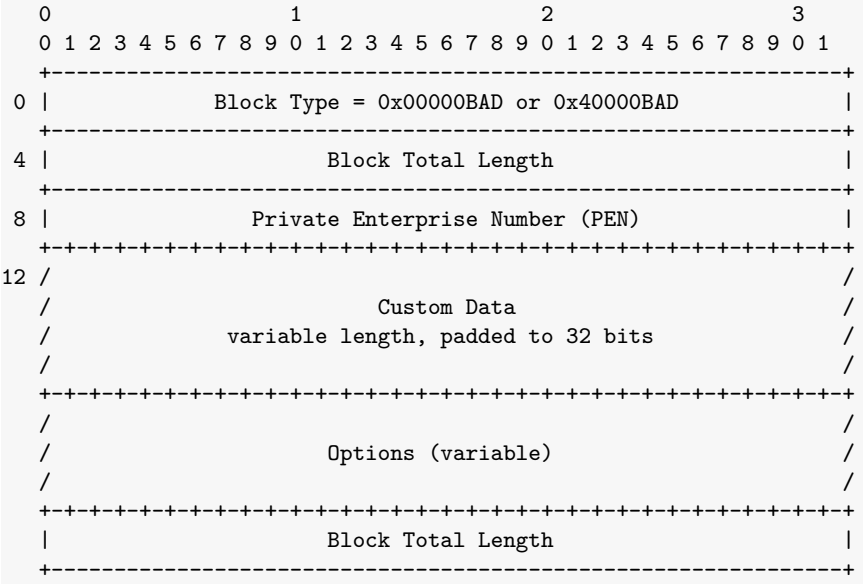


Figure 3.6: Pcap-ng - Custom Block Format[28]

3.3 Summary

Various software uses different file formats. Some are extensible, like NMSG[16] or *pcap-ng* and some not. The original *pcap* file format is part of an API for performing captures on a network interface. In Linux, this is implemented through the `libpcap` library and in Microsoft Windows through the `WinPcap` library. These libraries have been used to create a number of tools over the years, all able to work with *pcap* file format. However, this format is very simple, and as you could see, there is no possibility how to extend it.
A new format called *pcap-ng* is thanks to standardized blocks logical and extensible. *Pcap-ng* supports captures from more interfaces, embedding comments right in the capture file, storing additional metadata in capture file and the most important thing for us - extensibility. Individual block fields are described in more detail in [28].
Pcap-ng can be extended using Custom Options and Custom Blocks. Custom Option is embedded in an existing block. This is good because we have all information about packet in the same block and we do not have to search for additional information in other blocks through the file. During common college TV stream, when there are thousands of packets received in one second for one application (in this case VLC), there would be a huge amount of redundant data. This can slow down capturing process and increase disk space usage. For this reason, it is better to use Custom Block which will contain application name and identification of corresponding packets. In order to use either Custom Options or Custom Block the local or registered Private Enterprise Number has to be used. Extensions will be ignored by tools that are not able to understand them.

[16] https://www.farsightsecurity.com/2015/01/28/nmsg-intro/

Chapter 4

Application Identification from Captured Packet

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*, and all important information is stored in this virtual file system. In Windows we can use *IP Helper API*[1] to retrieve information about connections and their *PIDs*, and then we can get process's command line from *Windows Management Instrumentation (WMI)*[2]. Following section describes in more detail Linux and Windows platforms, as other platforms have not been fully explored yet.

4.1 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 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 they get initialized, either at boot time or at module load time. When a user reads the file, it causes the kernel to indirectly run a set of kernel functions that return some output. The files registered by the networking code are located in `/proc/net`. The most important files are in Table 4.1. Information about individual processes can be then retrieved from files inside `/proc/[pid]/` directory [1].

4.2 Windows

The *Internet Protocol Helper API* can be used to retrieve information about the network configuration of the local computer. Using these functions one can retrieve connections table for both IPv4 and IPv6 which contains identification number of the process which opened the connection.
Process command line can be then found in *Windows Management Instrumentation (WMI)*, which is the Microsoft implementation of Web-Based Enterprise Management (WBEM).

[1] https://msdn.microsoft.com/en-us/library/windows/desktop/aa366073(v=vs.85).aspx
[2] https://msdn.microsoft.com/en-us/library/aa384642(v=vs.85).aspx

Table 4.1: Procfs files

`/proc/net/tcp6`	TCP sockets (IPv6)
`/proc/net/udp6`	UDP sockets (IPv6)
`/proc/net/udplite6`	UDPLite sockets (IPv6)
`/proc/net/raw6`	Raw sockets (IPv6)
`/proc/net/tcp`	TCP sockets
`/proc/net/udp`	UDP sockets
`/proc/net/udplite`	UDPLite sockets
`/proc/net/raw`	Raw sockets

WMI is installed by default on all Windows desktop and server platforms. It uses *Management Object Format (MOF)*[3] files to describe the information made available through their respective providers. However, certain *WMI* providers may or may not be installed, depending on the OS release and configuration. Various providers can be used to retrieve and set information about boot applications and their settings, installed applications, Domain Name System records, event log service, Hyper-V virtual machines, network adapters, system processes, Remote Desktop Services environment, drivers, performance characteristics and capabilities of a computer and much more [30]. Specifically, the *Win32 Provider*[4] retrieves data relevant to Windows system, such as settings of environment variables, characteristics of a user's desktop, time zone information, disk quota settings, but also hardware-related information like properties of a fan devices, capabilities of a floppy disk drive and more. One of the Operating System related objects provided by the *Win32 Provider* is a `Win32_Process` class which contains *Handle*, which is a process identifier and a *CommandLine* used to start the process.

4.3 Summary

Linux uses *procfs* file system where needed information is located. However, parsing of the text files and searching through all *PIDs* is slow because the kernel converts data from memory into text files and then they are converted back, in our application. In the future, *procfs* will be replaced with system calls. Lastly, on Windows, we can use *IP Helper API* functions to get information about network connections and then we can find needed process information in *WMI*. FreeBSD and MacOS platforms will be explored in the future.

[3] https://msdn.microsoft.com/en-us/library/aa823192(v=vs.85).aspx
[4] https://msdn.microsoft.com/en-us/library/aa394388(v=vs.85).aspx

Chapter 5

Implementation

The goal of the project is a multi-platform tool, which captures traffic and saves it to output file extended with information about communicating network applications associated with their packets. Determining of packet's source/destination network application is time-consuming process. To speed up this process, it is divided onto smaller tasks that can be done separately. These individual tasks are executed in threads. Threads and their roles are described in section 5.2. The project was implemented in C++ programming language. Determining network application for every packet would be very ineffective and slow. The tool uses an application cache, which holds determined applications and needed information about their network connections. Section 5.1 describes the cache and its levels.
Later sections contain implementation details of associating network applications with captured packets on Linux and Windows, and third party libraries that were used in the project.

5.1 Internal Data Representation

The application works mainly with a socket identification structure listed in Figure 5.1. Firstly, the packet direction is determined using capture interface mac address. Then local port, local IP and its version, transport-layer protocol of the captured packet and time of the packet capture are saved in this structure. The structure is then used to determine a network application and saved in a `TEntry` cache record with the results.

```
1 struct SocketId {
2     uint8_t  ipVersion;
3     void *localIp; // in_addr* or in6_addr*
4     uint16_t localPort;
5     uint8_t  proto;
6     uint64_t startTime;
7     uint64_t endTime;
8 };
```

Figure 5.1: Socket Identification structure

The application cache is used to speed up identification of network applications. After an application is successfully determined, the application 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 defines time period how long is a cache entry valid. This time is stored in every `TEntry` cache record.

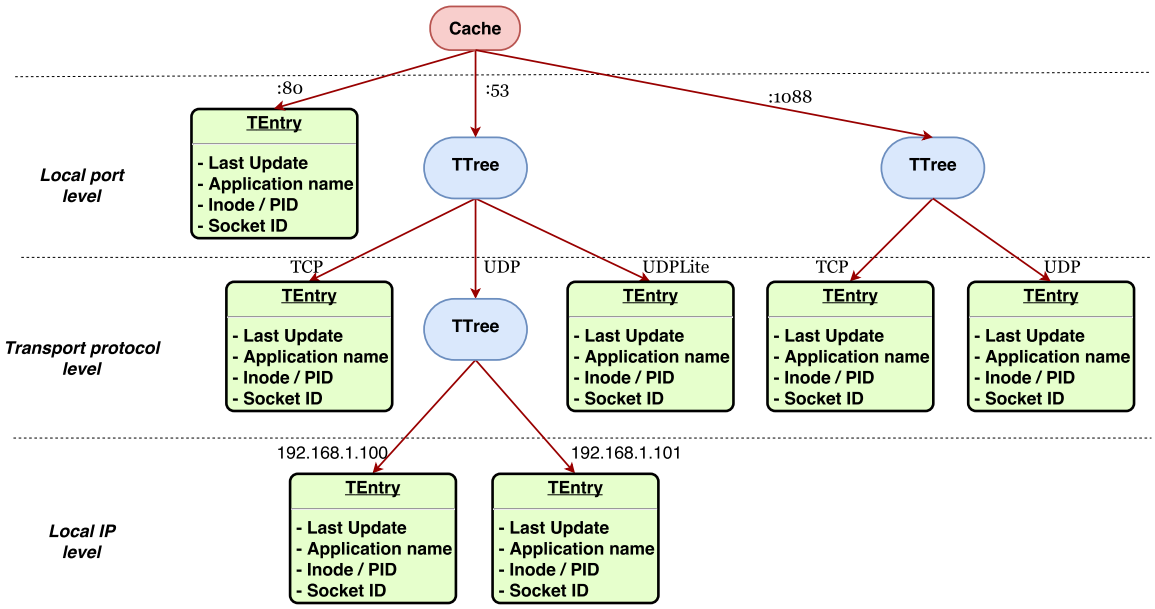


Figure 5.2: Application cache structure example

The application cache is implemented as a binary search tree and consists of three levels – local port level, local IP address level and transport protocol level. Figure 5.2 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 [14].
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 interfaces are not captured because capturing is not done in promiscuous mode. In our case, this 3-tuple is enough to identify an application's socket uniquely, so the remote side of a connection is not stored. This saves needed memory and amount of data which are being processed.

The cache consists of two types of entries. Both types contain level in the tree and identification of their type. The first type is `TEntry` record. It contains time of the last update of the record, name of the network application with its either socket's number (Linux) or PID (Windows) and identification of its socket. Structure of socket identification structure is shown in Figure 6.4 in case of IPv4 address and Figure 6.5 for IPv6 address. Both contains version of the IP address, the address itself and protocol number according to Internet Protocol Numbers assigned by IANA[1] followed by time of the first and the last packet which belongs to the application. Every packet with this 3-tuple (local IP, local port

[1] https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

```
1  while (!shouldStop)
2  {
3      char *packet = receive_packet();
4      fileBuff.push(packet);
5
6      static Netflow n;
7      parseIPLayer(&n, packet);
8      parseL4Layer(&n, packet);
9      cacheBuff.push(&n);
10 }
```

Listing 5.1: The main thread simplified code

and L3 protocol) which is within the range between the two times belongs to the stored application.

The other type is `TTree` which consists of value which is common on its level in the cache and the pointers to children of this node. Tthe first level uses `std::unordered_map`[2] to store pointers to children where the key is a local port. Lower levels use `std::vector`[3] instead, which is searched sequentially. Operations with the cache are described in section 5.2.

Cache also contains netflows for which a network application wasn't successfully determined. There can be two reasons why the application wasn't found. The first one is that when a network application's packet is captured, its socket has been already closed. In this case the application will neither be determined in the future because of closed socket. The second option is that source network application does not exist. In both cases, determining of source application for later packets is useless. Therefore this information is saved in the cache – as an empty application string. The cache improves the tool performance as the source application does not have to be determined for every packet when it is not needed.

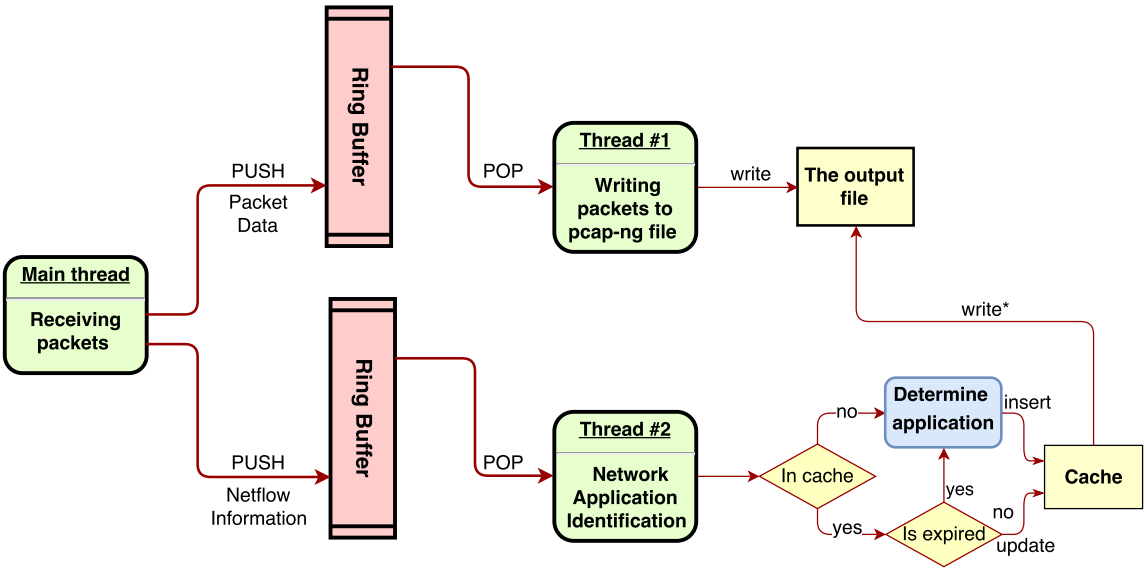
Entries from the cache are never deleted. If there is an application with the same 3-tuple like some application had before, copy of the record is inserted into a final global `map` and the application name and packet times are updated in the old record. When capture stops, the application fetches nonempty records from the application cache and inserts them to the final `map`. Records with an empty application name are ignored. This map has the same structure as the final *pcap-ng* block, and additional actions can be done with the data in the `map`, such as computing statistics, they can be sorted, filtered, etc. Currently all records are just written to the output file, and they are not further processed.

5.2 Application Design

The tool works in three threads. Their functionality is illustrated in Figure 5.3.

The main thread receives a packet, pushes it into ring buffers of the other two threads and notifies them about a new packet in their buffer. Then it can receive another packet. The other two threads wait on condition variable[4] until new packet is received. After notification, they start processing received packets until the ring buffer is empty. Then they again wait for new packets. Simplified code of the main thread is in Listing 5.1.

[2] http://en.cppreference.com/w/cpp/container/unordered_map
[3] http://en.cppreference.com/w/cpp/container/vector
[4] http://en.cppreference.com/w/cpp/thread/condition_variable



*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 5.3: Threads and their roles

```
1 while (!shouldStop)
2 {
3     wait_for_new_packet();
4     while (!fileBuff.empty())
5     {
6         write(&fileBuff[first], file);
7         fileBuff.pop();
8     }
9 }
```

Listing 5.2: Writing packets to the output file

The second thread writes all packets from the ring buffer right into the output file in *Enhanced Packet Block* format and then waits for notification about a new packet as can be seen in Listing 5.2.

The third thread determines source application for every packet (Listing 5.3). It searches in the application cache which was described in section 5.1. If an application is already in the cache and the record is not expired, it updates time of the last packet in packet's netflow which belongs to the application. Expired records are checked using `determineApp` call with `UPDATE` argument. Code of the function is pretty much the same, with difference that if identified application has changed, the cache record is moved into the global map, otherwise just the time of the last packet is updated. If the network application wasn't found in the cache, the thread tries to determine it, 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

```
while (!shouldStop)
{
    wait_for_new_packet();
    while (!cacheBuff.empty())
    {
        TEntryOrTTree *record = cache.find(&cacheBuff[first]);
        if (record != nullptr && record->isEntry())
        {
            if (!record->valid())
                determineApp(&cacheBuff[first], record, UPDATE);
            else
                record->updateEndTime(cacheBuff[first].endTime);
        }
        else
        {
            TEntry *entry = new TEntry;
            if (!determineApp(&cacheBuff[first], entry, FIND))
            {
                if (record)
                    record->insert(entry);
                else
                    cache->insert(entry);
            }
            else
                delete entry;
        }
        cacheBuff.pop();
    }
}
```

Listing 5.3: Determining of the network application

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.

`Find()` method of the cache returns either `nullptr`, `TEntry` pointer or a `TTree` pointer. If the `nullptr` is returned, it means that the record wasn't found in the top level of the cache. Otherwise, if returned value does not point to a `TEntry` record, a `TTree` node with the closest match is returned. This saves time, because the `find()` method knows to which node the new `TEntry` record will be inserted, so `insert()` method does not have to find the node with the closest match again by itself.

In the future, determining of applications will be moved to the fourth thread. The thread will use either `inotify()` to detect newly opened sockets or a system calls instead of *procfs* to find proper socket information.

5.3 Cross-platform Implementation

Most of the platform-dependent code is isolated in separate files. The right file is selected during compilation. The core of the application is written to be portable and without platform dependencies. Only differences between platforms are in obtaining of the capture inter-

face mac address and determining a source application of captured packet. The latter is described in section 5.4. Interface mac address is obtained from `/proc/class/net/<ifname>/address` file in Linux and using `GetAdaptersAddresses()`[5] method in Windows. Other platforms will be implemented in the future. Another difference is in used capture library, but code remains the same. Used libraries are described in section 5.5

5.4 Associating Network Application with Captured Packet

This section describes process of a network application identification only on supported platforms, which are currently Windows and Linux. Association of the captured packet with a network application follows the same principles on both platforms. Firstly, the packet direction is determined to find out which IP address and port are the local ones. Then L3 and L4 headers are parsed and socket identification structure is filled. Using information in this structure, the network connection endpoint is identified and after that the application command line arguments are determined. The process of the application identification is described in more detail in following sections.

5.4.1 GNU/Linux

Linux uses a virtual file-system *procfs* as was mentioned in section 4.1. Follows description of a process of the network application identification.

Identification of connection owner using received packet

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.
The most important files are in Table 4.1. They contain information about open sockets, local ports and IP addresses of connections and also their `inode` numbers. An example of the `/proc/net/tcp` file is shown in Figure 5.4.

```
sl local_address  rem_address   st   ...   uid  timeout inode
0: 00000000:006F 00000000:0000  0A   ...     0        0 10080  ...
1: 00000000:D4B3 00000000:0000  0A   ...   108        0 10167  ...
2: 00000000:0016 00000000:0000  0A   ...     0        0 14796  ...
3: 0100007F:0277 00000000:0000  0A   ...     0        0 17284  ...
4: 0100007F:0019 00000000:0000  0A   ...     0        0 14678  ...
5: 0F03000A:97B9 1D5DC036:01BB  01   ...  1000        0 18051  ...
```

Figure 5.4: `/proc/net/tcp` file example

Once we have found socket's `inode` number which corresponds to the packet's local IP address and local port, 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. File descriptors for pipes and sockets are symbolic links whose content is the file type with the inode. A `readlink()` call on a socket symbolic link return string in the format `socket:[inode]` [19]. Examples of open file descriptors are shown in Figure 5.5.

[5] https://msdn.microsoft.com/en-us/library/windows/desktop/aa365915(v=vs.85).aspx

```
ja@debian:~$ ls -l /proc/1848/fd | head
total 0
lr-x------ 1 ja ja 64 May  9 17:29 0 -> /dev/null
l-wx------ 1 ja ja 64 May  9 17:29 1 -> /dev/null
l-wx------ 1 ja ja 64 May  9 17:29 10 -> pipe:[18021]
lrwx------ 1 ja ja 64 May  9 17:29 11 -> socket:[18022]
lrwx------ 1 ja ja 64 May  9 17:29 12 -> socket:[18024]
l-wx------ 1 ja ja 64 May  9 17:29 13 -> /home/ja/.mozilla/firefo...
lrwx------ 1 ja ja 64 May  9 17:29 14 -> anon_inode:[eventpoll]
lrwx------ 1 ja ja 64 May  9 17:29 15 -> socket:[18026]
lrwx------ 1 ja ja 64 May  9 17:29 16 -> socket:[18027]
```

Figure 5.5: Different file descriptors example

Associate PID with the process name

Next step is to find the process command line arguments. 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 [19].

5.4.2 Windows

On Windows, one can use *IP Helper API* to retrieve network connections. While on Linux we got socket's *inode* number from table of network connections, on Windows it contains directly *PID* of the owners process. Following sections contain more detailed description.

Identification of connection owner using packet

Functions `GetExtendedTcpTable()` and `GetExtendedUdpTable()` returns table of network connections. When the functions are called with `TCP_TABLE_OWNER_PID_ALL` (TCP connections) or `UDP_TABLE_OWNER_PID` (UDP connections) as *TableClass* argument, row entry in the returned table contains also *PID* of the process that called the `bind()` function. Once we have found *PID* which corresponds to the packet's local IP address and local port, we have to determine process command line arguments.

Associate PID with the process name

Process's command line arguments are stored in *WMI*, which was described in section 4.2. *WMI* is based on *Component Object Model (COM)*[6] technology, so firstly we must initialize *COM* interface using `CoInitializeEx` and `CoInitializeSecurity` calls.
Next step is to create a connection to a WMI namespace. In this step a proxy to an *IWbemServices* interface is received that is used to access the local or remote WMI namespace. After we retrieve a pointer to an *IWbemServices* proxy, we must set the security on the proxy because the *IWbemService* proxy grants access to an out-of-process object. In general, *COM* security does not allow one process to access another process if the proper security properties are not set.
Now various capabilities of WMI can be accessed. In our case, we retrieve a WMI instance of `Win32_Process`[7] class which represents a process on an operating system. This

[6] https://msdn.microsoft.com/en-us/library/ms680573(v=vs.85).aspx
[7] https://msdn.microsoft.com/en-us/library/aa394372(v=vs.85).aspx

class contains `handle`, which is process's PID and its `CommandLine` arguments in a string. A WMI instance is retrieved using `SELECT` query[8], so we can retrieve just an instance with a specified PID.
After using *WMI* is finished, all open *COM* interfaces must be released, `CoUninitialize` must be called and WMI application must be shut down. Information in this section were taken from [31] where also example codes of WMI application can be found.

5.5 Used Libraries

The application uses `libpcap` libprary to capture packets, described in section 2.2.3. On Windows the `Npcap` library is used. As was described in section 2.2.4, `Npcap` is fully compatible with `libpcap` API. `Npcap` was chosen because of support of the newest Windows 10 and the latest `libpcap`, and because it is still maintained. `PcapPlusPlus`[9] is a multi-platform C++ network sniffing and packet parsing framework with support of `PF_RING`, `DPDK` and `pcap-ng`. It was considered to be used as a capture library but it is approximately 3 times slower than `libpcap`, so finally it was not used.
To speed up capture process in Linux, the latest `PF_RING` library (05/08/2017) version was used during tests. Results compared with capturing without `PF_RING` are in chapter 7. In order to use `PF_RING`, the tool has to be compiled and linked with `PF_RING` library and its modified `libpcap` library. `Netmap` and `PFQ` were also considered, but I wasn't able to compile `PFQ` in none of Ubuntu 14.04.5 LTS, Ubuntu 16.04.2 LTS, Debian 8 Jessie, Kali 2016.1 and neither in the latest Kali Nightly build distribution. `Netmap` is available for FreeBSD, Linux and also Windows. In the future it will be tested and considered as a replacement of the `libpcap` library.

The tool currently supports both `Netmap` and `PFQ` prefixes and also `Netmap`'s interface postfixes, so if these libraries are installed on the system, and the tool is compiled with these libraries it can also work with them.

5.6 Summary

The application uses cache to speed up identification process by reducing unneeded application searches. The tool job is divided into smaller tasks which are executed in three threads. Most of the platform-dependent code is located in separate files so in the future it will be easy to extend support of new platforms. Work with *procfs* file system in Linux is slower than it was expected and in the future, in the application either a new thread with `inotify()` will be added or *procfs* will be avoided completely. I want also make some test using *Netmap* API and maybe it will be used instead of libpcap.

[8] https://msdn.microsoft.com/en-us/library/aa393243(v=vs.85).aspx
[9] http://seladb.github.io/PcapPlusPlus-Doc/

Chapter 6

Output File

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 6.1.

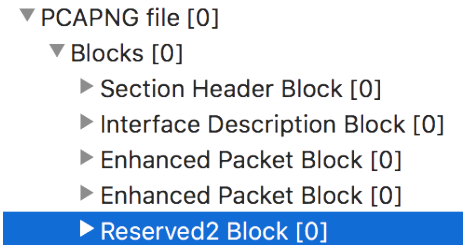


Figure 6.1: Structure of the pcap-ng file

The block contains mandatory fields, namely type of the block, its length and identification PEN number, and then follow application tags. Every recognized application has one `Application Tag` even if it has multiple connections opened. The size of each `Application Tag` depends on application name length, number of sockets the application opened and if they were IPv4 or IPv6. Exact Custom Block structure with sizes of all fields is in Figure 6.2. The last `Application Tag` is followed by padding to 32 bits.

```
1 struct CustomBlock {
2     uint32_t blockType = 0x40000BAD;
3     uint32_t blockTotalLength;
4     uint32_t PrivateEnterpriseNumber = 0x1234;
5     AppTag appTags[n];
6     /* n == (blockTotalLength - 16) / sizeof(AppTag); */
7     // Padding to 32 bits
8     uint32_t blockTotalLength2;
9 };
```

Figure 6.2: Pcapng - Custom Block

The Custom Block can contain multiple `Application Tags`. Every tag contains a name of the application with its arguments preceded by their length, number of records for the application and records themselves (Figure 6.3). The first field is a length of the recognized application with its arguments including ending '\0'. Next follows the application's name. Application name and its arguments are delimited by null bytes. In Figure B.5 looks that

it contains application names without their arguments. This is caused by the software that was used to read the output *pcap-ng* file. There is specified, in the *pcap-ng* grammar used by the tool, that the `App name` field is `appnameLen` bytes long, but despite this, the software ends reading the string at the first NULL byte. Application arguments can be seen in a hexadecimal view on the left side of the figure.

```
1 struct AppTag {
2     uint8_t appnameLen;
3     char appname[appnameLen];
4     uint32_t records;
5     SocketId socketRecords[records];
6 };
```

Figure 6.3: Custom Block - Application Tag

An `Application Tag` can contain several `SocketId` records. Each `SocketId` record identifies a group of packets which was sent using the same socket, and thus they belong to one application.

```
1 struct SocketId {
2     uint8_t  ipVersion = 4;
3     uint32_t localIp;
4     uint16_t localPort;
5     uint8_t  proto;
6     uint64_t startTime;
7     uint64_t endTime;
8 };
```

Figure 6.4: Custom Block - Socket Identification (IPv4)

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 [14]. The size of the record depends on IP address version. Exact structure is shown in Figure 6.4 for IPv4 sockets and Figure 6.5 for IPv6 sockets. The `startTime` and `endTime` values contain the Unix time in microseconds, same as is stored in `timestamp` in an Enhanced Packet Block (see section 3.2.2). Real output file example with values is shown in Appendix B.

```
1  struct SocketId {
2      uint8_t  ipVersion = 6;
3      union {
4          uint8_t   addr8[16];
5          uint16_t  addr16[8];
6          uint32_t  addr32[4];
7      } addr;
8      uint16_t localPort;
9      uint8_t  proto;
10     uint64_t startTime;
11     uint64_t endTime;
12 };
```

Figure 6.5: Custom Block - Socket Identification (IPv6)

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 [28].

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 tags at the end of the file or not. *PEN* is recently set to the `0x1234` value, but in the future, it will be updated with the value assigned by the IANA.

Chapter 7

Testing

Implemented application was tested on *Intel Core i5-3570K 3.4GHz* with *Intel Corporation 82579V Gigabit Network Connection* adapter running *lubuntu 17.04* and *Windows 7*. The tool efficiency depends on several factors, such as validity time set in the cache, size of used ring buffers, collisions in the cache, number of communicating network applications, frequency of their communication, etc. Follow performance tests, which shows dependence of performance on packet size on Linux and Windows. Tests were made in the topology shown in Figure 7.1 where two computers were connected through switch. Network traffic was captured on the server side.

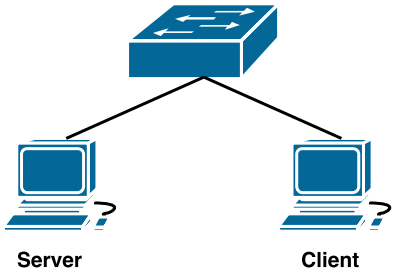


Figure 7.1: Tests network topology

Performance tests

Figure 7.2 shows an amount of data which can be processed in real-time using standard `libpcap` and using `libpcap` with `PF_RING` on Linux. In this case, just one network application communicated. In the figure, we can see that the Linux version of the tool can capture and save traffic without any packet loss, and it can identify one network application without packet loss at 1 Gbps speed with packets around 200 B and bigger. Tests were run five times and their average value was used as the result. Searching in the `procfs` will be sped up in the future.

Figure 7.3 shows same tests run on the same machine, but on Windows 7 Professional. Here, the results are not so good, and it has not been fully explored yet what is the slowest part of the network application identification process. In the figure, there can be seen that the tool either wasn't able to write captured packets to the hard drive with the same speed which packets were generated. Tests were run five times and their average value was used

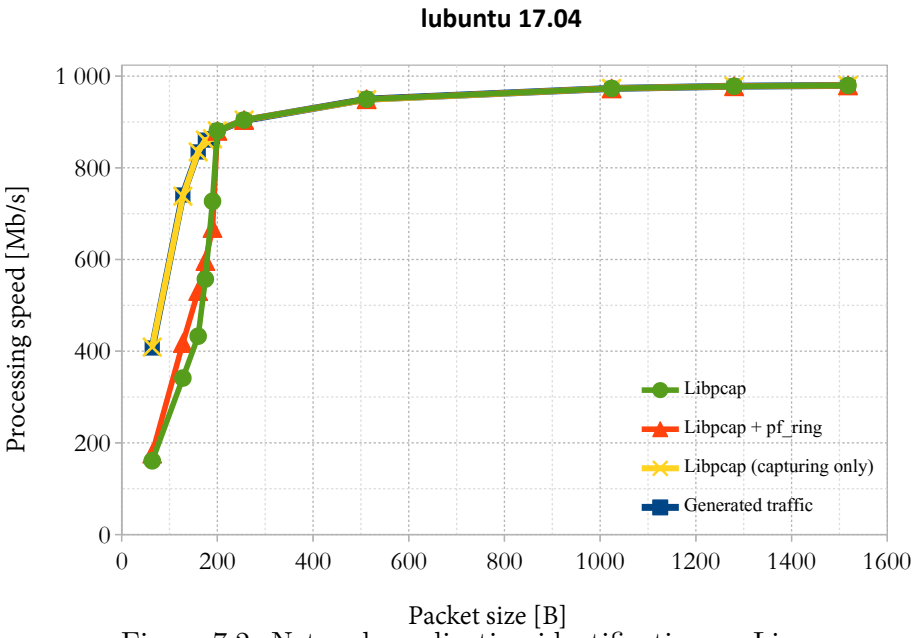


Figure 7.2: Network application identification on Linux

as the result. Unfortunately, Windows drivers weren't able to handle outgoing traffic over 100 Mbps, which affected test results. We can say that on Windows, the tool can handle packets bigger than 1000 kB at speed over 400 Mbps. For exact numbers, tests have to be repeated with different network adapter.

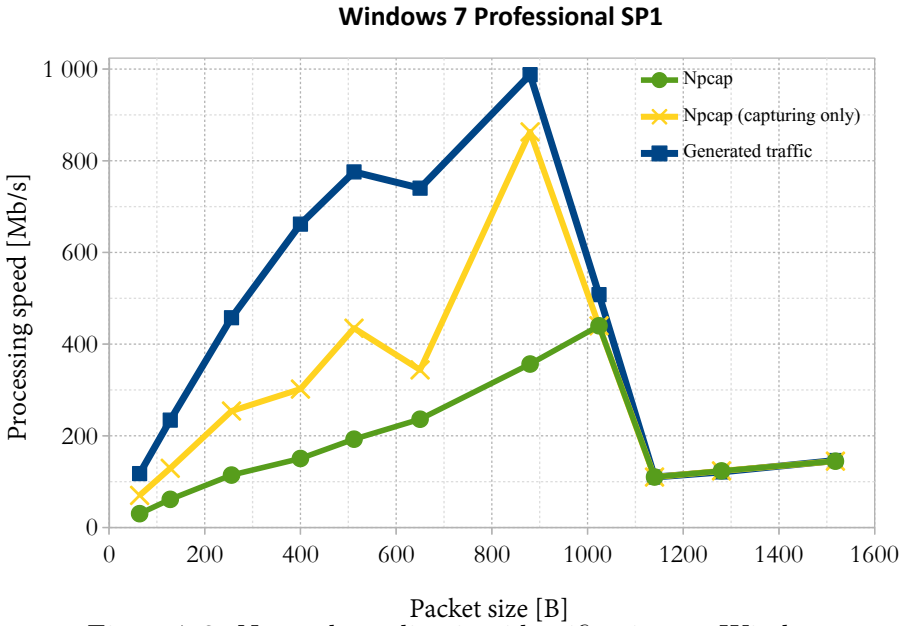


Figure 7.3: Network application identification on Windows

Finally, Figure 7.4 shows performance decrease in case of more communicating network applications. It necessarily doesn't have to be more applications, it also can be one application with multiple opened sockets. Tests were made running multiple instances of simple UDP server.

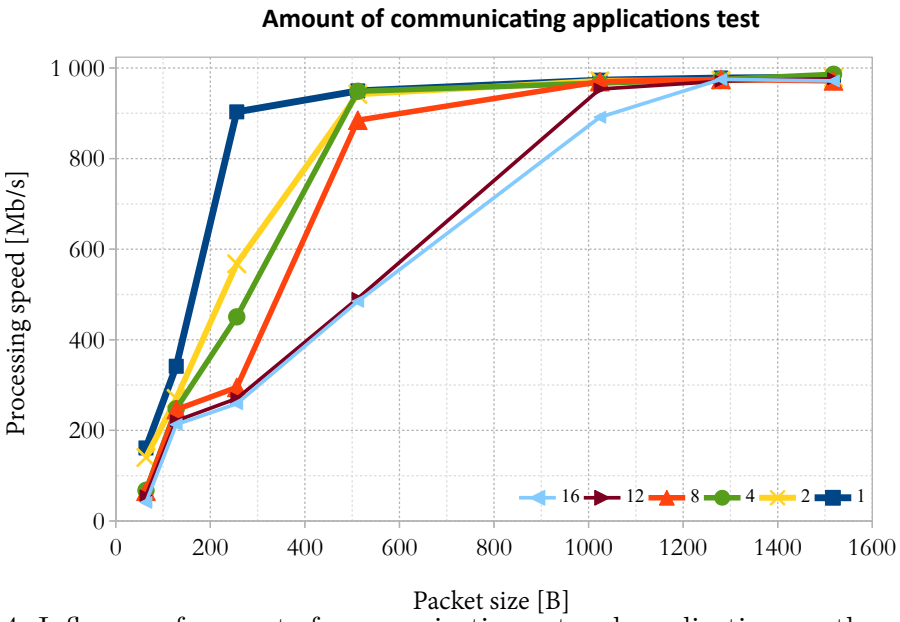


Figure 7.4: Influence of amount of communicating network applications on the performance

Validity of implementation

Table 7.1 shows reliability of network traffic capture. It contains number of generated packets by the client, number of received packets on the server's side and number of captured packets. This test wasn't done to show exact amount of lost packets, but to validate that all packets are successfully captured when there are no packets dropped. As you can see, no packets were lost because of implementation fault.

Table 7.1: Packet capture reliability, lubuntu 17.04

Payload	Generated bps	Sent packets	Received packets	Captured packets	Packet loss
64	96 M	1304926	1304926	1304926	0 %
64	164 M	3377233	3377233	3377233	0 %
256	82 M	478908	478908	478908	0 %
256	378 M	1237085	1237085	1237085	0 %
256	741 M	2360585	2360585	2360585	0 %
512	104 M	128220	128220	128220	0 %
512	463 M	945186	945186	945186	0 %
512	804 M	1759830	1759830	1759830	0 %
1024	140 M	117986	117986	117986	0 %
1024	403 M	258220	258220	258220	0 %
1024	925 M	1040582	104582	104582	0 %
1518	104 M	116783	116783	116783	0 %
1518	421 M	392115	392115	392115	0 %
1518	980 M	522936	522936	522936	0 %

Chapter 8

Conclusion

Most systems use *sockets* application programming interface. Normally, *sockets* get data without protocol headers of the lower layers. Network sniffers cannot work without these headers, as they contain the most important information. *Raw sockets* can be used to receive all packets, but the whole packet is still not passed to the application. To retrieve a copy of the whole packet, access to the datalink layer can be used. Different platforms use various techniques how to access the datalink layer. Tcpdump had several multi-platform applications, so they decided to make a portable library with a single API on all platforms. We can use this *libpcap* library which is portable and hides the platform dependencies. We can also use frameworks like *DPDK* or *PF_RING* to speed up capture process, which use additional buffers, modified drivers and other methods in order to reduce packet loss.

To save captured traffic with additional information, we need an extendable file format. *Pcap-ng* was developed to improve old *pcap* file format. This format is extendable in two ways. A `Custom Option` can be added to each block, but in our case, this option cannot be used because of caused data redundancy. The other option is to use `Custom Block`. This block can contain any data, and it doesn't have to be in every block. To use either `Custom Option` or `Custom Block`, the PEN number has to be registered. This number can be a random number or registered by IANA.

The goal of this project was a tool that would be multi-platform and could process 1 Gbps links with minimal packet loss. The only application with desired functionality is *Microsoft Network Monitor*, but it is only for Windows and uses undocumented Netmon capture file format.

In my 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.

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.

Thanks to the use of pcap-ng file format, applications that support this format will be, after small modification, able to process also block with information about communicating applications.

Currently, the application works on Linux for both IPv4 and IPv6 connections and Windows for IPv4 connections. The application is implemented in C++ programming language. 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 1 Gbps traffic only with packets bigger than 200 Bytes in Linux. Although the main core of the application could process all packets at speed, they were generated, 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, and thus the application cannot be determined. The Windows version is even slower and IPv6 functionality has not been fully implemented yet. Connection table on Windows does not contain information about UDP clients. Solution to this problem will be explored in the future.

In the future, the application will be implemented on other platforms, and new ways how to speed up application identification process on currently supported platforms will be explored. Further, *netmap* API will be tried, because "using the card in *netmap* mode and bridging in software is often more efficient than using standard mode, because the driver uses simpler and faster code paths" [24].

This research was introduced at a student conference on innovation and technology in computer science *Excel@FIT 2017*[1]. The paper presented at the conference is enclosed in Appendix C.

[1] http://excel.fit.vutbr.cz

Bibliography

[1] Benvenuti, C.: *Understanding Linux Network Internals: Guided Tour to Networking on Linux*. O'Reilly Media. 2005. ISBN 9780596552060.

[2] Bonelli, N.: Software Acceleration for Network Applications and multi-core architectures. In *NetV: IRISA / Technicolor Workshop on Network Virtualization*. February 2017.
http://www.bretagne-networking.org/wp-content/uploads/NetV_Bonelli.pdf.

[3] Bonelli, N.; Giordano, S.; Procissi, G.: Network Traffic Processing With PFQ. *IEEE Journal on Selected Areas in Communications*. vol. 34, no. 6. June 2016: pp. 1819–1833. ISSN 0733-8716. doi:10.1109/JSAC.2016.2558998.

[4] Bonelli, N.; Giordano, S.; Procissi, G.; et al.: A Purely Functional Approach to Packet Processing. In *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS '14. New York, NY, USA: ACM. 2014. ISBN 978-1-4503-2839-5. pp. 219–230. doi:10.1145/2658260.2658269.
Retrieved from: http://doi.acm.org/10.1145/2658260.2658269

[5] Camaró, U.A.; Baudy, J.: Packet_mmap.
https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt.
[Online; visited on 01/28/2017].

[6] Degioanni, L.: *Development of an Architecture for Packet Capture and Network Traffic Analysis*. PhD. Thesis. Politecnico di Torino Facoltà di Ingegneria, Corso di Laurea in Ingegneria Informatica. March 2000.
https://www.winpcap.org/docs/th_degio.zip.

[7] Deri, L.: nCap: Wire-speed packet capture and transmission. In *End-to-End Monitoring Techniques and Services, 2005. Workshop on*. IEEE. 2005. pp. 47–55.

[8] Gregg, B.: Linux 4.x performance: Using BPF superpowers.
http://www.brendangregg.com/blog/2016-03-05/linux-bpf-superpowers.html.
March 2016. performace @Scale 2016.

[9] Haryachyy, D.: Understanding DPDK. SlideShare. February 2015.
http://www.slideshare.net/garyachy/dpdk-44585840.

[10] Insolvibile, G.: Inside the Linux Packet Filter, Part II. Web page. March 2012.
https://www.linuxjournal.com/article/5617?page=0,0.

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

[12] Liu, C.; Wang, L.; Yang, A.: *Information Computing and Applications: Third International Conference, ICICA 2012, Chengde, China, September 14-16, 2012. Proceedings*. Number part 2 in Communications in Computer and Information Science. Springer Berlin Heidelberg. 2012. ISBN 9783642340413.

[13] McCanne, S.; Jacobson, V.: The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, vol. 93. 1993.

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

[15] ntop: PF_RING - ntop. http://www.ntop.org/products/packet-capture/pf_ring/. [Online; visited on 01/28/2017].

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

[17] *packet(7) Linux Programmer's Manual*. October 2016. http://man7.org/linux/man-pages/man7/packet.7.html.

[18] *Misc. Reference Manual Pages (3PCAP)*. January 2017. http://www.tcpdump.org/manpages/pcap.3pcap.html.

[19] *proc(5) Linux Programmer's Manual*. September 2014. https://linux.die.net/man/5/proc.

[20] Rago, S.: *UNIX System V Network Programming*. Addison-Wesley Professional Computing Series. Pearson Education. 1993. ISBN 9780133893014.

[21] Risso, F.; Degioanni, L.: An architecture for high performance network analysis. In *Proceedings. Sixth IEEE Symposium on Computers and Communications*. IEEE. 2001. ISSN 1530-1346. pp. 686–693. doi:10.1109/ISCC.2001.935450.

[22] Rizzo, L.: Netmap: a novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*. 2012. pp. 101–112.

[23] Rizzo, L.; Deri, L.; Cardigliano, A.: 10 Gbit/s Line Rate Packet Processing Using Commodity Hardware: Survey and new Proposals. 2011. http://luca.ntop.org/10g.pdf.

[24] Rizzo, L.: The netmap project. http://info.iet.unipi.it/~luigi/netmap/. [Online; visited on 01/28/2017].

[25] Sanders, C.; Smith, J.: *Applied Network Security Monitoring: Collection, Detection, and Analysis*. Elsevier Science. 2013. ISBN 9780124172166.

[26] Comparing Ring-buffer–based Packet capture solutions. http://prod.sandia.gov/techlib/access-control.cgi/2015/159378r.pdf. 2017. [Online; visited on 05/05/2017].

[27] Stevens, W.; Fenner, B.; Rudoff, A.: *UNIX Network Programming*. Number vol. 1 in Addison-Wesley professional computing series. Addison-Wesley. 2004. ISBN 9780131411555.

[28] Tuexen, M.; Risso F.; Bongertz J.; Combs G.; Harris G.: 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.

[29] Wireshark: Development/LibpcapFileFormat - The Wireshark Wiki. https://wiki.wireshark.org/Development/LibpcapFileFormat. [Online; visited on 01/28/2017].

[30] About WMI. Web page. https://msdn.microsoft.com/en-us/library/aa384642(v=vs.85).aspx.

[31] Creating a WMI Application Using C++. Web page. https://msdn.microsoft.com/en-us/library/aa389762(v=vs.85).aspx.

[32] Wright, G.; Stevens, W.: *TCP/IP Illustrated*. Number vol. 2 in Addison-Wesley Professional Computing Series. Pearson Education. 1995. ISBN 9780321617644.

[33] yusuf@linux: Linux Unleashed: Raw socket, Packet socket and Zero copy networking in Linux. http://yusufonlinux.blogspot.cz/2010/11/data-link-access-and-zero-copy.htm. [Online; visited on 05/05/2017].

Appendices

List of Appendices

Appendix A

CD Content

Enclosed CD contains following files and directories:

- `latex/` – LaTeXsource files
- `BP/` – The tool source files
- `xzuzel00-BP.pdf` – PDF version of the Bachelor's thesis
- `README` – Text file containing instructions how to compile the tool
- `52_clanek.pdf` – The paper used at Excel@FIT conference
- `52_poster.pdf` – The poster used to present the research at Excel@FIT

Appendix B

Example Pcap and Pcap-ng Blocks

Position	Offset	Length	Index	Element	Value
0x00	0	253144	0	▼ LibPCAP File [0]	
0x00	0	24	0	▼ File Header [0]	
0x00	0	4	0	magic_number	LIttle Endian: 0xD4C3B2A1
0x04	+4	2	1	version_major	2
0x06	+6	2	2	version_minor	4
0x08	+8	4	3	thiszone	0
0x0C	+12	4	4	sigfigs	0
0x10	+16	4	5	snaplen	65535
0x14	+20	4	6	network	1
0x18	+24	58	1	▼ Packets [0]	
0x18	0	16	0	▼ Packet Header [0]	
0x18	0	4	0	frame.time	1210953938
0x1C	+4	4	1	frame.time_usec	216729
0x20	+8	4	2	frame.len	42
0x24	+12	4	3	frame.cap_len	42
0x28	+16	42	1	▶ Packet Data [0]	
0x52	+82	76	2	▶ Packets [1]	
0x9E	+158	90	3	▶ Packets [2]	
0xF8	+248	90	4	▶ Packets [3]	
0x152	+338	82	5	▶ Packets [4]	
0x1A4	+420	223	6	▶ Packets [5]	
0x283	+643	371	7	▶ Packets [6]	

Figure B.1: PCAP - Global Header is located at the start of the file. Packet Header-Data pairs follow immediately after the Global Header.

Offset	Length	Index	Element	Value
0	17402972	0	▼ PCAPNG file [0]	
0	17402972	0	▼ Blocks [0]	
0	148	0	▼ Section Header Block [0]	
0	4	0	BlockType	SectionHeader: 0xA0D0D0A
+4	4	1	TotalLength	0x94
+8	136	2	▼ Body [0]	
0	4	0	Byte-OrderMagic	0x1A2B3C4D
+4	2	1	MajorVersion	1
+6	2	2	MinorVersion	0
+8	8	3	SectionLength	-1
+16	120	4	▼ Options [0]	
0	48	0	▼ OS Option [0]	Mac OS X 10.12, build 16A323 (Darwin 16.0.0)
0	2	0	Code	OS: 0x3
+2	2	1	Length	44
+4	44	2	▼ Value [0]	Mac OS X 10.12, build 16A323 (Darwin 16.0.0)
0	44	0	OS	Mac OS X 10.12, build 16A323 (Darwin 16.0.0)
+48	68	1	▼ User Application Option [0]	Dumpcap (Wireshark) 2.2.0 (v2.2.0-0-g5368c50 from master-2.2)
0	2	0	Code	userappl: 0x4
+2	2	1	Length	61
+4	61	2	▼ Value [0]	Dumpcap (Wireshark) 2.2.0 (v2.2.0-0-g5368c50 from master-2.2)
0	61	0	Application	Dumpcap (Wireshark) 2.2.0 (v2.2.0-0-g5368c50 from master-2.2)
+65	3	3	Padding	00 00 00
+116	4	2	▼ End of Options [0]	
0	2	0	Code	End of Options: 0x0
+2	2	1	Length	0
+144	4	3	TotalLength2	0x94
+148	88	1	▶ Interface Description Block [0]	
+236	1392	2	▶ Enhanced Packet Block [0]	01 00 5E 16 24 79 B8 AF 67 82 7D AA ...

Figure B.2: Pcap-ng - Section Header Block

Offset	Length	Index	Element	Value
0	17402972	0	▼ PCAPNG file [0]	
0	17402972	0	▼ Blocks [0]	
0	148	0	▶ Section Header Block [0]	
+148	88	1	▼ Interface Description Block [0]	
0	4	0	BlockType	0x00000001: 0x1
+4	4	1	TotalLength	0x58
+8	76	2	▼ Body [0]	
0	2	0	LinkType	LINKTYPE_ETHERNET: 1
+2	2	1	Reserved	0
+4	4	2	SnapLen	262144
+8	68	3	▼ Options [0]	
0	8	0	▼ Interface Name Option [0]	en3
0	2	0	Code	if_name: 0x2
+2	2	1	Length	3
+4	3	2	▼ Value [0]	en3
0	3	0	Name	en3
+7	1	3	Padding	00
+8	8	1	▼ Interface Timestamp Resolution...	
0	2	0	Code	if_tsresol: 0x9
+2	2	1	Length	1
+4	1	2	▼ Value [0]	
0	1	0	Resolution	0x6
+5	3	3	Padding	00 00 00
+16	48	2	▼ Interface OS Option [0]	Mac OS X 10.12, build 16A323 (Darwin 16.0.0)
0	2	0	Code	if_os: 0xC
+2	2	1	Length	44
+4	44	2	▼ Value [0]	Mac OS X 10.12, build 16A323 (Darwin 16.0.0)
0	44	0	OS	Mac OS X 10.12, build 16A323 (Darwin 16.0.0)
+64	4	3	▼ End of Options [0]	
0	2	0	Code	End of Options: 0x0
+2	2	1	Length	0
+84	4	3	TotalLength2	0x58
+236	1392	2	▶ Enhanced Packet Block [0]	01 00 5E 16 24 79 B8 AF 67 82 7D AA ...

Figure B.3: Pcap-ng - Interface Description Block

Offset	Length	Index	Element	Value
0	17402972	0	▼ PCAPNG file [0]	
0	17402972	0	▼ Blocks [0]	
0	148	0	▶ Section Header Block [0]	
+148	88	1	▶ Interface Description Block [0]	
+236	1392	2	▼ Enhanced Packet Block [0]	01 00 5E 16 24 79 B8 AF 67 82 7D AA ...
0	4	0	BlockType	0x00000006: 0x6
+4	4	1	TotalLength	0x570
+8	1380	2	▼ Body [0]	01 00 5E 16 24 79 B8 AF 67 82 7D AA ...
0	4	0	InterfaceID	0
+4	4	1	Timestamp (High)	343475
+8	4	2	Timestamp (Low)	2354128032
+12	4	3	CapturedLength	1358
+16	4	4	PacketLength	1358
+20	1358	5	PacketData	01 00 5E 16 24 79 B8 AF 67 82 7D AA ...
+1378	2	6	Padding	00 00
+1388	4	3	TotalLength2	0x570
+1628	1392	3	▶ Enhanced Packet Block [0]	01 00 5E 16 24 79 B8 AF 67 82 7D AA ...

Figure B.4: Pcap-ng - Enhanced Packet Block

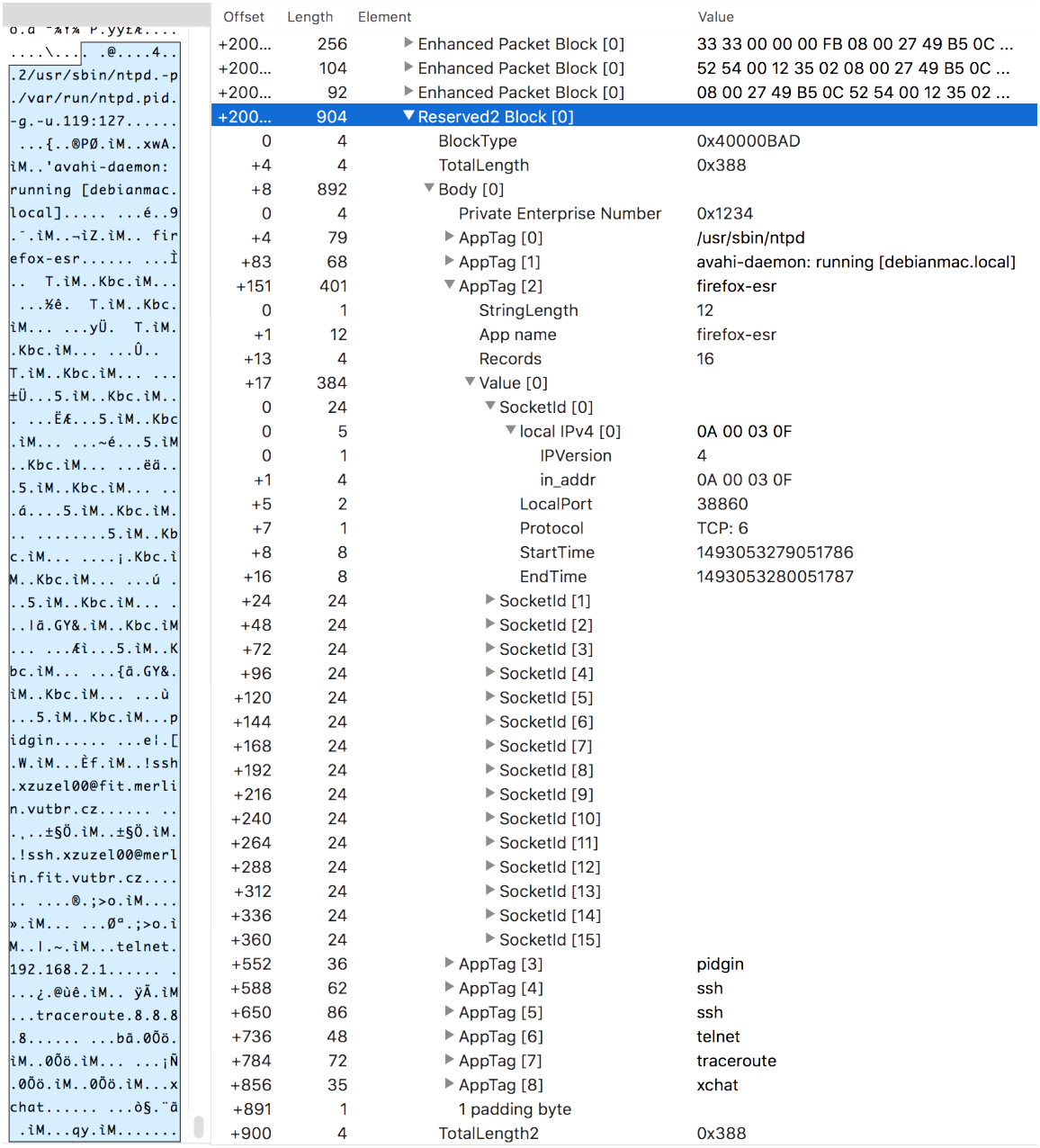


Figure B.5: Pcap-ng - Custom block structure

Appendix C

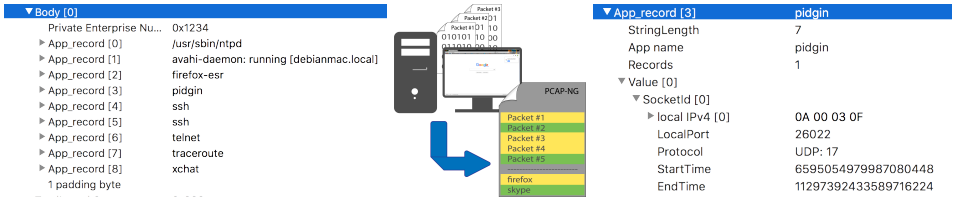
Paper Presented at Excel@FIT 2017 Conference

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 research 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.

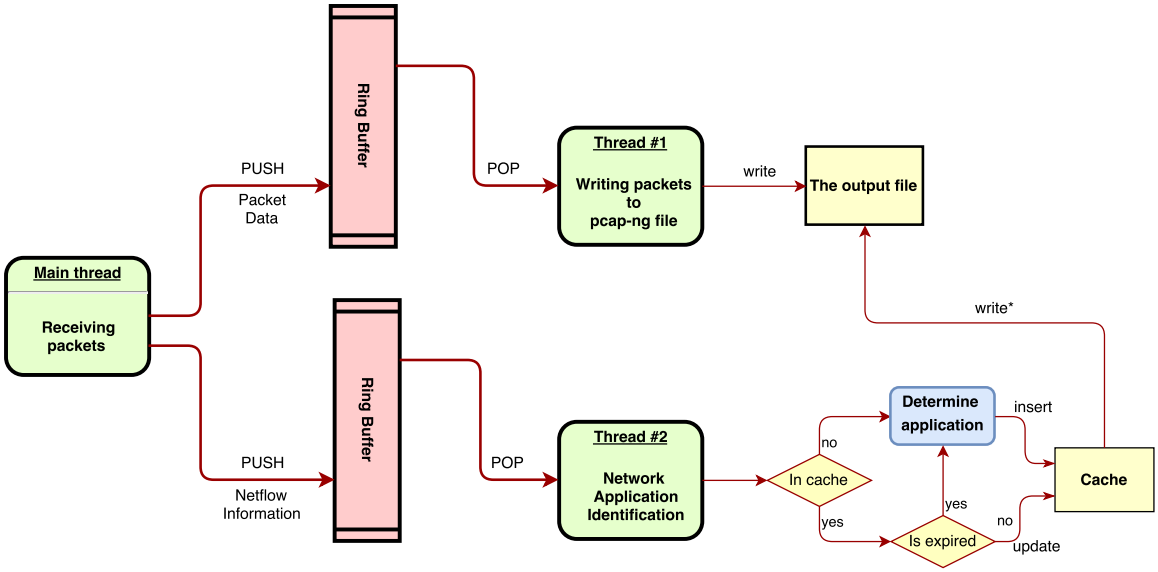
2.1 Capture speed up

`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

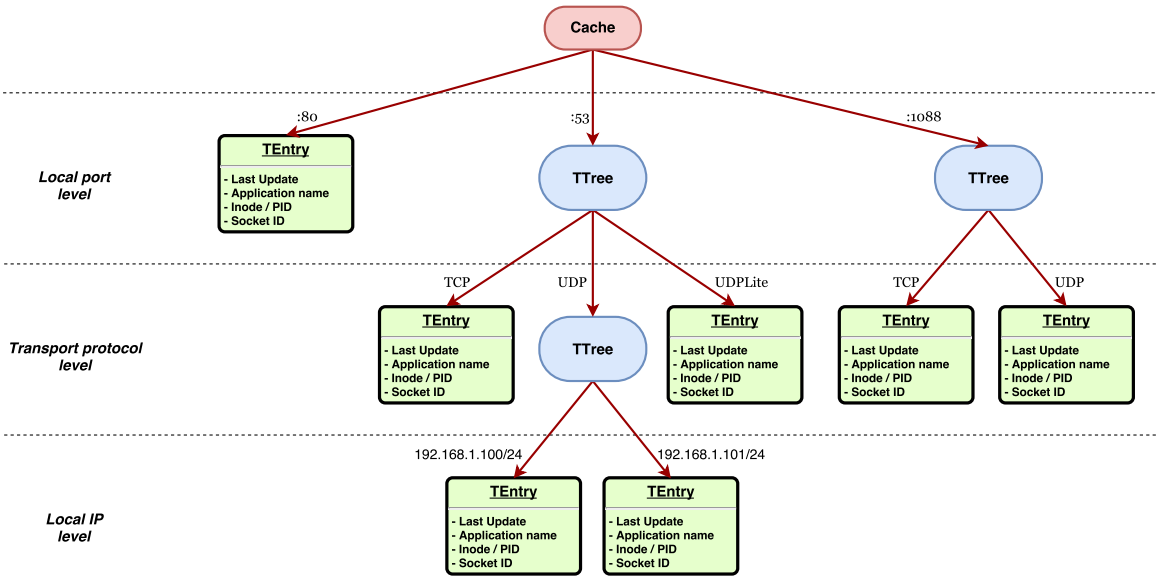


Figure 2. Application cache structure example

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, local ports and IP addresses of connections and also their `inode` numbers. Once we had found 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.

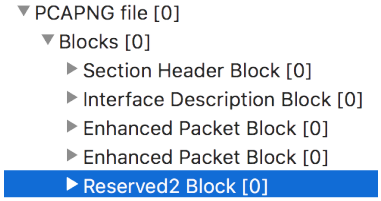


Figure 3. Structure of the pcap-ng file

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 Intel Core i5 (I5-5257U) 2,7GHz with Broadcom BCM5701 Ethernet adapter 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.

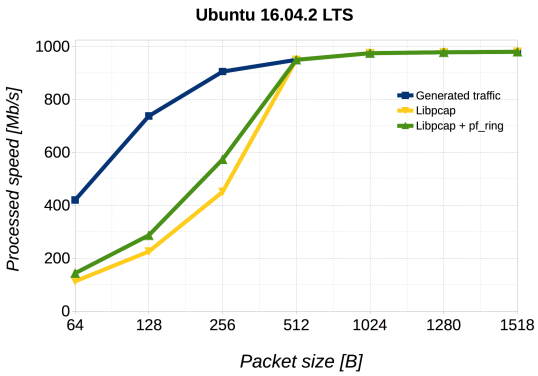


Figure 7. Network application identification on Linux

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.