MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Network Traffic Analysis with Deep Packet Inspection Method

MASTER'S THESIS

Jakub Svoboda

Brno, Spring 2014

Declaration

Hereby I declare that this thesis is my original authorial work, which I have worked out by my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jakub Svoboda

Advisor: Ing. Pavel Čeleda, Ph.D.

Acknowledgement

I'd like to thank to my advisor Ing. Pavel Čeleda, Ph.D. for patiency, guidance, assistance and encouragement and to Tomáš Plesník for support in setting up the test environment.

Abstract

This thesis deals with network traffic monitoring. We provide an overview of existing network traffic monitoring approaches. Notable software implementations of said approaches are described. The network traffic monitor best matching specified criteria is chosen. A network traffic analysis method is implemented for the chosen network traffic monitor. The analysis method is used to show the differing behavior of two distinct network traffic monitoring approaches (deep packet inspection and flow monitoring). Properties of the chosen network traffic monitor, along with performance measurements, are discussed.

Keywords

Bro Network Security Monitor, DPI, Deep Packet Inspection, IPFIX, Network Flow, Network Topology Mapping Method

Contents

List of Figures

License Notice

Chapter 1

Introduction

Two principal methods of network traffic monitoring exist — flow-based monitoring and deep packet inspection. The Institute of Computer Science of Masaryk University (ICS MU) has long history of using flow-based monitoring. This thesis aims to give an overview of deep packet inspection with the goal to extend traffic analysis capabilities at ICS MU. Hence, this work is intended for readers familiar with flow-based monitoring, keen to learn about deep packet inspection.

This thesis focuses on the Bro Network Security Monitor and is divided in the following chapters.

We present two major approaches to network traffic monitoring in Chapter 2. Varying differentiations of the approaches are explained.

Chapter 3 describes deep packet inspection (DPI) and analyzes its parts. There are multiple approaches to DPI and we show differences between them. The last section of the chapter discusses selection of the most suitable DPI implementation. We chose Bro as the most promising instance of DPI.

Chapter 4 introduces a network topology mapping method. The method is used to show properties of DPI in practice, as well as to compare properties of two versions of the method — one with DPI and one that is flow-compliant. Thus, the principles described theoretically in chapters two and three are implemented and evaluated. Various aspects of the method show the importance of selected capabilities in network traffic monitoring tools. DPI and extensibility are the two most important ones. Performance measurements show how Bro compares with a flow-based network traffic monitor. The chapter closes with summary of the most important findings.

The final, fifth, chapter presents conclusions about deep packet inspection and the Bro Network Security Monitor.

Chapter 2

Network Monitoring Approaches

Network monitoring can be either active or passive. Passive network monitoring reads data from the line, without affecting the traffic. Active network monitoring adds option to modify the data on the line [1]. This thesis deals with passive network monitoring.

Passive network monitoring exists in several forms. Simple monitoring may be easy for manual assessment as the amount of data monitored and produced is small. However, faults and attackers can slip through undetected. Monitoring of all sorts of details about the network and its traffic bears a similar hurdle — information about faults and attackers are gathered, but there is so much information that it gets lost in the sea. Also, the more data captured, the more technologically demanding it is to save and handle the data. Therefore, various ways of doing network monitoring compete with each other, as each has different tradeoffs, being targeted for different purposes, environments, and users.

This chapter looks at existing approaches to network traffic monitoring, their architecture, and properties.

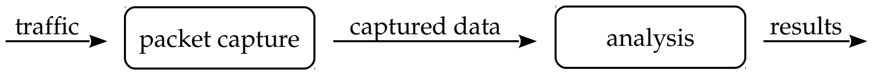

Figure 2.1: General architecture of network monitoring.

The process of network monitoring consists of two major steps — traffic duplication and traffic analysis. Section 2.1 describes traffic duplication. Section 2.2 describes various approaches to the analysis.

2.1 Traffic Duplication

All types of network traffic monitoring have one common property — the traffic from the line is duplicated so that the copy can be analyzed. The duplication can take place in one of two modes — inline or mirroring [2]. A traffic duplication device in inline mode is placed in link. In mirroring mode, the duplication facility is already a built-in feature of a router or switch.

There are several ways of traffic mirroring; port mirroring, TAP and a TAP-like setup using bypass NICs. The following subsections describe each way.

2.1.1 Port Mirroring

Port mirroring is a functionality usually available in enterprise-oriented network switches and routers [3]. The traffic passing through selected ports of the switch or router is mirrored to another selected port. The port used for output of the duplicated traffic is usually called *mirror port* or *SPAN port* (Switched Port ANalyzer).

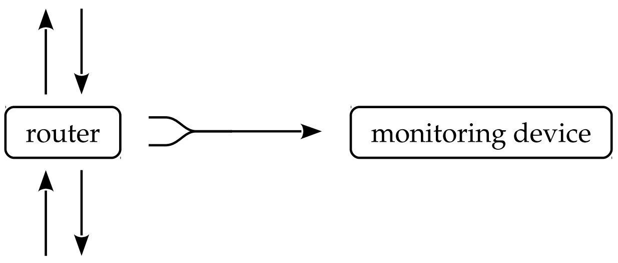

Figure 2.2: Principle of port mirroring. Both directions of the monitored link are transmitted in one direction over the mirror port.

There are two downsides of mirror ports. First, if the sum throughput of the traffic is larger than the mirror port can transmit, the mirror port becomes congested and drops packets. A full-duplex traffic is transmitted in one direction over the mirror port. That is up to twice the bandwidth of a single port for two ports serviced by the switch, and even more if more than two ports are serviced [4]. Second, most switches do not have enough computational power to handle both switching and mirroring. The switch's primary function is prioritized and the mirroring may not work properly during periods of peak traffic.

2.1.2 Test Access Port

Test Access Port (TAP) is a packet capture device positioned in inline mode since the observed line is split. A TAP device is connected between the split parts of the line and the traffic is duplicated. Single TAPs duplicate the traffic to a single output, consisting of two physical ports for both downlink and uplink of the full-duplex link. Regeneration TAPs duplicate the traffic into multiple outputs. Aggregation TAPs merge both channels into one output port. There are three types of TAPs — copper, fiber, and virtual.

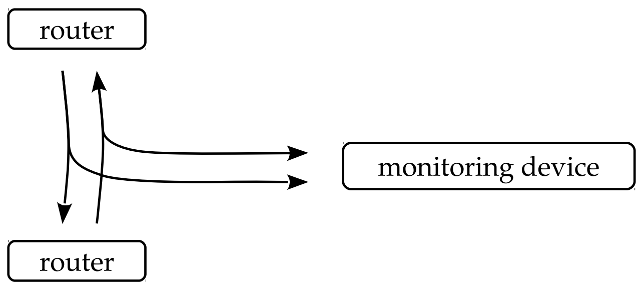

Figure 2.3: Traffic mirroring using Test Access Port. Both directions of the monitored link are transmitted separately.

Passive copper TAPs connect directly to the line. Since passive TAPs are not powered, a power outage cannot introduce a fault on the line. A disadvantage to passive copper TAPs is that only 10-Mbps and 100-Mbps connections are possible to tap this way. The passive connection distorts the signal in such a way that it's not possible to tap a gigabit Ethernet passively [5]. A patent by NetOptic presents a method that uses an active gigabit TAP equipped with capacitors to maintain the connection while the built-in bypass relays are switching [6].

Active copper TAPs function in a way resembling the approach in the preceding subsection. The signal going through the TAP is actively retransmitted and duplicated and no signal distortions are introduced, short of the negligible delay caused by the electronic circuitry. Downside of this approach is that power failure of the TAP causes a failover relay to switch, introducing a several hundred microseconds long delay [7].

Passive optical TAPs divert a percentage of the original signal to the mirrored output. The fact that no power failure of the TAP can occur is an advantage. A disadvantage lies in the fact that the signal in the line is weakened by the TAP [8].

Regeneration optical TAPs divert a very small fraction of the original signal to the mirrored output and amplify it to the full strength. A power failure just disables the mirrored output and introduces no faults on the line.

2.1.3 TAP-like Setup Using a Bypass NIC

Setup using a network interface card (NIC) integrates traffic mirroring with traffic analysis. The observed line is split. Both ends in the split are connected to two NIC interfaces. The NIC is installed in a computer. The interfaces are configured in software as a network bridge. Acting as a bridge allows the split line to still function properly. Having the traffic pass through the computer allows traffic observation. This setup is positioned in inline mode, similarly to a TAP.

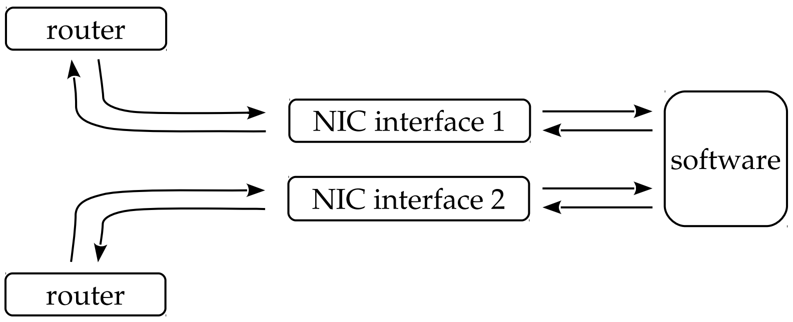

Figure 2.4: Inline mirroring using two interfaces of a NIC.

Mirroring using NIC is possible using consumer-grade NICs. This introduces a point of failure. Once the software or hardware fails, the line is not connected anymore.

Specialized, so-called *bypass NICs* exist. Bypass NICs have the ability to bypass the two network interfaces whenever a failure occurs; e.g., a software crash or power loss [9].

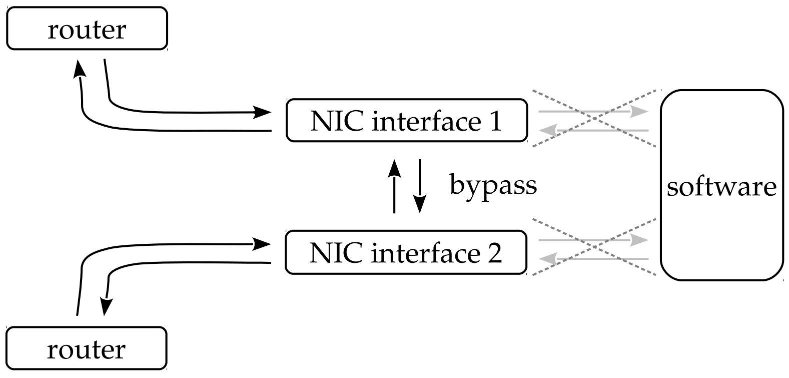

Figure 2.5: Bypass NIC in bypass mode bridges the connection in hardware so as not to break the connection.

A disadvantage is that the computer is locked in the particular location and cannot be moved without interrupting the connection.

2.2 Packet Capture

Packet capture has three meanings in no particular order. First, it is an interactive approach to network monitoring. Second, packet capture is a packet trace file. Third, it is the act of capturing packets from network link. The capture can be saved to a file or read directly by a network traffic analyzer in real time. Methods relying on packet capture as a source of packets for analysis are described in section 2.3 and section 2.4. The terms *packet capture*, *capture*, and *packet trace* are used interchangeably with the three aforementioned meanings; the particular meaning being clear from context [10].

2.2.1 Packet Capture as a Packet Duplication Method

Network traffic is captured from an observation point. It is not necessary that the capture is temporally and spatially dependent on subsequent analysis since the captured data can be saved to a file. This can be a temporary file as a part of the whole network monitoring process or it can be saved to a file for later use explicitly. The captured data is the same as it was on the line.

The process of making a packet capture can be both manual and automatic [11].

2.2.2 Packet Capture as a Network Monitoring Approach

The packet capture network monitoring approach consists of two basic steps; first, creating the packet capture file, and second, performing network traffic analysis on the captured file.

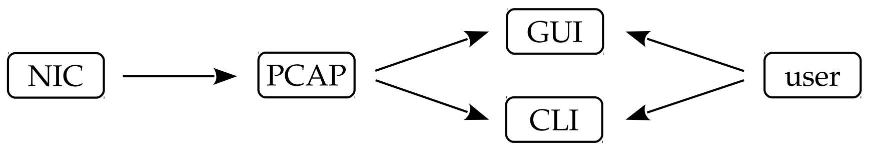

Figure 2.6: The packet capture approach to network monitoring is interactive rather than automated.

Packet capture as a network monitoring approach can be both manual and automated. The automated approach is used for malware behavior recording and analysis [11]. Additional manual analysis of selected packet captures from such an automated system may also be possible [12]. The fully-manual approach is presented in the following paragraphs.

The Layer 3 of the OSI model is usually used so that the traffic is seen as a series of IP packets. The captured traffic in this representation can then be viewed, searched in, or filtered [13]. It is also possible to filter the packets *before* capturing the packet trace [14].

Both graphical user interface (GUI) and command line interface (CLI) are used. In some setups, automation through scripting of the actions is possible. This is merely intended as a help for the human user and not designed to implement a complicated automated system. An intrusion detection system (IDS) may be considered to be a complicated automated system. In the context of packet capture analysis, even an IDS-like scripted functionality is still intended for individual interactive analysis [15].

The accessibility of full network data for free viewing and searching is unsurpassed among all the architectural approaches mentioned in this chapter. The interactivity and access to any part of the traffic data can be an immense advantage. The user can search for highly specific artifacts without having to program anything and without being constrained by more automated software. It is useful for dealing with new traffic patterns — such as new malware or unknown communication protocols [12]. The interactivity is, however, a disadvantage when the pattern is already known, and the work is repetitive and automatable. The approach doesn't scale and it becomes burdensome to search through large amounts of data.

PCAP [16] is a commonly used file format for storage of the captured traffic. Tshark [17] and tcpdump [18] are examples of software operated through CLI. Wireshark [19] is an example of software with GUI. Wireshark's architecture is shown on Figure 2.7.

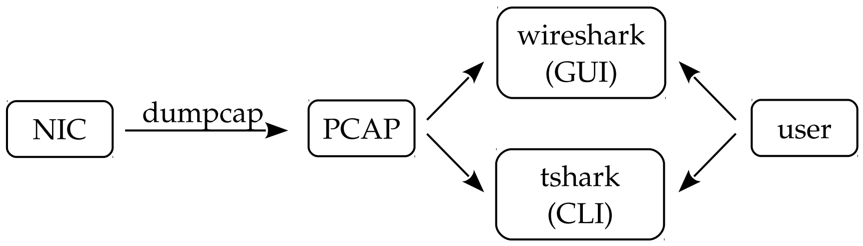

Figure 2.7: Wireshark is a GUI application, while tshark is a CLI application. Both use the same underlying dumpcap tool.

2.3 Deep Packet Inspection

Automation is a characteristic feature of deep packet inspection (DPI), especially in comparison to manual packet capture and analysis described in the previous section.

Deep packet inspection is a technique of seeing the payload of IP packets. It is, however, also used to denote those architectural approaches to network traffic monitoring that use DPI in an automated fashion; DPI is incorporated into inherently automated systems.

Traffic capture and further analysis can be either separate processes in both time and space or they can be integrated in one process pipeline, as shown on Figure 2.8. The packet capture approach can serve as a source of a PCAP file for further DPI-based analysis [20].

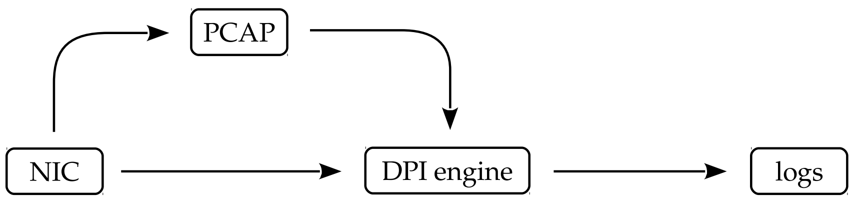

Figure 2.8: DPI-based approaches analyze network traffic directly and usually can also analyze a PCAP file in the same way.

There are two major types of DPI-based analysis — pattern matching and event-based analysis. Both are used in various IDS/IPS (Intrusion Detection Systems / Intrusion Prevention Systems).

2.3.1 Pattern Matching

Pattern matching is a DPI method that involves searching through full network data for known sequences of bytes or for regular expression matches [21]. The principle of operation is shown on Figure 2.9. The search can be limited to specific parts of packets or to specific packets.

The relative simplicity is an advantage of this approach, which is why it is a popular type of DPI. Describing the sought data by sequences of bytes or by regular expressions is often straightforward.

This strength, however, becomes a problem when we want to search for patterns that are not possible or feasible to describe using regular expressions. If the data has to be decoded before further pattern matching and the decoding functionality is not already built in the network security monitor, it is usually impossible to craft a regular expression that also does the decoding. Compression may serve as an example of such decoding necessary before the pattern matching.

Complicated decision logic is also infeasible to do using just regular expressions. Example task; alert on expired SSL certificates on HTTPS connections coming from a specified list of IP addresses and from nowhere else. Translating detection of SSL certificate data to regular expressions might be impossible. The check whether a specific certificate belongs to a list might result in a complicated regular expression. Even if the first problem is ameliorated by an SSL decoder, the second one still stands. One step further, if the list of cues is dynamically changing at runtime, conversion of the algorithm to a regular expression becomes impossible at all — an example for that might be a threshold-based detection, e.g., alerting on hosts receiving more than 10 DNS errors per hour.

Today's network traffic monitors employing the pattern matching DPI method usually decode the most used protocols [22].

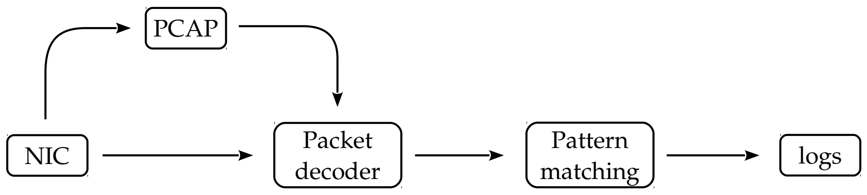

Figure 2.9: Packets are first decoded into representations that can be matched.

The pattern matching approach is slow, compared to the flow observation approach in the section 2.4. A concrete pattern matching implementation for 10 Gbps requires hardware acceleration using FPGA [23, p. 115–116]. In contrast, a concrete flow observation implementation with no hardware acceleration handles 40 Gbps [24]. Compared to the event-based approach in the next subsection, the pattern matching approach is also rather simplistic.

There are numerous algorithms for pattern matching. Pattern matching algorithms in the context of network monitoring are described by J. Kelly [25]. Besides algorithms, there are software packages for pattern matching, ready to be built into other software — Flex and MultiFast for instance. Use of Flex and MultiFast in the context of network traffic analysis is researched by T. Šíma [26].

Snort [27] and Suricata [28] are software implementations of pattern matching DPI. ngrep [29] is a command line utility for pattern matching in captured packet traces.

2.3.2 Event-based Analysis

The previous subsection describes cases where pattern matching is clearly an insufficient technique. Its inability to perform decoding or multiple steps of decision making is addressed in the architectural approach of event-based analysis.

In the approach of DPI with event-based analysis, packets are processed into events that are in turn processed by scripts [30]. Scripts may implement complex processing algorithms and add new DPI-related functionality.

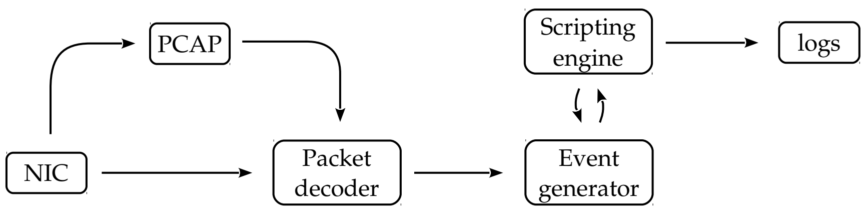

Figure 2.10: DPI with event-based analysis.

Such an architectural approach replaces the pattern matching part with algorithms implemented as a computer program. The algorithms can be both stateful and stateless. Stateless algorithms are just an immediate reaction or chain of reactions to specific events. Stateful algorithms can use program variables to remember state between event occurrences.

Bro Network Security Monitor [31] is a network monitor with such an architecture.

2.4 Flow Observation

An approach differing from the ones described in preceding sections is flow observation. The contents of the packets are not analyzed beyond information from packet headers. These information are aggregated into flows. RFC 7011 [32] provides the following definition of a flow: *"A Flow is defined as a set of packets or frames passing an Observation Point in the network during a certain time interval. All packets belonging to a particular Flow have a set of common properties."*

The following five-tuple is used as the common property distinguishing flows from each other: source IP address, destination IP address, source IP port, destination IP port, Layer 4 protocol.

In the architectural approach of flow observation, the network traffic monitor stores information about the observed flows — like the identifying

5-tuple, number of transmitted bytes and packets and L4 protocol flags — but it does not analyze nor store the payload.

Because the data itself is not handled, flow observation has several advantages. Since the payload is not analyzed, flow observation is faster than the other approaches on the same hardware. Also, not storing the payload results in considerably less data stored than in the case of packet capture mentioned in section 2.2. Not processing the payload also makes it less of a privacy concern, compared to packet capture or DPI. Flow data may be used to comply with data retention laws [33].

Figure 2.11 shows the flow observation architecture. Upon observation, packets are sent to a metering process. The metering process identifies flows and counts their statistics. From this point on, the original packets are not processed. The metering process sends the information about flows to an export process after a certain period of time. The metering and export processes usually reside together on a network probe. The export process sends the finalized flow information to a collector for storage and subsequent processing.

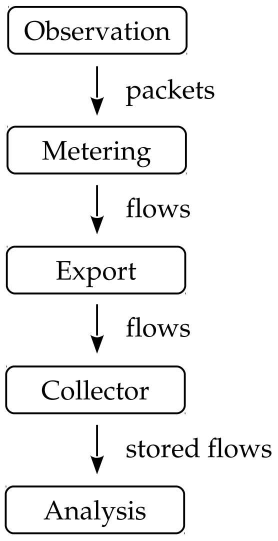

Figure 2.11: Flow observation architecture.

There are two notable formats for the transmission of flow information — NetFlow [34] developed by CISCO and IPFIX [35] developed by IETF. Some of the existing software flow exporters are nProbe [36] softflowd [37], YAF [38], and FlowMon [39]; selected flow collectors are nProbe [36], nfdump [40], flowd [41], IPFIXcol [42], and SiLK [43].

Chapter 3

Network Security Monitoring Implementations

The previous chapter serves as an overview of network monitoring approaches. This chapter presents concrete software implementations of the aforementioned approaches.

The section 3.1 presents representative software implementations of the packet capture approach. The section 3.2 presents implementations of the deep packet inspection approach. The section 3.3 lists flow-based software implementations.

The section 3.4 chooses Bro for evaluation. Bro is a network security monitor utilizing deep packet inspection. The choice was made with respect to the circumstances mentioned in the chapter 1. The evaluation of Bro is described in the chapter 4.

3.1 Packet Capture Representatives

3.1.1 Tcpdump

Tcpdump is a command line tool for packet capture analysis. Tcpdump can analyze both live traffic using the libpcap library and captured packet traces in PCAP format. Packets may be filtered both before and after the capture. Filtering before the capture can be done using BPF (Berkeley Packet Filter). Filtering after capture can be achieved using tcpdump's filters, described later in this section.

Data are printed out in text format. The output displays individual packets with information that include source and destination addresses, L4 protocol used, and L4 protocol flags. Figure 3.1 show output listing two packets.

Packets to be displayed can be filtered using expressions. Filters can be imposed on source and destination addresses, ports, L3 and L4 protocols, and L4 protocol flags. Addresses can be expressed in format of individual addresses or in CIDR notation. Multiple rules in the expression can be composed using boolean operators. The example on Figure 3.1 uses three filters. `src host 10.0.2.15` selects only packets originating in the IP address 10.0.2.15. `dst port 22` selects only packets destined for the port 22. Finally, `tcp[13] = 2` selects packets in which the decimal value of the 14th byte is 2. The filters are composed using the `and` word, which means only packets that meet all the criteria pass through the filter.

```
$ tcpdump -r pcap -n \
 "src host 10.0.2.15 and dst port 22 and tcp[13] = 2"
reading from file pcapfile, link-type EN10MB (Ethernet)
20:20:40.512613 IP 10.0.2.15.54346 > 192.168.1.42.22: Flags [S],
 seq 3123841387, win 29200, options [mss 1460,sackOK,TS val 509640
 ecr 0,nop,wscale 7], length 0
20:20:41.356843 IP 10.0.2.15.45749 > 192.168.1.41.22: Flags [S],
 seq 1764864395, win 29200, options [mss 1460,sackOK,TS val 509851
 ecr 0,nop,wscale 7], length 0
```

Figure 3.1: Example of a tcpdump filter and tcpdump's output.

Tcpdump needs root privileges to open the network interface. Operation without full root is possible using SUID or Linux capabilities. Granting the `tcpdump` executable `cap_net_raw` and `cap_net_admin` capabilities allows tcpdump to be run as a regular user.

3.1.2 Wireshark

Wireshark is a graphical tool for packet capture analysis. While Wireshark and tcpdump are implementations of the same architectural approach, their underlying ideas differ. Tcpdump is as close to the raw data as possible, while Wireshark strives to provide higher-level representation of the same data.

Wireshark can analyze both live traffic using the libpcap library and captured packet traces in PCAP format. Captured packets can be filtered both during and after the capture. Filtering after capture can be achieved using filter expressions. Filtering during capture can be done using BPF.

Data are displayed as text arranged in a scrollable colored list and expandable boxes. Wireshark's main window has two frames. The upper frame displays list of captured packets with their basic attributes displayed. A line may be colored, based on the protocol the individual packet belongs to. When the user selects packet in the upper frame, this particular packet is displayed in the lower frame. The lower frame's representation includes several boxes that can be expanded and collapsed. The boxes contain various attributes of the packet as well as representations of the packet's data in ISO OSI layers' data the packet is part of. Also, related data from other packets may be displayed there — HTTP TCP stream for instance.

Packets displayed in the upper frame can be filtered using expressions entered in the text box on the top of the window. The expression vocabulary is richer that the one of tcpdump.

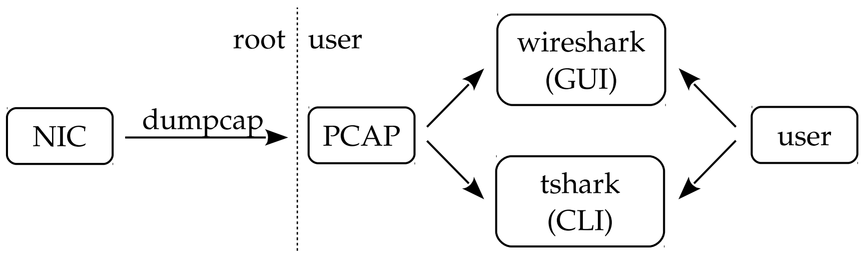

Figure 3.2: Wireshark architecture showing privilege separation.

Wireshark uses a separate program to capture the traffic — dumpcap. The reason is to allow separation of privileges [44]. Wireshark can be run as a regular user and only dumpcap has to be given special permissions. Dumpcap needs root privileges to open the network interface. Operation without the user having root access is possible using either SUID or Linux capabilities. Granting the `dumpcap` executable `cap_net_raw` and `cap_net_admin` capabilities allows dumpcap to be run as a regular user without SUID.

3.2 Deep Packet Inspection Representatives

3.2.1 Snort

Snort is an intrusion detection system performing deep packet inspection using pattern matching.

The pattern matching is implemented in the form of rules [45]. Rules are structured text files describing network traffic data of interest. Typically, rules are used to generate alerts when a security-related incident occurs, such as malware activity, attack, or breach of security policies. A rule contains information specifying when the rule should be triggered. An important part of these information is one or more patterns that are searched in the network traffic. The pattern can be a sequence of characters or bytes or a regular expression.

```
rule header       alert tcp any any -> 192.168.1.0/24 111 \
rule options       (content:"|00 01 86 a5|"; msg:"mountd access";)
```

Figure 3.3: Snort rule structure — the rule header and rule options [46].

Snort rules have a specific structure. The beginning of the rule before the parentheses describes which network flows the rule refers to. This is called the rule header. The rule header specifies the action the rule should perform (`alert` for instance), L3 protocol and source/destination IP addresses and ports on which to match. Variables may be used in place of IP addresses. The rest of the rule inside the parentheses is called the rule options. Figure 3.3 shows the rule structure. Rule options specify content on which the rule matches and other properties of the rule — its name, classification type, etc. The most important part of the rule is the `content` keyword that specifies a pattern to be found in the packet's payload. The `content` keyword's function can be further changed using modifiers. For instance, modifiers `offset`, `distance`, `depth`, and `within` control in which areas of the packet the rules are matched [47]. Figure 3.4 shows these keywords used in a rule. Text or binary strings are used in the `content` patterns. The `pcre` keyword allows the use of regular expressions. Figure 3.5 features a rule that uses a regular expression.

```
alert tcp $EXTERNAL_NET 5050 -> $HOME_NET any \
 (msg:"POLICY-SOCIAL Yahoo IM successful chat join"; \
  flow:to_client,established; \
  content:"YMSG"; depth:4; nocase; \                      first 4 bytes
  content:"|00 98|"; depth:2; offset:10; \           11th and 12th byte
  \     The patterns must be at the specified place of the payload for a successful match.
  \
  metadata:ruleset community; classtype:policy-violation; \
  sid:2458; rev:9;)
```

Figure 3.4: Example of a rule using the `depth` and `offset` keywords [48].

Snort has three special keywords, `byte_test`, `byte_jump`, and `byte_extract`, that allow to adjust the pattern matching based on data in an individual packet [49]. The first two keywords behave as patterns that match when their conditions are true. `byte_test` performs arithmetic ($<$, $\leq$, $=$, $>$, $\geq$) and bitwise (AND, OR) comparisons on sequences of bytes. An example is shown on Figure 3.6. `byte_jump` puts a space before the next pattern with size inferred from payload's byte value. If the jump is possible, `byte_jump` also behaves as a match. This behavior can be used to match packets with a specific length based on specific data in the payload. `byte_extract` converts specified bytes into a numerical variable that can be used later in the rule. These keywords do not allow more complicated decoding or processing, as mentioned in the section 2.3.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS \
 (msg:"MALWARE-CNC Win.Trojan.CryptoLocker variant connection"; \
  flow:to_server,established; \
  content:"/crypt_1_sell"; fast_pattern:only; http_uri; \
  \                          simple pattern allowing fast packet preselection
  \
  pcre:"/\/crypt_1_sell\d\d-\d\d.exe$/Ui"; \
  \            perl-compatible regular expression confirming or rejecting the match
  \
  metadata:impact_flag red, policy balanced-ips drop, \
   policy security-ips drop, ruleset community, service http; \
  reference:url,www.virustotal.com/en/file/d4b16269c9849c33a7bb\
   2fdc782173a00e99db12a585689618dde3f4c6fcb101/analysis; \
  classtype:trojan-activity; sid:28044; rev:3;)
```

Figure 3.5: Example of a rule using the `pcre` keyword [48].

```
alert tcp $EXTERNAL_NET $FILE_DATA_PORTS -> $HOME_NET any
 (msg:"BROWSER-IE Microsoft Internet Explorer bitmap\
 BitmapOffset integer overflow attempt";\
 flow:to_client,established; flowbits:isset,file.bmp;\
 file_data; content:"BM";\
 byte_test:4,>,2147480000,8,relative,little;\
 metadata:ruleset community, service ftp-data, service http,\
 service imap, service pop3; reference:bugtraq,9663;\
 reference:cve,2004-0566; reference:url,technet.microsoft.com\
 /en-us/security/bulletin/ms04-025; classtype:attempted-user;\
 sid:2671; rev:18;)
```

Figure 3.6: Example of a rule using the `byte_test` keyword in a check for a buffer overflow [48].

There are multiple attack classifications defined in Snort and rules can be assigned to them using the `classtype` keyword. Categorizing attacks helps to organize the event data produced by Snort. Figure 3.7 shows several rules with various classification.

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any        \
 (msg:"BROWSER-OTHER Mozilla Netscape XMLHttpRequest \
 local file read attempt"; flow:to_client,established;      \
 file_data; content:"new XMLHttpRequest|28|";               \
 content:"file|3A|//"; nocase; metadata:ruleset community,  \
 service http; reference:bugtraq,4628;                      \
 reference:cve,2002-0354;                                   \
 classtype:web-application-attack; sid:1735; rev:13;)
alert tcp $EXTERNAL_NET 5050 -> $HOME_NET any               \
 (msg:"POLICY-SOCIAL Yahoo IM successful chat join";        \
 flow:to_client,established; content:"YMSG"; depth:4;       \
 nocase; content:"|00 98|"; depth:2; offset:10;             \
 metadata:ruleset community;                                \
 classtype:policy-violation; sid:2458; rev:9;)
```

Figure 3.7: Example of various classification types in Snort [48].

The Snort's architecture allows the implementation of so-called preprocessors [50]. Preprocessors read the packet before rule evaluation, serially in the order specified by Snort's configuration. This allows implementation of additional rule keywords. Moreover, preprocessors allow implementation of functionality more complicated than just pattern matching, such as data decoding and anomaly detection. For instance, the Normalizer preprocessor converts equivalent values to a unified format with the goal of making IDS evasion harder. Snort's architecture including the preprocessors is depicted on Figure 3.8.

There are several preprocessors for anomaly detection available. Frag3 and stream5 preprocessors are integrated in the official Snort distribution and detect protocol anomalies. SPADE [51], PHAD [52], and snortad [53] are 3rd party preprocessors detecting traffic anomalies.

Frag3 detects anomalies in IP packet fragmentation. Different operating systems implement IP packet defragmentation in different ways. An attacker can try to evade the IDS by exploiting the inconsistency between IP packet defragmentation implementations of the IDS and the attacked system. Frag3 is aware of the varying implementations and applies the appropriate defragmentation algorithm for the particular destination. This is called target-based analysis.

Stream5 is quite similar to frag3 in its use of target-based analysis. Different operating systems have differing implementations of TCP, e.g., whether to allow data in SYN TCP packets or how overlapping TCP segments are handled. Stream5 also provides TCP stream reassembly for use by other

Snort's components.

SPADE (Statistical Packet Anomaly Detection Engine) used to be a Snort preprocessor for anomaly detection. SPADE was funded by DARPA (Defense Advanced Research Projects Agency) and disappeared from the Internet after funding cuts [51]. SPADE analyzes network activity using probabilities of destination ports based on destination addresses. Although SPADE's source code was released under GNU GPL, we were unable to find a copy.

PHAD (Packet Header Anomaly Detection) is a standalone application that detects anomalous values outside of expected ranges for fields in L2, L3, and L4 packets [54]. PHAD uses nonstationary model; the longer an unusual activity has not been seen, the higher is the severity of a new anomaly [55]. There is an implementation of PHAD as a Snort preprocessor [56].

SnortAD is a project consisting of two programs — Snort preprocessor, called simply *preprocessor*, and a standalone application called *profilegenerator* analyzing *preprocessor*'s logs. *Preprocessor* is able to detect anomalies solely based on minimum and maximum bounds for absolute numbers of packets or bytes in the whole traffic, and for several similar types of values. *Profilegenerator* implements the Holt-Winters method [57].

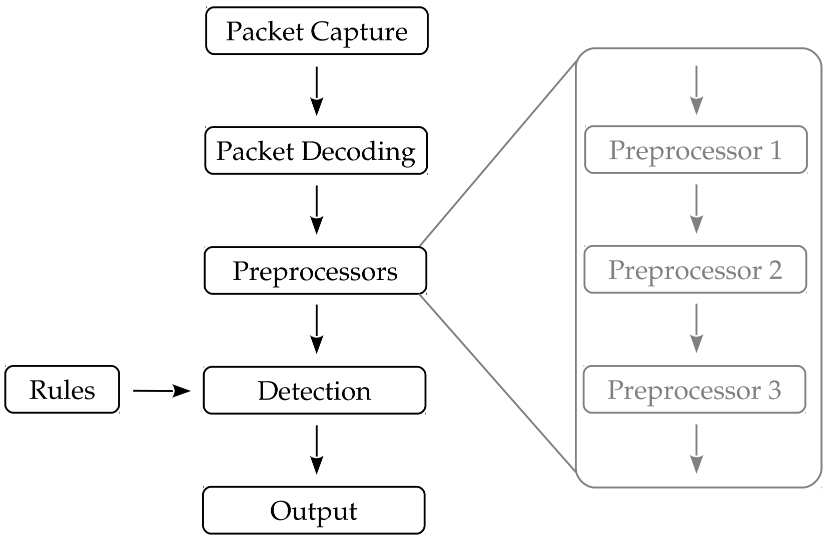

Figure 3.8: Snort architecture [58]. The preprocessors allow processing patterns not supported by the rules.

Snort preprocessors are usually implemented in C. They allow implementation of similar concepts to those that can be implemented in the event-based architecture. It is, however, more complicated to share common primitives in the additional preprocessors than it is in the event-based Bro Network Security Monitor, which is described in the section 3.2.

Snort needs root privileges to open the network interface. It is possible to configure Snort to drop its privileges to a non-root user once it opens the network interface.

Snort is a single-threaded application. Multithreaded Snort setups work in the following way: The monitored traffic is divided by flows into multiple parts; each part of the traffic is fed to a single Snort instance [59].

3.2.2 Suricata

Suricata is an intrusion detection system performing deep packet inspection using pattern matching.

rule header	`alert tcp any any -> 192.168.1.0/24 111 \`
rule options	`(content:"|00 01 86 a5|"; msg:"mountd access";)`

Figure 3.9: Suricata rule structure — the rule header and rule options. Same as the Snort rule structure.

Suricata uses similar rules to Snort and is compatible with Snort rules. The rule structure is the same for both Snort and Suricata (Figure 3.3). The difference between the two is in the keywords and protocols that can be specified. Suricata allows specification of several L7 protocols on top of the L3 protocols supported by Snort — `http`, `ftp`, `tls`, `smb`, and `dns` [60]. Some keywords behave differently than in Snort — for instance, the `fast_pattern:only` keyword doesn't make a difference in processing, unlike in Snort. Some keywords are supported only by Suricata, such as the `iprep` keyword for matching IP reputation data and the `dns_query` keyword for analyzing only the DNS response body.

Suricata's architecture is similar to Snort's one with a difference. What corresponds to the preprocessor part in the Snort's architecture is divided in two in Suricata — decoding and detection. We found out about this by studying the source code [61]. Decoding modules add information to the internal representation of packets in Suricata. Detection modules rely on this internal representation and provide keywords for use in rules. Overview of the Suricata's architecture is shown on Figure 3.10.

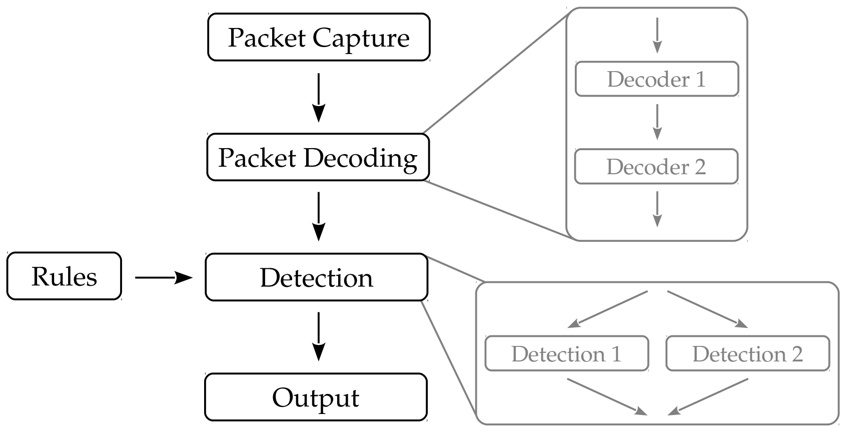

Figure 3.10: Suricata's architecture. The decoding and detection modules allow processing patterns not originally supported by the rules.

Each packet is first processed in decoding functions and then in detection modules. Decoding functions read the packet and save the decoded data into an internal representation of the packet. The decoding functions are called one at a time on the packet. Extending the decoding functionality is possible by implementing a new decoding function and placing it into the decoding pipeline. The decoding pipeline starts with the source of captured packets, then L2 is decoded, and then protocols on higher layers are decoded.

Upon decoding, the packets pass detection. The detection is governed by rules and depends on the decoding step. The rules are matched with the internal packet representation. The matching process is broken into several detection modules in all of which the matching takes place. Unlike decoding, detection is parallelized and one packet can be processed in multiple detection modules at the same time. Extending the detection functionality is possible by implementing a new detection module and registering it in the table of detection methods.

Suricata is written in C and the modules for Suricata have to be written in C. There are no plans supporting C++. C requires greater programming expertise than the Bro language. Therefore, this property makes Suricata not the best prototyping tool available.

Suricata needs root privileges to open the network interface. It is possible to configure Suricata to drop its privileges to a non-root user once it opens the network interface. This option is similar to Snort.

Suricata is multithreaded out of the box. Even though it's not as fast as Snort on a single-CPU computer, Suricata is designed to scale on computers with tens of CPUs [62]. The multithreading approach is different from Snort. Multithreaded Snort setups divide the monitored traffic by flows into multiple parts, each processed by an individual Snort instance. Suricata, on the other hand, doesn't require the traffic balancing since it manages multithreading itself. This approach makes it more user-friendly.

3.2.3 Bro

Bro [31] is a network security monitor performing deep packet inspection using event-based analysis.

In contrast to Snort and Suricata, Bro is primarily not rule-driven. Instead, it implements a Turing-complete scripting environment [30]. Rule-based detection as well as arbitrary detection algorithms can be implemented in this environment. Bro detection rules are described by scripts.

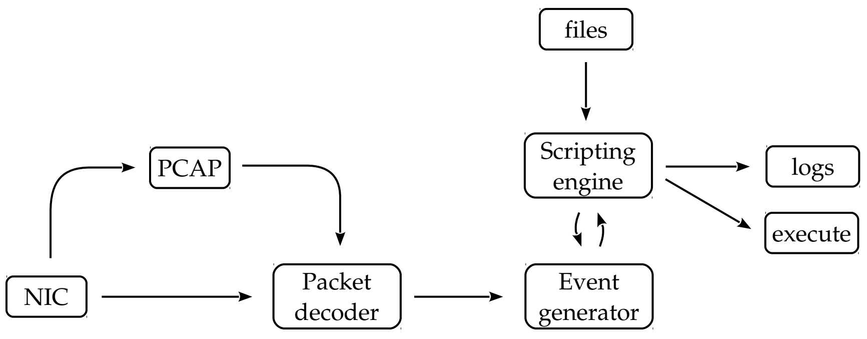

Figure 3.11: Bro architecture.

The Bro's scripting environment uses the Bro programming language. It's an interpreted, typed language. What makes it special are domain-specific types. For example, the `addr` type holds an IP address [63]. Variables of structured types are reference type variables. This makes processing of large sets or tables efficient, since only the references are copied, not the data itself. There are two types of collections — sets and tables. Loops are available in the form of iteration through collections. The Bro programming language lacks other forms of loop control, presumably serving as a deterrent against overly complex algorithms. This is a reasonable requirement for network traffic monitoring when the processing is done in real time. And that exactly is the most significant goal of Bro — to allow real-time network traffic analysis and save already processed results to log files.

The default installation of Bro contains many scripts implementing various sorts of traffic analysis. Some of the items the default Bro setup monitors are:

- Bidirectional flows,
- DHCP leases,
- DNS queries and responses,
- MD5 and SHA1 hashes of files transmitted over unencrypted protocols,
- HTTP requests and user agents,
- port scans,
- email headers from SMTP traffic,
- successful and unsuccessful SSH connections,
- SSL certificates,
- SYSLOG messages,
- traffic tunnels.

The full list of monitored areas is in the Bro documentation.

Since the preinstalled scripts usually expose an API in the form of events, they can be used by user scripts, extending the default functionality.

The core of Bro, implemented in C, processes network traffic, performs DPI and generates events about what's happening in the traffic. Events generated by the core are listed in the bif files [64]. Many events are generated, spanning L2 through L7. Examples are a new ARP packet, closed TCP connection, HTTP request, etc. In other words, this type of DPI performs semantic matching of network events instead of simple pattern matching, as opposed to Snort and Suricata. Majority of the events provide context, typically in the form of information about the relevant connection. The events are then processed by the Bro scripts.

Bro scripts use so-called event handlers to listen to the events. The usual reactions to events vary. On the one hand, the simplest possible processing saves the event information to a log file. On the other hand, some scripts implement fairly complex processing and generate additional types of events.

This further extends DPI abilities of Bro. Scripts can handle events generated both by the core and by other scripts.

```
module helloworld;
event bro_init() {
        print "Hello world!";
}
```

Figure 3.12: A simple Hello world! script.

The scripting engine hosts the scripts and dispatches events generated both by the scripts and the core to the scripts listening to these events. It also allows operations like file access and execution of applications native to the operating system. This functionality can be used by advanced scripts. File access may be used to fetch information from external sources, e.g., a blacklist. Execution facility may be used for many purposes. One example is reporting issues to a ticket managing software via email. The `sendmail` executable can be used by such a script. Another example is automatic triggering of a remotely triggered black hole by executing a program that does the blackholing.

The Bro's architecture is shown on Figure 3.11. Figure 3.12 shows a very short module that just writes "Hello world!" to the standard output when Bro starts.

Figure 3.13 shows commented code that manipulates data structures and dispatches two events. It is an excerpt from the `database.bro` script, which is part of Appendix A.

Bro scripts are organized in so-called *modules*. A module can be implemented wholly in one file or can be broken into several files. Two identifiers with the same name in two different modules do not collide with each other. Cross-module references can be made using the name `name_of_module::name_of_identifier`.

A module can define types, variables, functions, and event handlers. These entities can be either local to the module or globally accessible from other modules.

It is possible to define custom types using `enum`, `set`, `table`, `vector`, and `record`. `Enum` in Bro is similar to `enum` in other languages. `Set` is similar to `HashSet<T>` in C# in its functionality [65], albeit the syntax is different. `Table` is similar to `Dictionary<TKey,TValue>` in C# [66] with the difference that C# allows only one key while Bro allows multiple keys. To present this feature on an example, Figure 3.14 shows declaration of a `table` indexed by two `addr` type keys. Such a table can be comfortably used to store information relevant to unidirectional communication between pairs of hosts. `Vector` is a table indexed by `count`. `Count` is the name for int in Bro. `Record` is similar to C# class [67] that contains only fields [68]. Both Bro `record` and C# `class` are reference types, meaning assignment of its instance copies only the reference (pointer), not the whole instance. This can be compared to C# `struct` which is a value type, meaning assignment of its instance copies the whole instance.

```
## Private function that updates the IP-MAC database.
function internal_mac_table_update(ip: addr, mac: string){
        ## A new instance of a set.
        local new_exp_set_addr: expiring_set_addr;

        ## Add the IP->MAC mapping to the database if it is not in
        ##  there yet.
        if (ip !in database::ip_to_mac)
                database::ip_to_mac[ip] = mac;

        ## Add the new instance of a set to the MAC->IP database
        ##  if there is no set for the MAC yet.
        if (mac !in database::mac_to_ip)
                database::mac_to_ip[mac] = new_exp_set_addr;

        ## Add the MAC->IP mapping set. (Does nothing if it already
        ##  is in the database.)
        add database::mac_to_ip[mac][ip];

        ## Call the events so that any listeners have the chance to
        ##  know something happened. The arguments contain information
        ##  for the currently processed IP-MAC mapping.
        event database::mac_table_update_event_single(ip, mac);
        event database::mac_table_update_event_fullip(ip, mac,
                                                     mac_to_ip[mac]);
}
```

Figure 3.13: Demonstration of data manipulation and event dispatching.

Variables can use both built-in types and custom types — described in the previous paragraph — defined both in the same module and in other modules. Regarding visibility, there are three types of variables — those local to functions and event handlers, global variables visible in its own module only, and global variables visible by all modules.

The visibility of global identifiers is controlled by the `export` section and the `global` keyword.

```
## Declaration of a table indexed by two values.
global profiles: table[addr, addr] of individual_profile
                                &write_expire = alias_expiration ;
```

Figure 3.14: Table indexed by 2 indices. A part of the `database.bro` script.

Functions and event handlers in Bro are similar but distinct. Function, using the `function` keyword in code, is a part of code that can be only explicitly called by other parts of code. In other words, you would have to specifically know about a function to be able to call it. Event handler, using the `event` keyword in code, is too a part of code that can be called. The distinction lies in the fact that the caller doesn't have to be aware of an event handler's existence in order to call it. The caller of an event handler just executes an event. All event handlers handling this particular event are then called. It's the scripting engine, not the event's caller, that has to be aware of each event handler.

Bro can be run both as a single-threaded application and as a multi-threaded distributed application. The single-threaded mode is called *standalone* while the multithreaded one is called *cluster*. If Bro is used as a platform for development of proof-of-concept methods, the standalone mode is usually more appropriate than the cluster mode. Development for the cluster mode is more difficult than for the standalone mode because additional functionality has to be used by scripts [69].

```
#fields note
#types  count
PacketFilter::Dropped_Packets
tor_detection::alert
SSL::Invalid_Server_Cert
domain_flux::alert
correlation::apt_detection_one_step
correlation::apt_detection_two_steps
correlation_same_host::apt_detection_one_step_same_host
```

Figure 3.15: An excerpt from the `notice.log` file showing various notice types in use, both built-in and custom.

Snort uses *classtypes* to differentiate between various types of alerts, as shown on Figure 3.7. Bro allows scripts to generate *notices* of arbitrary types informing or alerting about current affairs. Figure 3.15 shows custom and built-in notice types. For instance, the built-in `SSL` module detected an invalid certificate and the `tor_detection` module generated an alert.

Bro needs root privileges to open the network interface. Operation without full root is possible thanks to Linux capabilities. Granting the `bro` executable `cap_net_raw` and `cap_net_admin` capabilities allows Bro to be run as a regular user.

3.3 Flow-based Observation Representatives

The section 2.4 mentions that flow-based observation architecture contains two main components — a flow exporter and a flow collector. This section covers representative implementations of both flow exporters and flow collectors.

3.3.1 Flow Exporters

nProbe

nProbe [70] is a commercial open-source flow exporter. Data can be exported in NetFlow v5, NetFlow v9, and IPFIX formats. nProbe has an *application visibility (nDPI)* ability, which is used for detection of application-specific protocols. This information is saved in a custom column in NetFlow v9 or IPFIX format. It is difficult to obtain nProbe source code for free.

YAF

YAF [38] is an open-source flow exporter. Data is exported in the IPFIX format. A passive OS fingerprinting functionality based on the p0f software can be compiled into YAF. YAF supports modules that implement DPI. However, YAF doesn't provide DPI in default setup.

QoF

QoF [71] is a fork of YAF. It removes all payload inspection abilities and instead focuses on passive performance measurements.

ipt-netflow

ipt-netflow [72] is a plugin for iptables for flow export. Data can be exported in NetFlow v5, NetFlow v9, and IPFIX formats. There's no special functionality besides standard network flows. There's also no apparent focus towards high-throughput networks. ipt-netflow is open-source.

pmacct

pmacct [73] is an open-source flow exporter and flow collector. Data can be exported in NetFlow v5, NetFlow v9, sFlow v5, and IPFIX formats. Supports high-throughput networks thanks to PF_RING. No DPI-related functionality is available in pmacct.

softflowd

softflowd [37] is an open-source flow exporter performing export to NetFlow v1, v5, and v9 formats. There is no apparent effort to provide anything on top of regular NetFlow data export.

3.3.2 Flow Collectors

nProbe

nProbe is not only a flow exporter, it is also a flow collector. Available storage backends are MySQL, SQLite, text files, and binary files. The nProbe flow collector was created because its author deemed other collectors available at the time to be too cumbersome to use.

IPFIXcol

IPFIXcol [74] is an IPFIX collector designed for high-throughput networks. IPFIXcol claims to be flexible. Storage backend can be customized using output plugins. IPFIXcol also allows implementation of so-called IPFIX mediators, used for processing of the collected data before it hits the collector.

flowd

flowd [41] is a NetFlow v1, v5, v7, and v9 collector. It is created under the UNIX philosophy to do just one thing. The collected data is saved in a binary format. flowd is provided with Perl and Python interfaces for reading the binary data. flowd strives for security using privilege separation of components. flowd is open-source and freely available.

nfdump

nfdump [40] consists of several tools. The `nfcapd` tool listens to NetFlow v5, v7, v9 streams and saves them to nfcap files. The `nfdump` tool can be used for analysis of nfcap files. nfdump uses similar filter syntax to tcpdump. nfdump is open-source and freely available.

pmacct

pmacct [75] as a collector has several storage backends available. It can use MySQL, PostgreSQL, SQLite, MongoDB, BerkeleyDB, and flat files. Among other formats, it can collect NetFlow v1–v9 and IPFIX. pmacct is open-source and freely available.

SiLK

SiLK [43] is a collector for NetFlow v5, v9, and IPFIX data. It is designed for high-throughput networks. SiLK consists of multiple tools and plugins for filtering, analysis, and processing of flow data. SiLK is open-source and freely available.

3.4 Choice of Software for Evaluation

Among other things, this thesis deals with the selection of a network traffic monitor suitable for DPI. The following criteria are evaluated for each mentioned traffic monitor:

- Prototyping. Is the network traffic monitor suitable for creation of method prototypes?
- User-friendliness. Does the network traffic monitor allow development in an easy to use way?
- Extensibility. Is it possible to extend the existing functionality of the network traffic monitor in a reasonable way?
- High-throughput network support. Does the network traffic monitor support networks with high throughput?

The descriptions of individual network traffic monitors in this chapter indicate answers to these criteria. Table 3.1 shows the summary.

Monitor	Prototyping	User-friendly	Extensible
Tcpdump	No	No	No
Wireshark	No	No	No
Snort	No	No	Somewhat
Suricata	No	No	Somewhat
Bro	Yes	Yes	Yes

Table 3.1: Bro is a suitable tool for creation of prototypes.

We chose Bro because it is a perspective technology for prototyping and implements DPI in a way different than pattern matching.

Chapter 4

Network Topology Mapping Method in Bro

This chapter demonstrates DPI properties on a detection method implemented in Bro. The chapter first introduces DPI capabilities of Bro and sets them in the context of the previous chapter. The detection method is then introduced and it's explained how it demonstrates the importance of DPI. A comparison table of DPI and flow-based versions of the method is shown.

4.1 Bro's DPI-related Capabilities

The section 3.2 presents general characteristics of the Bro Network Security Monitor. This section shows concrete DPI-related features on code snippets, similarly to those shown on Figure 3.3 through Figure 3.7.

4.1.1 Signature Matching

Despite Bro not being in the group of pattern-matching DPI-capable network traffic monitors — it is an event-based network traffic monitor — it supports pattern matching [76]. Thus, its functionality is generally a superset of pattern-matching network traffic monitors.

Figure 4.1 shows a so-called *signature* instructing Bro to perform regular expression matching in packet payload. This particular signature is used by Bro to detect FTP events.

```
signature dpd_ftp_client {
  ip-proto == tcp
  payload /(|.*[\n\r]) *[uU][sS][eE][rR] /
  tcp-state originator
}
```

Figure 4.1: An example of signature from the `ftp/dpd.sig` file.

Since Bro is event-driven, each and every match generates an event that is then further processed by event handlers.

DPI Events

Only a small fraction of Bro's functionality is implemented using the signatures. Most of DPI-based detection in Bro is built around events. The ubiquity of events in Bro results in a predictable structure of any subsequent processing — handle an event, read information from the event, process the information, and optionally generate another event. An example of an event handler is shown on Figure 4.2.

```
# Listens to updates and logs the result
 event database::ttl_table_update_event_single (
     ip: addr, ttl: count, tcpwin: count, synsize: count) {
         check_ip(ip);
 }
```

Figure 4.2: An example of an event handler. A part of the `natdet.bro` script, which is part of Appendix A.

The code shows a so-called event handler. The body of the handler is executed every time this event is generated. The arguments contain information relevant to the particular instance of the event. The code can read the arguments to get details about what just happened.

Since the default installation of Bro offers a wide range of flow-based and DPI-based events, scripts can take advantage of them without having to use signatures or having to look directly in the raw packet payload.

Data Processing

Immediate handling of network data is only a part of the big picture. Bro has a set of specialized data types that can be used and arranged in arbitrary patterns. They are a part of the Bro programming language.

Figure 4.3 shows a complicated data structure that is used in the method further described in the section 4.2. Each type on the picture is used in the declaration of the successive type, resulting in a tree-like structure for storing data. Then a table with two indices of a specialized built-in type is declared. The table stores the previously defined tree-like structures.

```
## Declaration of complicated types.
## Each type is used by the type declaration that follows.

type profile_set: record {
        bifl_out: count;
        bifl_out_ports_src: set[count];
        bifl_out_ports_dst: set[count];
        sifl_out: count;
        sifl_out_ports_src: set[count];
        sifl_out_ports_dst: set[count];
        ports_src: set[count];
        ports_dst: set[count];
        out_byt: count
        out_pkt: count;
};

type proto_stats_struct: record {
        tcp: profile_set;
        udp: profile_set;
};


type individual_profile: record {
        data: profile_set;
        proto: proto_stats_struct;
        timestamp: time;
};

## addr is a type holding IPv4 or IPv6 address
global profiles: table[addr, addr] of individual_profile;
```

Figure 4.3: User-defined types can be further used in other user-defined types. A part of the `database.bro` script.

Records are types resembling C# classes that contain only fields. Each field in the record can be of any previously declared type. Sets are collection-like types of any specified type. Tables are array types of any specified type, indexed by any number of any specified types. All types — built-in and user-defined — are treated as equal. As long as types are declared before first use, and types can be used inside other types. Sets and tables can be assigned an *expire* attribute that causes individual stored items to be automatically deleted after a specified time.

The whole type system can be put in use by creating a data structure closely matching the needs of a particular algorithm. The network topology mapping method, introduced in section 4.2, uses a complicated structure of types that effectively mimics a *hierarchical database* [77]. Other pieces of the structure mimic a *key-value store*. All with automatic data expiration and using the specialized native types, such as `addr` for storing IPv4/IPv6 addresses.

Given the fact the scripting engine is connected with the specialized type system and the events providing traffic data, we consider data processing in Bro to be an integral part of its DPI capabilities.

4.2 Method Description

We chose an actual network traffic processing method to show the properties of DPI. The method used for the demonstration is called a network topology mapping method. It is a set of algorithms passively analyzing network data and identifying devices on the network. In the section 4.4 we compare capabilities of two versions of the network topology mapping method — the flow-based one and the DPI-based one. The flow-based version is inferior to the DPI-based version and cannot identify some important features in the network topology.

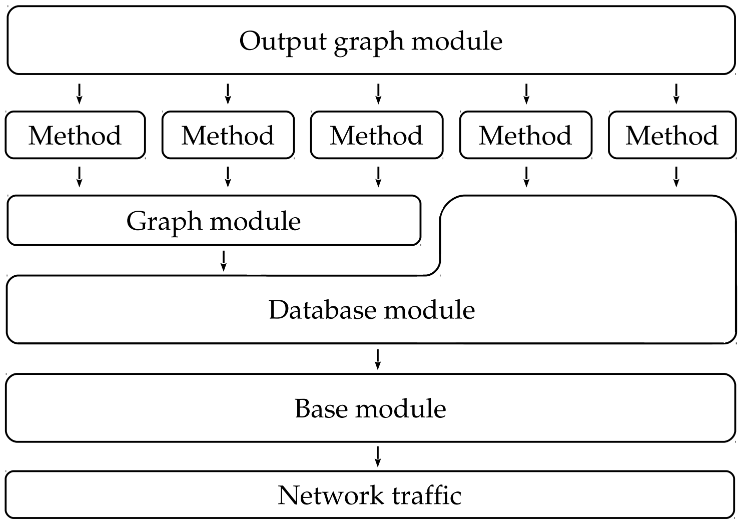

Figure 4.4: Architecture of the network topology mapping method.

The whole network topology mapping method comprises of five layers, shown on Figure 4.4, not counting the data source. The base layer captures network traffic and uses only relevant information. The information are stored in the database layer which provides a hierarchical database and key-value stores. The graph layer serves as an abstraction of the database for the method layer — it allows for selection of subsets from the database layer based on criteria provided by the method layer. The method layer hosts number of individual detection modules, each detecting one type of device. Data from the individual detection modules are then used in the `output_graph` module that produces pictures of the network topology in the graphviz [78] file format.

4.2.1 Base Layer

The base layer abstracts data captured from the concrete underlying events and sends it into the database layer.

The `connection_state_remove` event provides information about finished bidirectional flows. This event largely reports flow-based information. Although it also conveys data from DPI analysis — such as protocols and services used by the connection — we deliberately made the choice to ignore it in this case. There are two reasons for that. First, we wanted to realistically show what can be done without DPI and what cannot be done without DPI. Therefore, the method uses flow-based techniques to determine the services running on the connection. Second, the choice to use only flow-based data has been warranted by the wider context of the method's development, which is out of the scope of this thesis.

The `connection_SYN_packet` event provides information about SYN packets using DPI, in contrast with the previously mentioned event. The first SYN packet of a TCP connection conveys information vital to TCP fingerprinting [79]. Three attributes of the SYN packet are processed — IP TTL, SYN packet size, initial TCP window size. Although these information are not in the TCP *payload* per se, they are not detected by flow-based network traffic monitors and DPI capabilities are necessary.

The information from the `connection_SYN_packet` event are stored in an expiring table. There it waits up to a few minutes for the relevant connection to be closed and the information from both events are then paired and sent to the database layer. In case of a long connection, the SYN packet table entry will have been expired by the time it can be paired with the closed connection. This is of no great concern as the method doesn't need 100% of connections to be paired with the SYN packet. The reason for pairing the information is that the network topology mapping method will harbor an individual detection method that specifically needs this pairing in the future. In the context of this thesis, it just shows what can be easily done in Bro. Figure 4.5 shows the correlation.

```
event connection_state_remove(c: connection) {
        local corrl_dat_tmp : correlated_syn_data;
        if (c$id in corr_syn_tbl) {
                corrl_dat_tmp = corr_syn_tbl[c$id];
                delete corr_syn_tbl[c$id];
                database::profiles_table_update(c, corrl_dat_tmp, T);
        } else {
                database::profiles_table_update(c, corrl_dat_tmp, F);
        }
}
```

Figure 4.5: Code performing the correlation of L3 and L4 data. A part of the `profilecapture.bro` script.

The base layer captures DNS queries and sends them to the database layer. There is currently no module at the method layer to take advantage of DNS query data. This is not to be taken as deficiency in the method's implementation for this thesis. The opposite is true — the network topology mapping method is planned to be further developed for a specific purpose that is anticipated to be in need of DNS query data. The current method's implementation is designed with this in mind and is capable of DNS query capture well in advance. Data about DNS queries are provided by the `dns_request` event, which is clearly in the domain of DPI.

The last group of events used for data capture are `DHCP::dhcp_ack`, `arp_reply`, `dhcp_request`, and `dhcp_inform`. These events use DPI to provide information about DHCP and ARP messages. The base layer uses them to extract information about MAC–IP address pairings. There are flow-based network traffic monitors that are able to extract MAC–IP pairings directly from the network frames. Bro lacks this ability [80]. The reason for that is that Bro is intended to be used at network borders, presumably resulting in seeing just the two neighboring routers. This assumption proved to be true on the ICS MU network the testing was performed on. Nevertheless, we consider the lack of this feature to be a deficiency.

4.2.2 Database Layer

The database layer accepts updates from the base layer and offers the data to the graph and method layers. The database layer itself does no processing.

Data are stored in two ways — in a hierarchical database and in key-value stores. The `table`, `set`, and `record` types are used for that. Items in the databases automatically expire so that out-of-date data doesn't pollute the analysis.

The hierarchical database stores statistics about pairs of communication from the destination to the source. Some of the statistics are numbers of transmitted packets, used ports, and number of unidirectional and bidirectional flows. Unidirectional flow is regarded as communication without reply in this context, while bidirectional flow signifies communication where reply followed a request. The source is deemed to be the party that initiated the communication. Since the statistics are stored in several instances, aggregated by different L4 protocols, the database has a hierarchical structure.

The key-value stores save other data, such as TCP fingerprints, TTL values, and IP–MAC pairs.

The key-value stores are accessed directly by the method layer. The hierarchical database can be accessed through the graph layer.

4.2.3 Graph Layer

The graph layer allows the method layer to query data from the hierarchical database. It can filter the data based on criteria evaluated deep in the hierarchy. Since the hierarchical database contains *pairs* of communication, the graph layer can treat the pairs as edges in a graph. For instance, the `filter_profiles_subgraph_by_addr_src()` function selects the directed graph component accessible from the node specified by an IP address.

The functions receive and return the same format of data and can be easily chained. Since Bro uses reference type variables, data is not copied in the process, only the references are. This puts a constrain on the method layer not to change the data, because the only original copy is always referenced. We don't consider this to be an issue. The real-time processing deserves as much performance as it can get and careful handling of the data is not difficult in the network topology mapping method.

4.2.4 Detection modules

The following individual detection modules are used in the whole network topology mapping method:

- DNS client/server identification,
- DHCP client/server identification,

- FTP client/server identification,
- HTTP client/server identification,
- SSH client/server identification,
- NTP client/server identification,
- printer client/server identification,
- email client/server identification,
- router identification,
- NAT identification,
- operating system identification.

Operating system identification is particularly useful as one of the techniques of NAT identification. As nearly any network contains at least a router or a router with NAT, the last three methods (router, NAT, OS detection) are significantly more important than the others. And as it turns out, they also need DPI capabilities more than the other methods.

The two following subsections describe each detection module. The modules can be categorized as flow-compliant and DPI-dependent.

4.2.5 Flow-compliant Version

DNS Client/Server Identification

DNS client/server detection is based on the existence of a bidirectional flow to the port 53 on the server.

DHCP Client/Server Identification

DHCP client/server detection is based on the existence of UDP communication from the port 68 on the client to the port 67 on the server.

FTP Client/Server Identification

FTP client/server detection is based on the existence of a TCP bidirectional flow to the port 21 on the server.

FTP uses port 21 for transmission of control data and port 20 for transmission of file data. Monitoring communication of port 20 would be redundant in this detection method, hence only port 21 is monitored.

HTTP Client/Server Identification

HTTP client/server detection is based on the existence of a TCP bidirectional flow to at least one of the ports 80, 443, 8080, 8443 on the server, as these are the most commonly used HTTP and HTTPS ports.

SSH Client/Server Identification

SSH client/server detection is based on the existence of a TCP bidirectional flow to the port 22 on the server.

NTP Client/Server Identification

NTP client/server detection is based on the existence of a UDP bidirectional flow to the port 123 on the server.

Printer Client/Server Identification

Printer client/server detection is based on the existence of a bidirectional flow to at least one of the ports 515, 631, 9100 on the server, as these are the most commonly used printer service ports.

Email Client/Server Identification

Email client/server detection is based on the existence of a bidirectional flow to at least one of the ports 25, 110, 143, 465, 587, 993, 995 on the server, as these are the most commonly used email service ports.

4.2.6 DPI-enhanced Version

While the above detection modules don't need DPI at all, the other detection modules cannot work without DPI.

DNS Query Monitoring

DNS Query Monitoring is not done by any detection module yet, as stated in the description of the base layer earlier. However, the data is already captured by the base layer and saved by the database layer. Therefore, we consider this functionality being on the same level of importance as already implemented detection modules.

DNS query monitoring can help identify various classes of devices. For example, computers running Ubuntu usually contact `security.ubuntu.`

`com` some time after start. Computers running Microsoft Windows usually contact `ctldl.windowsupdate.com`. Identification of these operating systems can be also done using the TCP fingerprinting method, described later. The DNS query monitoring method is, however, able to identify other types of devices that may look too similar to the TCP fingerprinting method. For instance, some Xerox printers try to connect to `xeroxdiscoverysupernode1.com` through `xeroxdiscoverysupernode3.com` domains [81]. Android has similar TCP fingerprints to GNU/Linux but can be identified by looking up `android.clients.google.com`. Furthermore, some devices look up manufacturer-specific addresses, such as `update.sonymobile.com`.

It is obvious that this detection method doesn't work without DPI.

Bro allows implementation of DNS query monitoring in an elegant manner. The `dns_request` event provides data about DNS requests in a structured way. The method is interested only in the `query` field and others are ignored.

Operating System Detection Using TCP Fingerprinting

Passive operating system fingerprinting requires data not present in flow-based network traffic monitors: TCP SYN packet size, initial TCP window size, and IP TTL value. In the context of the network topology mapping method, passive OS fingerprinting is most useful for detection of routers hiding multiple devices behind a NAT.

C. Smith and P. Grundl [82] recommend use of two of the three aforementioned attributes for fingerprinting. S. Bellovin [83] uses IP ID sequences to discover hosts behind NAT. Some operating systems use monotonous series of IP ID values, while others use randomly scattered values. NAT detection then relies of observation of multiple parallel series or of a monotonous series with scattered values. However, Bellovin shows the problems in IP ID-based detection. Most notably, packet sampling and discontinuous monitoring make reliable detection difficult. Nmap [84] has a large database of device and OS fingerprints. Empirical results on ICS MU network showed unreliability of nmap's guesses. Moreover, nmap mainly uses an active approach to fingerprinting while my work requires a passive one. A new fingerprinting strategy without these shortcomings was devised for the network topology mapping method.

The following text describes how a list of known fingerprints was created and how these known fingerprints are then used in the network topology mapping method.

Building The Fingerprint List

A static list of well known fingerprints is necessary for actual detection of a yet-unknown operating system. The following method of building such a list was proposed. This was a one-off procedure and only its result is incorporated into the network topology mapping method. Figure 4.6 shows a summary of the procedure.

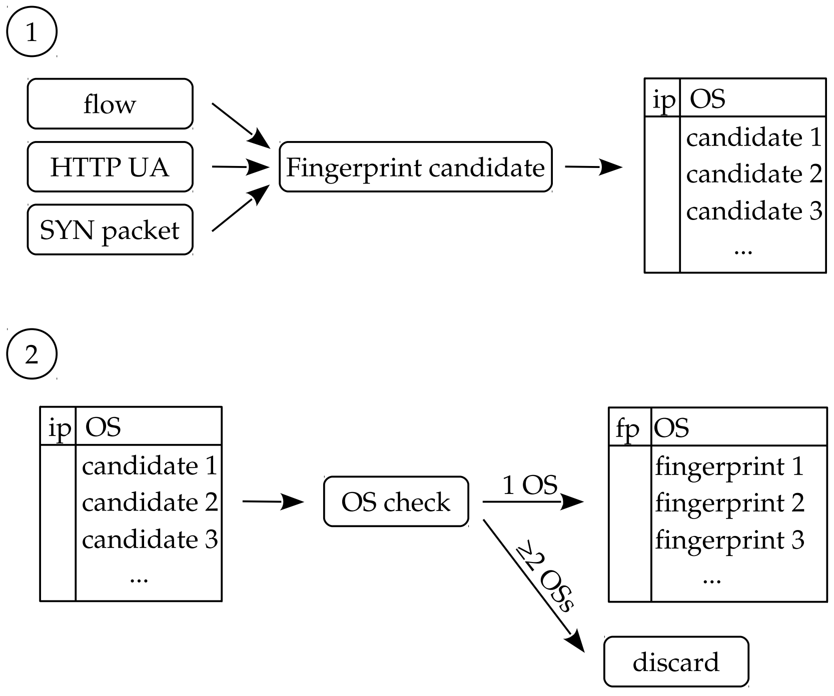

Figure 4.6: The principle of fingerprint discovery. First, L3, L4, L7 data is correlated. Second, consistent fingerprints are selected.

First, 883540 connections on the ICS MU network were captured in the form of fingerprint candidates. L3, L4, and L7 data was correlated to capture an HTTP user agent and the first TCP SYN packet of the corresponding connection. The user agents were then categorized into groups of operating systems running the application that sent the user agent. Most applications directly or indirectly imply the used operating system in the user agent string. Those user agents that do not do so were assigned to a special, unknown, OS category.

Second, the fingerprint candidates from the previous step were processed into a database of IP addresses and associated fingerprints. If an IP address used user agents belonging to more than one OS, the host was marked as "do not process further". This was necessary to filter data that may be inaccurate. For instance, a faked user agent still associates to the underlying operating system's true fingerprint. Some users choose to fake the user agent of their web browser. However, not all applications expose the fake user agent and usually run out of users' control. The motivation behind discriminating against IP addresses exhibiting multiple operating systems was to remove incorrect pairings of L3/L4 fingerprints and corresponding operating systems.

Third, the data from the previous step was processed in hourly segments. One reason for that was hourly segmentation of Bro logs. The other reason was to allow the use of data from computers that were rebooted into another operating system. The hour in which the reboot occurred would be marked as "do not process" but the hours before and after that would be used by the algorithm, provided no more reboots occurred for that IP.

Fourth, the hourly data from the previous step was merged and sorted. Further, it was manually inspected, cleaned, and formatted. Although the input dataset is rather large, the output is small. The results are shown in Table 4.1.

The result is a set of fingerprints that can be used to determine the operating system used to make a connection. Use of multiple operating systems on the same IP address at the same time signifies the presence of NAT. The dataset used for OS detection is based on real, recent data and thus works well on contemporary networks. Although HTTP information had to be parsed for building of the fingerprint dataset, subsequent detection only needs data from the first TCP packet. Still, these data — TCP SYN packet size, initial TCP window size, and IP TTL value — are not reported by flow-based network traffic monitors and DPI-capable network traffic monitor is necessary for the detection.

It is worth noting that such a detection is not perfect. However, it is not realistically possible to get a large number of fingerprints with guaranteed correct mapping to operating systems. The described method thus worked on a best-effort basis, using data with the highest degree of probable correctness.

IP	IP TTL	TCP Window Size	SYN packet size	OS
IPv4	128	16384	48	windows
IPv4	128	22592	48	windows
IPv4	128	60984	48	windows
IPv4	128	63443	52	windows
IPv4	128	64240	48	windows
IPv4	128	64512	48	windows
IPv4	128	65535	48	windows
IPv4	128	65535	52	windows
IPv4	128	8192	48	windows
IPv4	128	8192	52	windows
IPv4	128	8192	52	winphone
IPv4	64	12400	60	android
IPv4	64	13140	60	android
IPv4	64	13880	60	android
IPv4	64	14600	60	android
IPv4	64	14600	60	ios
IPv4	64	14600	60	linux
IPv4	64	16384	64	bsd
IPv4	64	29200	60	android
IPv4	64	29200	60	linux
IPv4	64	42900	60	android
IPv4	64	42900	60	ios
IPv4	64	4380	60	android
IPv4	64	5840	60	android
IPv4	64	5840	60	ios
IPv4	64	5840	60	linux
IPv4	64	64240	64	ios
IPv4	64	65535	60	android
IPv4	64	65535	60	linux
IPv4	64	65535	64	blackberry
IPv4	64	65535	64	ios
IPv4	64	65535	64	osx
IPv6	128	8192	72	windows
IPv6	64	14400	80	linux
IPv6	64	28800	80	linux
IPv6	64	65535	72	windows
IPv6	64	65535	84	osx
IPv6	64	8192	68	windows
IPv6	64	8192	72	windows

Table 4.1: Resulting TCP fingerprints. Extracting IP TTL, TCP window size and SYN packet size from TCP SYN packet makes it possible to detect the operating system running on the source of the SYN packet.

TCP Fingerprinting in the Topology Mapping Method

The OS detection module in the network topology mapping method uses the static list of well known fingerprints and corresponding operating systems shown in Table 4.1 to determine possible operating systems running on observed IP addresses.

Each connection that includes the fingerprint information (IP TTL, TCP window size, SYN packet size) is used in fingerprinting the originator of the connection.

A set of possible operating systems is assembled for each IP address. Some fingerprints are shared between multiple operating systems while others are more specific. Thus, each fingerprint has an associated set of possible operating systems. The detection module processes all observed fingerprints for the particular IP address and creates an intersection of the sets. The intersection resulting in one or more operating systems signifies that one of these OSes runs on the IP address. The intersection resulting in an empty set signifies use of multiple operating system in such a way that no fingerprint is shared between used OSes.

NAT Detection

NAT (Network Address Translation) is used to allow multiple devices operate on the same IP address. The internal side of the network typically has addresses in a private address range. The router keeps track of connections and accordingly translates addresses in packets passing between the two address spaces. There are several types of NAT. All NAT types behave similarly from the outside as far as NAT detection is concerned.

The NAT detection method identifies IP addresses used by NAT-enabled devices, such as routers or computers sharing the network connection.

Multiple detection techniques are used in this method:

- detection of multiple OS fingerprints,
- detection of multiple IP TTL values.

M. Zalewski [85] offers a superset of the aforementioned functionality. The p0f tool itself was, however, not used. The reason for a clean implementation is that p0f is not suitable in the wider context of this work. The network topology mapping method doesn't exist merely as a showcase for this thesis. It was originally created for a specific application favoring simplicity. The NAT detection method in its current form is sufficiently powerful, simple and language-agnostic. It also nicely integrates in the architecture on Figure 4.4, requiring only the very few selected kinds of data. p0f, on the other hand, has dependencies on the received data as well as on the programming language used — full packet capture is required out of box and it's programmed in C.

NAT detection based on operating system detection uses the aforementioned OS detection module. If an IP address is associated to a non-empty set of possible operating systems, this step doesn't detect NAT. Likewise, if there is no set of OSes associated to the IP, no NAT detection takes place. If the IP address is associated to an empty set of possible operating systems, NAT is detected in this step. This principle is described above in the text about OS detection.

NAT detection based on IP TTL values checks whether multiple originating IP TTL values are observed for an IP address. If they are, NAT is detected in this step. IP TTL values are analyzed only in TCP connections. Therefore, a traditional UDP/ICMP `traceroute` doesn't affect NAT detection. However, TCP `traceroute` does affect the detection and the host performing TCP `traceroute` is mistakenly detected as NAT. We admit this is a shortcoming of the method and yet have to find an acceptable solution. Use of TCP `traceroute` is not common.

There is one case where router detection doesn't work, i.e. reports a false negative: If two or more machines with similar OSes are at the same distance behind a NAT-enabled router which itself generates no additional traffic. Similar OSes have similar fingerprints and thus the first step of NAT detection fails, and uniform distance behind the otherwise mute NAT ensures the second step fails as well, since all originating IP TTL values are the same.

Router Detection

Routers (without NAT enabled) are detected based on IP–TTL mappings. If more than one IP address is used by one MAC address, the MAC address is considered to be a router.

This technique could also be used to detect ARP spoofing attacks. Anomaly detection is not incorporated into the network topology mapping method at the moment. The architecture of the the network topology mapping method is, however, open to the implementation of anomaly detection. This is an example of one feature it may have in the future.

Mere detection of routers is not the only task the module does. It is possible to observe the hop distance from the network probe to any IP address, provided some observed traffic originated from the IP address. The distance calculation is based on the difference between the observed IP TTL value and the presumed initial TTL (iTTL) value of the originating IP packets. The detection method groups IP addresses behind a router based on the observed hop distance. In other words, the groups reveal shadows of otherwise invisible topology behind routers.

4.2.7 Output Graph Module

The output graph module gathers information from all previously mentioned modules and builds a graph of the network's visible topology. The graph is saved as image in the graphviz file format, intended to be processed by `fdp` [78].

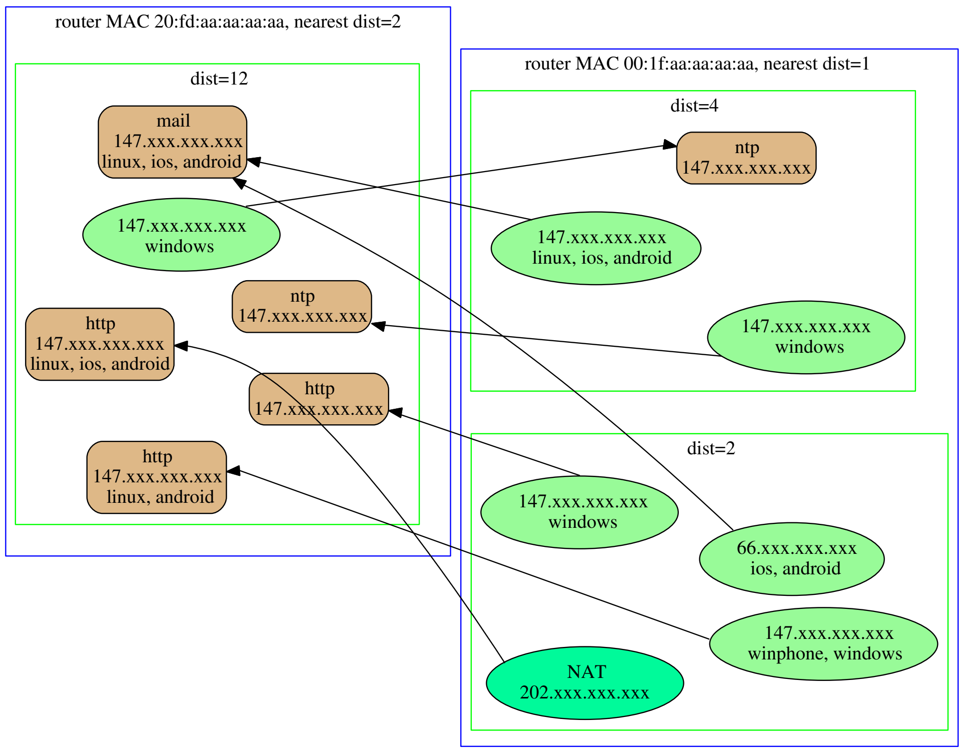

Figure 4.7: Sample graph in the format produced by the output graph module.

The output graph contains devices detected with the detection modules with the following criteria; for each pair of communicating addresses, at least one of the addresses has to be in the monitored subnet (Masaryk University network in my case) in order for the communication pair to be included in the output graph. Moreover, if the graph is too large to be displayed, a threshold is introduced; the communication pair is included only if the number of bidirectional flows for the pair is higher than the threshold. The threshold value is automatically tuned so that the output graph is not too large. Bidirectional flow means a communication where a response follows a request. Detected NATs are added as lone nodes if they didn't make it through the filter otherwise.

Routers are displayed as boxes in the output graph. Devices behind a router are displayed as nodes inside the corresponding box. Groups of devices based on the observed distance behind the router are displayed in nested boxes. The principle is visually explained on Figure 4.7.

4.3 Evaluation of the Network Topology Mapping Method

The network topology mapping method detects topology of a network. This is useful in networks with no prior knowledge of their topology. The method is intended to be used by security administrators in environments requiring quick construction of computer networks and short timespans between planning and usage. The possibilities are vast but a simple example may be construction of a network for a conference, a fair, or a similar event. Other class of possible uses is where the data from the network topology mapping method is fed into an anomaly detection method, analysing suspicious changes in the network's topology.

The method detects well-known services used on the network, e.g., DNS, DHCP, FTP, HTTP, SSH, NTP, printers, and email servers. It also detects how servers and clients of these services connect together through routers. NATs are detected, as they may be of special interest for security administrators, especially if usage of NATs is prohibited on the monitored network. Operating systems are also detected.

The method detected misconfigured devices on our network, by showing us which DNS server was used by which host.

Information is presented using pictures of the topology. A user of the method therefore doesn't have problems interpreting the results.

4.4 Comparison of DPI-enhanced and Flow-based Method

This section uses the method introduced in section 4.2 to show practical difference of capabilities between flow-based approach and DPI-enhanced approach to the same detection method.

The flow-compliant subset of the method detects various types of servers and clients. There is no way of detecting operating systems and NATs. Router detection is debatable since some flow-based network traffic monitors do capture MAC addresses and inability of Bro to do so is just a technical limitation of Bro. Also, there is not a single feature supported by the flow-compliant method that is not supported by the DPI-enhanced method.

The DPI-enhanced method also detects operating systems and NATs, on top of client and server detection. Using good enough DPI-enabled network traffic monitor doesn't limit anyone from using flow-oriented techniques.

Table 4.2 shows features supported by either version of the method. It is clear that choosing a DPI-enabled network traffic monitor is advantageous, compared to the flow-compliant version.

Method	Flow-compliant	DPI-enhanced
DNS	Yes	Yes
DHCP	Yes	Yes
FTP	Yes	Yes
HTTP	Yes	Yes
SSH	Yes	Yes
NTP	Yes	Yes
Printer	Yes	Yes
Email	Yes	Yes
Router	Partially[1]	Yes
NAT	No	Yes
OS	No	Yes
DNS Query	No	Yes

1 Without distance grouping; only in network security monitors that support MAC detection.

Table 4.2: Table of methods supported by flow-compliant and DPI-enhanced versions of the method.

The feature disparity between the two versions of the method are readily apparent in Figure 4.8. The methods barred from function by the lack of DPI-enhanced capabilities are the most important ones.

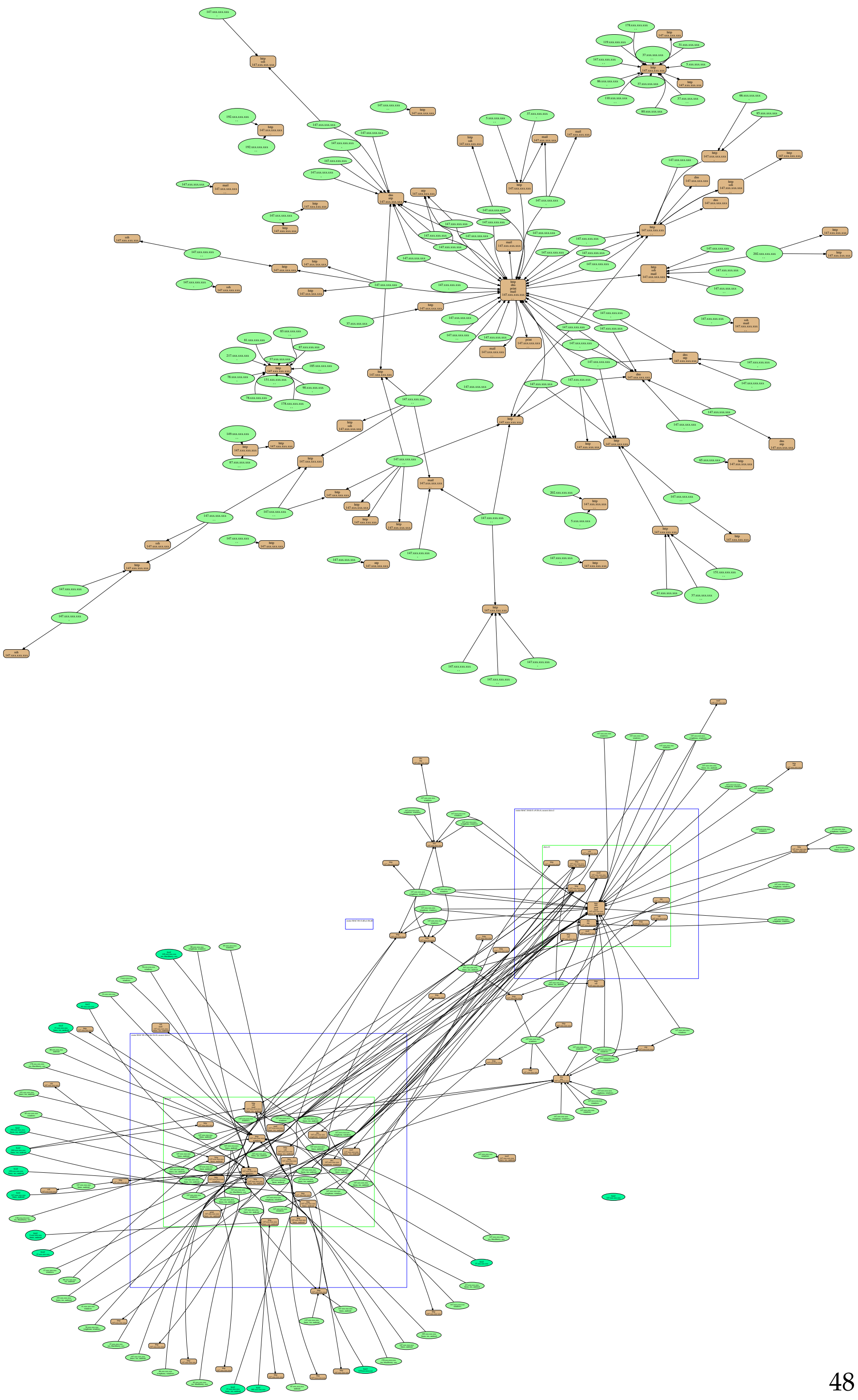

Figure 4.8: The picture shows differences in the overall detected topology. The top graph is generated only with information available using flow-compatible techniques in Bro, whereas the bottom graph takes advantage of important DPI-enhanced detection capabilities to show routers, depicted as boxes. The textual contents of the graphs do not matter.

4.5 Bro Development Experience

We chose Bro as the most viable network traffic monitor in the section 3.4. We have gathered experience with Bro during the development of the method described in the section 4.2.

Bro is user-friendly, with the exception of a few counterintuitive parts. For instance, nowhere in the official documentation is stated that the Bro language uses function scope for variables — like JavaScript and unlike Java, C#, C, and other languages. Similarly, nowhere in the documentation is explained that Bro uses so-called duck typing [86] for `record` types. Mistakenly using the wrong `record`-derived type doesn't cause the Bro interpreter to report an incompatible type right away. Instead, it produces seemingly nonsensical errors while trying to assign `records` with fields that don't match. We found out about this behavior in the course of programming in Bro. There is no relevant documentation we are aware of. However, once these features are documented, the described behavior no longer poses a problem. The scoping works as expected from a function-scoped language. The type system works as expected from a duck-typed language. Bro is more flexible than a strictly strongly typed language. The duck typing for records, for example, allows processing of externally-provided data without having to reference the original type declaration. Any compatible type declaration is equal under the duck typing scheme. The strong typing for non-record types prevents most type mismatch errors. All of this makes development easier.

Once the poorly documented details are found out, the Bro programming environment is pleasant and easy to use. The code of the network topology mapping method, which is a part of this thesis, is several thousand lines long and contains comments explaining various features. The code and the thesis can therefore serve as an addendum to the official documentation and help other developers at ICS MU.

We mentioned what is considered to be a deficiency in Bro in the section 4.2 — the inability to extract MAC and IP addresses from network frames. A third-party script [87] partially solves this problem by extracting this information from ARP and DHCP packets. Tests showed that this works well in small networks. However, further tests revealed that this approach doesn't work in large networks, such as the one at ICS MU. The probe at the ICS MU network is positioned between two routers. Any ARP and DHCP traffic happens behind one of the routers but doesn't reach the probe. The two routers surrounding the probe don't use ARP and DHCP between each other. We devised a partial workaround that uses a short shell script transforming tcpdump's output into Bro programming code. This generated code is then inserted into a prepared module that plugs into the architecture of the network topology mapping method. It sends observed mappings to the database, resulting in correct router detection and partial detection of devices behind routers. The network topology mapping method is flexible and works with even such partial data, displaying devices not detected behind a router anyway. Figure 4.9 explains the situation.

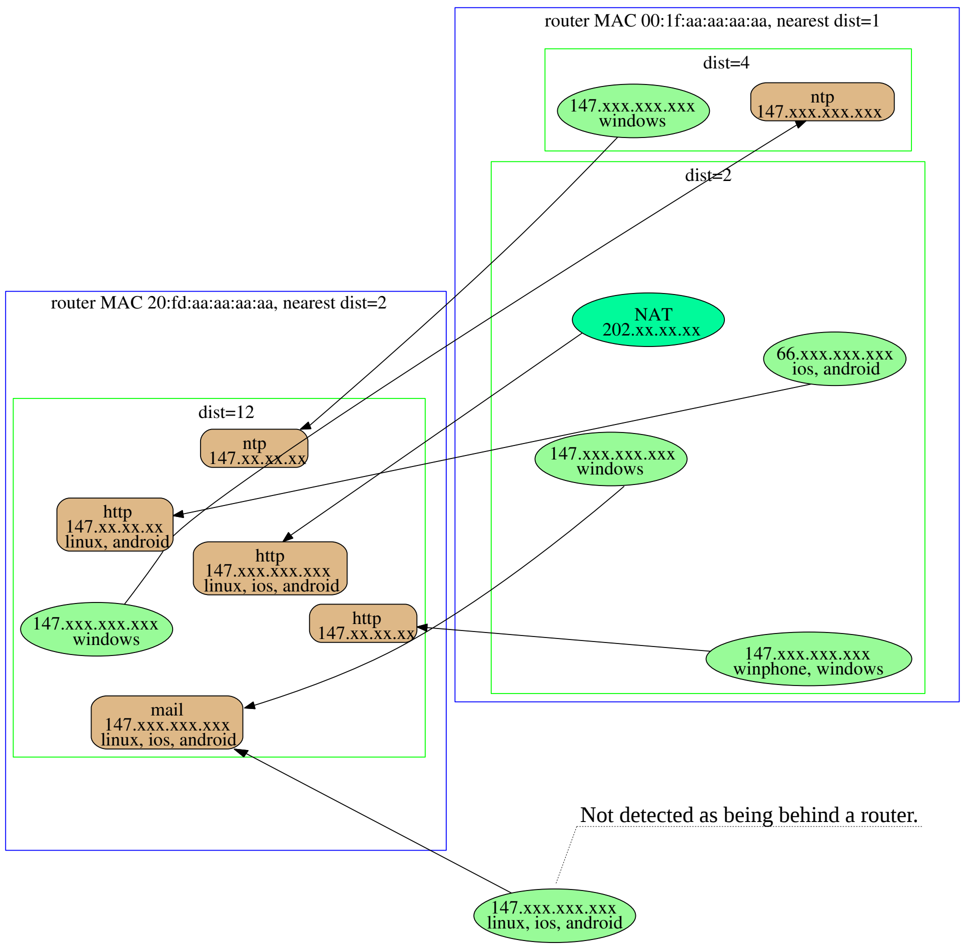
router MAC 00:1f:aa:aa:aa:aa, nearest dist=1
dist=4
147.xxx.xxx.xxx windows
ntp 147.xxx.xxx.xxx
dist=2
NAT 202.xx.xx.xx
66.xxx.xxx.xxx ios, android
147.xxx.xxx.xxx windows
147.xxx.xxx.xxx winphone, windows
router MAC 20:fd:aa:aa:aa:aa, nearest dist=2
dist=12
ntp 147.xx.xx.xx
http 147.xx.xx.xx linux, android
http 147.xxx.xxx.xxx linux, ios, android
147.xxx.xxx.xxx windows
http 147.xx.xx.xx
mail 147.xxx.xxx.xxx linux, ios, android
Not detected as being behind a router.
147.xxx.xxx.xxx linux, ios, android

Figure 4.9: The bottom node outside of any box is not detected as being behind a router.

4.6 Performance Measurements

Two kinds of performance measurements were made. One is a comparison between Bro and a flow-based monitor on a specific method implemented similarly in both traffic monitors. The second one is a standalone performance test showing how Bro with the network topology mapping method performs.

4.6.1 Performance Comparison

A relatively simple method was implemented in two network traffic monitors — Bro and FlowMon exporter (flowmonexp) [88]. The method detects expired SSL certificates seen in the monitored traffic.

The Bro version used the built-in script `expiring-certs.bro` included in the default Bro distribution.

The other version is a plugin built on top of the otherwise flow-based flowmonexp program. The plugin has been developed by Martin Bajaník in his Bachelor's thesis [89].

It is worth noting that implementing this functionality in Bro was trivial, whereas the flowmonexp version took a substantial amount of work.

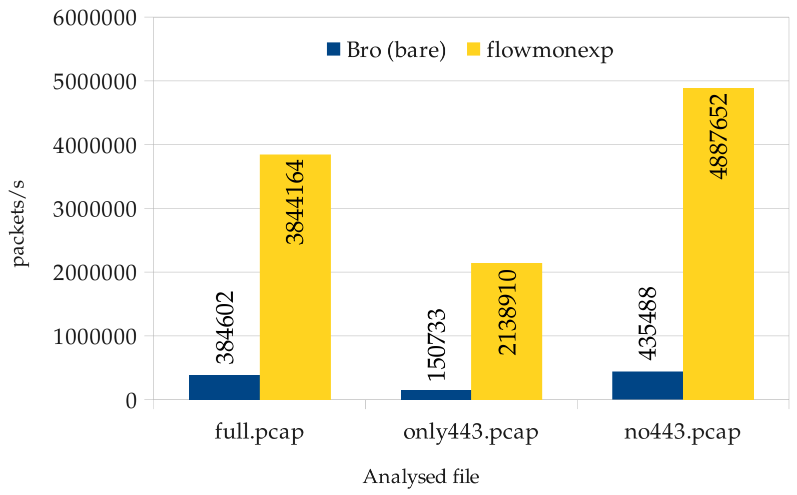

Figure 4.10: Performance of Bro in bare mode and flowmonexp in packets/s.

A capture of real traffic was taken, `full.pcap`. Two more files were filtered from `full.pcap`. The `only443.pcap` contains only packets to or from the port 443. The `no443.pcap` contains only packets that are neither to or from the port 443. Table 4.3 lists size of the PCAP files. Each file was cached into RAM before Bro performance measurement. In case of flowmonexp, a specialized benchmark plugin which explicitly loaded the PCAP into RAM was used.

Bro offers so-called bare mode in which only modules necessary for the task run. The certificate detection module was designed to be compatible with bare mode.

File	Number of packets	Size in MiB
full.pcap	409950	254
only443.pcap	63153	54
no443.pcap	346797	201

Table 4.3: PCAP files used for performance measurements.

	Bro		Bro (bare)		flowmonexp	
File	$\bar{x}$	s	$\bar{x}$	s	$\bar{x}$	s
full.pcap	573.9	4.6	1999.0	21.1	19980.0	2079.7
only443.pcap	875.4	15.3	1081.2	102.4	15342.0	1544.2
no443.pcap	506.8	5.5	2117.3	200.3	23763.5	2697.0

Table 4.4: Average ($\bar{x}$) and standard deviation (s) of performance in Mbit/s.

Each piece of software was tested 5 times on each PCAP file and the average and standard deviation values were calculated. Table 4.4 and Table 4.5 show the performance of the two implementations in Mbit/s and packets/s respectively.

	Bro		Bro (bare)		flowmonexp	
File	$\bar{x}$	s	$\bar{x}$	s	$\bar{x}$	s
full.pcap	110415	880	384602	4054	3844164	400134
only443.pcap	122041	2128	150733	14273	2138910	215279
no443.pcap	104228	1122	435488	41189	4887652	554715

Table 4.5: Average ($\bar{x}$) and standard deviation (s) of performance in packets/s.

Figure 4.10 shows a graph comparing performance of Bro in bare mode with flowmonexp. It is apparent that Bro is several times slower than flowmonexp. Therefore, Bro may not be the best choice for production-grade method implementations. Its status as a great prototyping tool remains unchallenged, however.

4.6.2 Standalone Bro Performance

The network topology mapping method has been tested on the same `full.pcap` file as mentioned in the previous subsection. Periodic timers in the method were removed for the test and were instead triggered exactly once using the `bro_done` event that triggers upon finishing reading the input PCAP file.

The PCAP file was cached into RAM before the performance measurement. The measurement was repeated 5 times and the mean value was calculated. Results showing throughput in Mbit/s and in packets/s are shown in Table 4.6.

Mbit/s		packets/s	
$\bar{x}$	s	$\bar{x}$	s
181.2	6.7	34931	1337.0

Table 4.6: Average ($\bar{x}$) and standard deviation (s) of performance in packets/s and Mbit/s of the network topology mapping method.

By its nature, the network topology mapping method still works even when packets are dropped due to high load. The amount of traffic on the network monitored by the development server is sometimes several times as high, with no bad effect on the detection.

All performance measurements were made on a machine with Intel Xeon E5530 at 2.4 GHz, 12 GB of DDR3 RAM, and 64-bit Debian 7.5.

4.7 Desirability of DPI

We used a real-world scenario to show the importance of deep packet inspection. The section 4.4 shows how a concrete method cannot work properly without DPI.

The Institute of Computer Science at Masaryk University wields great expertise in flow-based methods and network monitoring techniques. In sections 4.2 through 4.5 we show that choosing the right DPI-enabled traffic monitor doesn't bar the developer from using flow-based techniques. This allows existing experience to not be wasted. More importantly, an appropriate DPI-enabled traffic monitor allows for experimentation in development of flow-based methods, as well as experimentation in adding DPI-enhanced features. This makes the development comparatively quicker than using traditional approaches. Only the final version of a method has to be implemented in a flow-based network traffic monitor. DPI-enhanced analysis sometimes is possible in flow-based network traffic monitors but requires a substantial amount of work. The traditional approach to development of new methods has been to program all iterations for the flow-based network traffic monitor, which takes more time. We hope this thesis will change that and saves valuable developers' time.

Given all the previously stated facts, my stance is that DPI is a desirable feature in network traffic monitors.

4.8 Suitability of Bro

We implemented a useful method in the Bro Network Security Monitor. The method was designed according to requirements of ICS MU. Bro was therefore tested in circumstances specific to ICS MU. Bro withstood the test well and proved to be an indispensable tool for effective development of fully functional prototypes. Although Bro has lower performance than some flow-based network traffic analyzers and is thus not appropriate for use in production environment, other characteristics make it a good prototyping tool. Namely, the memory-safe programming language and event-driven architecture make rapid development possible.

Bro meets the expectations set in the section 3.4. We showed that implementation of a non-trivial method in Bro is possible and fairly easy. This proves the point that Bro is ready to be used as a prototyping tool for development and testing of novel methods at ICS MU.

Chapter 5

Conclusion

This work shows how deep packet inspection is useful in network traffic analysis and how Bro is useful in development of new detection methods. ICS MU currently uses only flow-based network traffic monitoring. This thesis makes the first step in introducing DPI to ICS MU, along with a new way of method prototyping.

We gave an introduction to network traffic monitoring approaches and presented their individual implementations. We chose Bro Network Security Monitor as the network traffic monitor suitable for the needs of ICS MU. Bro is able to perform deep packet inspection and provides extensibility through a specialized programming language. We implemented a network traffic analysis method in the chosen network traffic monitor. Finally, we analyzed properties of the chosen network traffic monitor on real-life problems provided by the traffic analysis method. The traffic analysis method automatically discovers topology of the monitored network, e.g. types of devices and services running on the network, and how they connect together.

We show Bro serves well as a prototyping tool. Our positive experience is not limited to the network topology mapping method. A high-profile OpenSSL vulnerability named Heartbleed was published during work on this thesis [90]. We used Bro to analyze a typical successful attack. We then implemented an experimental detection method in a matter of hours after becoming aware of Heartbleed. The detection method prototype allowed us to start protecting our assets in a very short time. The flexible nature of Bro allowed us to go a week back in time with the attack detection, as we analyzed existing Bro logs using the same detection method implemented in Bro.

We conclude that Bro is a viable tool for efficient development of network traffic analysis methods using both flow-based and DPI-based techniques. This work documents how to develop for Bro and provides several thousand lines worth of commented source code in the Bro programming language.

Bibliography

[1] Cottrell, L.: Passive vs. Active Monitoring [online]. [cit. 2014-03-06]. URL <https://www.slac.stanford.edu/comp/net/wan-mon/passive-vs-active.html>

[2] Worrall, A.; Carter, B.; Widley, G.: Network monitor and method. 2008 [cit. 2014-03-06], uS Patent 7,411,946. URL <http://www.google.com/patents/US7411946>

[3] CaptureSetup/Ethernet - The Wireshark Wiki [online]. [cit. 2014-03-06]. URL <http://wiki.wireshark.org/CaptureSetup/Ethernet>

[4] Cisco Systems, Inc.: Catalyst Switched Port Analyzer (SPAN) Configuration Example - Cisco [online]. [cit. 2014-03-06]. URL <http://www.cisco.com/c/en/us/support/docs/switches/catalyst-6500-series-switches/10570-41.html>

[5] Leong, P.: Ethernet 10/100/1000 Copper Taps, Passive or Active? [online]. [cit. 2014-03-06]. URL <http://www.lovemytool.com/blog/2007/10/copper-tap.html>

[6] Matityahu, E.; Shaw, R.; Carpio, D.; aj.: Gigabits zero-delay tap and methods thereof. 2011 [cit. 2014-03-06], uS Patent App. 13/034,730. URL <http://www.google.com/patents/US20110211446>

[7] Datacom Systems: Choosing a Network TAP [online]. [cit. 2014-03-06]. URL <http://justnetworktaps.com/article_info.php?articles_id=3>

[8] JDSU Storage Network Test: Understanding Fibre Optic Network Tapping [online]. [cit. 2014-03-06]. URL <http://www.jdsu.com/ProductLiterature/Understanding-Fiber-Optic-Network-Tapping-white-paper-30162800.pdf>

[9] PE210G2BPi9 Dual Port Fiber 10 Gigabit Ethernet PCI Express Bypass Server Adapter Intel® based [online]. [cit. 2014-03-02].
URL <http://www.silicom-usa.com/upload/Downloads/Product_95.pdf>

[10] Shepard, T. J.: TCP Packet Trace Analysis. Technical report, Laboratory for Computer Science, Massachusetts Institute of Technology, 1991 [cit. 2014-03-06].
URL <http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-494.pdf>

[11] Baecher, P.; Koetter, M.; Holz, T.; aj.: The Nepenthes Platform: An Efficient Approach to Collect Malware. In *Recent Advances in Intrusion Detection*, editace D. Zamboni; C. Kruegel, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006 [cit. 2014-03-06], ISBN 978-3-540-39723-6, s. 165–184, doi:10.1007/11856214_9.
URL <http://dx.doi.org/10.1007/11856214_9>

[12] Balas, E.; Viecco, C.: Towards a third generation data capture architecture for honeynets. In *Information Assurance Workshop, 2005. IAW '05. Proceedings from the Sixth Annual IEEE SMC*, June 2005 [cit. 2014-03-06], s. 21–28, doi:10.1109/IAW.2005.1495929.

[13] INTECO-CERT: TRAFFIC ANALYSIS WITH WIRESHARK [online]. [cit. 2014-03-06].
URL <http://www.csirtcv.gva.es/sites/all/files/downloads/cert_trafficwireshark.pdf>

[14] McCanne, S.; Jacobson, V.: The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, Berkeley, CA, USA: USENIX Association, 1993 [cit. 2014-03-06], s. 2–2.
URL <http://dl.acm.org/citation.cfm?id=1267303.1267305>

[15] Arcas, G.: WireShnork - A Snort plugin for Wireshark [online]. [cit. 2014-03-06].
URL <http://honeynet.org/node/790>

[16] Development/LibpcapFileFormat - The Wireshark Wiki [online]. [cit. 2014-03-06].
URL <http://wiki.wireshark.org/Development/LibpcapFileFormat>

[17] D.2. tshark: Terminal-based Wireshark [online]. [cit. 2014-03-06].
URL <https://www.wireshark.org/docs/wsug_html_chunked/AppToolstshark.html>

[18] TCPDUMP/LIBPCAP public repository [online]. [cit. 2014-03-06].
URL <http://www.tcpdump.org/>

[19] Wireshark · About [online]. [cit. 2014-03-06].
URL <https://www.wireshark.org/about.html>

[20] Kumar, S.; Sehgal, R.; Bhatia, J. S.: Hybrid honeypot framework for malware collection and analysis. In *Industrial and Information Systems (ICIIS), 2012 7th IEEE International Conference on*, Aug 2012 [cit. 2014-03-06], s. 1–5, doi:10.1109/ICIInfS.2012.6304786.

[21] Lin, P.-C.; Lin, Y.-D.; Lee, T.; aj.: Using string matching for deep packet inspection. 2008 [cit. 2014-03-06].
URL <http://speed.cs.nctu.edu.tw/~ydlin/string\%20matching.pdf>

[22] 2.3 Decoder and Preprocessor Rules [online]. [cit. 2014-03-06].
URL <http://manual.snort.org/node18.html>

[23] Franklin, M.; Crowley, P.; Hadimioglu, H.; aj.: *Network Processor Design: Issues and Practices*. The Morgan Kaufmann Series in Computer Architecture and Design, Elsevier Science, 2005 [cit. 2014-03-06], ISBN 9780080512501.
URL <http://books.google.com/books?id=WFHoNtoyd14C>

[24] INVEA-TECH a.s.: FlowMon Probe Models List [online]. [cit. 2014-03-06].
URL <https://www.invea.com/data/flowmon/flowmon_probe_specification_en.pdf>

[25] KELLY, J.: *An Examination of Pattern Matching Algorithms for Intrusion Detection Systems [online]*. Master's thesis, Ottawa-Carleton Institute for Computer Science, Carleton University, Canada, 2006 [cit. 2014-03-03].
URL <http://www.jameskelly.net/mcs/thesis.pdf>

[26] ŠÍMA, T.: *Měření HTTP a HTTPS provozu pomocí IP toků [online]*. Bachelor's thesis, Faculty of Informatics, Masaryk University, Czech Republic, 2014 [cit. 2014-03-03].
URL <https://is.muni.cz/th/374541/fi_b/>

[27] Snort :: Home Page [online]. [cit. 2014-03-06].
URL <http://www.snort.org/>

[28] Suricata Open Source IDS / IPS / NSM engine [online]. [cit. 2014-03-06].
URL <http://suricata-ids.org/>

[29] Ritter, J.: ngrep - network grep [online]. [cit. 2014-03-08].
URL <http://ngrep.sourceforge.net/>

[30] The Bro Network Security Monitor [online]. [cit. 2014-02-05].
URL <http://www.bro.org/documentation/overview.html>

[31] Paxson, V.: Bro: A System for Detecting Network Intruders in Real-time. *Comput. Netw.*, 1999: s. 2435–2463, ISSN 1389-1286, doi:10.1016/S1389-1286(99)00112-7.
URL <http://dx.doi.org/10.1016/S1389-1286(99)00112-7>

[32] Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information Types and attributes [online]. [cit. 2014-02-19].
URL <http://tools.ietf.org/search/rfc7011>

[33] INVEA-TECH a.s.: FlowMon Data Retention [online]. [cit. 2014-03-06].
URL <https://www.invea.com/data/flowmon/flowmon_data_retention_pb_en.pdf>

[34] Claise, B.: Cisco Systems NetFlow Services Export Version 9 [online]. [cit. 2014-03-06].
URL <https://tools.ietf.org/html/rfc3954>

[35] Trammell, B.; Claise, B.: Flow Aggregation for the IP Flow Information Export (IPFIX) Protocol [online]. [cit. 2014-03-06].
URL <https://tools.ietf.org/html/rfc7015>

[36] nProbe [online]. [cit. 2014-03-06].
URL <http://www.ntop.org/products/nprobe/>

[37] softflowd - A software NetFlow probe [online]. [cit. 2014-03-06].
URL <https://code.google.com/p/softflowd/>

[38] Christopher Inacio, B. T.: YAF: Yet Another Flowmeter [online]. [cit. 2014-04-07].
URL <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.368.3172&rep=rep1&type=pdf>

[39] INVEA-TECH: FlowMon – Network monitoring [online]. [cit. 2014-05-21].
URL <http://www.invea.com/en/products-and-services/flowmon>

[40] NFDUMP [online]. [cit. 2014-03-06].
URL <http://nfdump.sourceforge.net/>

[41] flowd - small, fast and secure NetFlow collector [online]. [cit. 2014-03-06].
URL <http://code.google.com/p/flowd/>

[42] IPFIXcol | Liberouter / Cesnet TMC group [online]. [cit. 2014-03-06].
URL <https://www.liberouter.org/technologies/ipfixcol/>

[43] Gates, C.; Collins, M.; Duggan, M.; aj.: More Netflow Tools for Performance and Security. In *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04, Berkeley, CA, USA: USENIX Association, 2004, s. 121–132.
URL <http://dl.acm.org/citation.cfm?id=1052676.1052691>

[44] Keuter, J.: Privilege Separation [online]. [cit. 2014-04-07].
URL <http://wiki.wireshark.org/Development/PrivilegeSeparation>

[45] Snort Syntax and Simple Rule Writing [online]. [cit. 2014-04-02].
URL <http://www.anotherchancecomputers.com/uncategorized/snort-syntax-and-simple-rule-writing/>

[46] The Basics [online]. [cit. 2014-04-02].
URL <http://manual.snort.org/node28.html>

[47] Esler, J.: Offset, Depth, Distance, and Within [online]. [cit. 2014-04-02].
URL <http://blog.joelesler.net/2010/03/offset-depth-distance-and-within.html>

[48] community-rules.tar.gz [online]. [cit. 2014-05-10].
URL <https://s3.amazonaws.com/snort-org/www/rules/community/community-rules.tar.gz>

[49] Writing Good Rules [online]. [cit. 2014-04-02].
URL <http://manual.snort.org/node36.html#testing_numerical_values>

[50] Esler, J.: Preprocessors [online]. [cit. 2014-04-02].
URL <http://manual.snort.org/node59.html>

[51] Biles, S.: Detecting the Unknown with Snort and the Statistical Packet Anomaly Detection Engine (SPADE) [online]. [cit. 2014-04-02].
URL <http://webpages.cs.luc.edu/~pld/courses/447/sum08/class6/biles.spade.pdf>

[52] Mahoney, M.: Network Anomaly Intrusion Detection Research at Florida Tech. [online]. [cit. 2014-04-02].
URL <http://cs.fit.edu/~mmahoney/dist/>

[53] AnomalyDetection: Home - Snort.AD [online]. [cit. 2014-04-02].
URL <http://www.anomalydetection.info/?home,1>

[54] Matthew Mahoney, P. C.: PHAD: Packet Header Anomaly Detection for Identifying Hostile Network Traffic [online]. Technical report, Department of Computer Sciences, Florida Institute of Technology, 2001 [cit. 2014-04-02].
URL <http://cs.fit.edu/~mmahoney/paper3.pdf>

[55] Matthew Mahoney, P. C.: Learning Nonstationary Models of Normal Network Traffic for Detecting Novel Attacks [online]. In *KDD '02 Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2002 [cit. 2014-04-02], s. 376–385.
URL <http://cs.fit.edu/~mmahoney/paper4.pdf>

[56] Guillon, B.: Snort Anomaly Detection [online]. [cit. 2014-04-02].
URL <http://www.engardelinux.org/modules/index/list_archives.cgi?list=snort-devel&page=0014.html&month=2010-09>

[57] Chatfield, C.: The Holt-Winters Forecasting Procedure. 1978: s. 264–279.
URL <http://www.jstor.org/discover/10.2307/2347162?uid=2&uid=4&sid=21104041062697>

[58] Houghton, N.: Single Threaded Data Processing Pipelines and the Intel Architecture [online]. [cit. 2014-04-02].
URL <http://vrt-blog.snort.org/2010/06/single-threaded-data-processing.html>

[59] Verplanke, E.: Understand packet-processing performance when employing multicore processors [online]. [cit. 2014-04-02].
URL <http://www.embedded.com/design/connectivity/4007065/Understand-packet-processing-performance-when-employing-multicore-processors>

[60] Suricata Rules [online]. [cit. 2014-04-02].
URL <https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Suricata_Rules>

[61] suricata/src at master [online]. [cit. 2014-04-02].
URL <https://github.com/inliniac/suricata/tree/master/src>

[62] Julien, V.: On Suricata performance [online]. [cit. 2014-04-02].
URL <http://blog.inliniac.net/2010/07/22/on-suricata-performance/>

[63] Types and attributes — Bro 2.2 documentation [online]. [cit. 2013-11-09].
URL <http://bro.org/sphinx/scripts/builtins.html>

[64] All Bro Scripts [online]. [cit. 2014-04-02].
URL <http://bro.icir.org/sphinx/scripts/scripts.html>

[65] Microsoft: HashSet<T> Class [online]. [cit. 2014-04-02].
URL <http://msdn.microsoft.com/en-us/library/bb359438.aspx>

[66] Microsoft: Dictionary<TKey, TValue> Class [online]. [cit. 2014-04-02].
URL <http://msdn.microsoft.com/en-us/library/xfhwa508\%28v=vs.110\%29.aspx>

[67] Microsoft: Classes and Structs (C# Programming Guide) [online]. [cit. 2014-04-02].
URL <http://msdn.microsoft.com/en-us/library/ms173109.aspx>

[68] Microsoft: Fields (C# Programming Guide) [online]. [cit. 2014-04-02].
URL <http://msdn.microsoft.com/en-us/library/ms173118.aspx>

[69] base/frameworks/cluster/main.bro [online]. [cit. 2014-05-12].
URL <https://www.bro.org/sphinx/scripts/base/frameworks/cluster/main.html>

[70] Deri, L.: nProbe: an Open Source NetFlow probe for Gigabit Networks. In *In Proc. of Terena TNC 2003*, 2003.
URL <http://luca.ntop.org/nProbe.pdf>

[71] Trammell, B.: YAF-derived flow meter for passive performance measurement [online]. [cit. 2014-04-07].
URL <https://github.com/britram/qof>

[72] NetFlow iptables module [online]. [cit. 2014-04-07].
URL <http://sourceforge.net/projects/ipt-netflow/>

[73] Lucente, P.: pmacct: steps forward interface counters [online]. [cit. 2014-04-07].
URL <http://www.pmacct.net/pmacct-stepsforward.pdf>

[74] Velan, P.; Krejci, R.: Flow Information Storage Assessment Using IPFIXcol. In *Dependable Networks and Services*, editace R. Sadre; J. Novotny; P. Celeda; M. Waldburger; B. Stiller, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, ISBN 978-3-642-30632-7, s. 155–158, doi:10.1007/978-3-642-30633-4_21.
URL <http://dx.doi.org/10.1007/978-3-642-30633-4_21>

[75] Lucente, P.: pmacct project: IP accounting iconoclasm [online]. [cit. 2014-04-07].
URL <http://www.pmacct.net/>

[76] Signature Framework [online]. [cit. 2014-05-10].
URL <https://www.bro.org/sphinx-git/frameworks/signatures.html>

[77] Database Models [online]. [cit. 2014-05-10].
URL <http://unixspace.com/context/databases.html>

[78] Graphviz - Graph Visualization Software [online]. [cit. 2014-05-10].
URL <http://www.graphviz.org>

[79] OS fingerprinting [online]. [cit. 2014-05-10].
URL <http://www.forensicswiki.org/wiki/OS_fingerprinting>

[80] Hall, S.: [Bro] Obtain src/dst mac addrs from connection record [online]. [cit. 2014-05-10].
URL <http://mailman.icsi.berkeley.edu/pipermail/bro/2012-January/005313.html>

[81] Hay, A.: XEROX PRINTER BEACONS AND THE IMPORTANCE OF DOCUMENTATION [online]. [cit. 2014-05-10].
URL <http://labs.opendns.com/2014/05/01/xerox-printer-beacons/>

[82] Craig Smith, P. G.: Know Your Enemy: Passive Fingerprinting [online]. [cit. 2014-05-14].
URL <http://old.honeynet.org/papers/finger/>

[83] Bellovin, S. M.: A Technique for Counting NATted Hosts [online]. [cit. 2014-05-14].
URL <https://www.cs.columbia.edu/~smb/papers/fnat.pdf>

[84] TCP/IP Fingerprinting Methods Supported by Nmap [online]. [cit. 2014-05-14].
URL <http://nmap.org/book/osdetect-methods.html>

[85] Zalewski, M.: p0f v3 [online]. [cit. 2014-05-14].
URL <http://lcamtuf.coredump.cx/p0f3/>

[86] Duck typing [online]. [cit. 2014-05-10].
URL <https://en.wikipedia.org/wiki/Duck_typing>

[87] bro-scripts/roam.bro [online]. [cit. 2014-05-10].
URL <https://github.com/bro/bro-scripts/blob/master/roam.bro>

[88] Velan, P.: FlowMon - IPFIX Export Plugin [online]. [cit. 2014-05-10].
URL <http://dior.ics.muni.cz/~velan/flowmon-export-ipfix/>

[89] Bajaník, M.: *Analyza sifrovanej prevadzky [online]*. Bachelor's thesis, Faculty of Informatics, Masaryk University, Czech Republic, 2014 [cit. 2014-05-10].
URL <http://is.muni.cz/th/396204/fi_b>

[90] Codenomicon: The Heartbleed Bug [online]. [cit. 2014-05-21].
URL <http://heartbleed.com>

Appendix A

List of attachments

The IS MU archive contains the following electronic attachments:

- Thesis.
- Source code and documentation for the network topology mapping method.
- Source code and documentation for the TCP fingerprint analyzer.

The PDF version of the thesis is licensed under the CC BY-SA 4.0 license. The source code and accompanying documentation are licensed under the CC0 1.0 license. License notices and full license text are bundled in the electronic attachments.