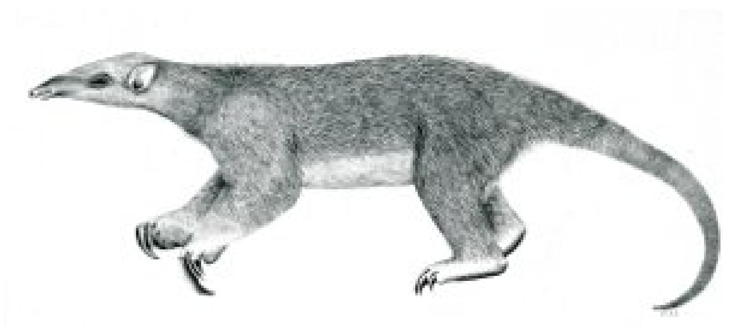
Tranalyzer 2.0 – Documentation

Flow based forensic and network troubleshooting traffic analyzer



Torben Rühl
Stefan Burschka

Contents

1 Introduction

Tranalyzer2 is a lightweight flow generator and packet analyzer designed for simplicity, performance and scalability. The program is written in C and built upon the *libpcap* library. It provides functionality to pre- and postprocess IPv4/IPv6 data into flows and enables a trained user to see anomalies and network defects even in very large datasets. It supports analysis with special bit coded fields and generates statistics from key parameters of IPv4/IPv6 Tcpdump traces either being live-captured from an Ethernet interface or one or several pcap files. The quantity of binary and text based output of Tranalyzer2 depends on enabled modules, herein denoted as **plugins**. Hence, users have the possibility to tailor the output according to their needs and developers can develop additional plugins independent of the functionality of other plugins.

1.1 Installing Tranalyzer2

1.1.1 Prerequisites

Do install **libpcap**, **libpcap-dev**, **automake** and **libtool**.

Ubuntu: `sudo apt-get install libpcap-dev libpcap0.8 automake libtool`

Red Hat and Fedora: `sudo yum install libpcap-devel libpcap automake libtool`

Suse: `yast -i ./libpcap-dev.version.rpm` or use yum as in Red Hat

Gentoo: You only need to install libpcap and automake version 1.11.1. If you already have an automake version higher than 1.11.1 installed, then you need to install 1.11.1 and run the following commands as root in the following order): `export WANT_AUTOMAKE=1.11`, `env-update`, `source /etc/profile`, `autoreconf`

Mac OS X: First you have to install Xcode[1] after this the command line tools[2] finally install Brew[3]. Then, run the command `brew install autoconf automake libtool`

1.1.2 Compiling altogether

Tranalyzer is available via its Sourceforge project page: `http://tranalyzer.sf.net` and `http://tranalyzer.com`. After download and extraction of the source files change to the root directory and type `./autogen.sh`. This script executes all necessary programs to configure and build the program and the standard plugins. If the full spectrum of Tranalyzer2 capability is required execute `./autogen_all.sh` or assemble your own build file. It is also possible to install the core or plugins separately. Just execute the *autogen.sh* script in the root folder of the plugin. The autogen_rall.sh script compiles all plugins under your directory, even the ones you developed yourselves.
If you have an older version of autoconf installed on your system, modify *configure.ac* in each plugins folder and replace the current version number with the one of your installation. You can get your version number by typing the command `autoconf -V`. Always make sure that the /.tranalyzer/plugins directory does not contain old plugins, otherwise tranalyzer2 may crash. If uncertain invoke rm /.tranalyzer/plugins/*.so and all plugins will be deleted.

1.1.3 Install Tranalyzer2 in /usr/local/bin

In order to install Tranalyzer2 in the standard directory for system dependent binaries, execute the autogen script with the option *install*. Tranalyzer is then accessible for all users from every directory by executing the following command: `tranalyzer`. Note that root rights are required. Ask your system administrator to install Tranalyzer if e.g. company policy does not grant general admin rights.

[1] Xcode is an IDE from Apple `https://developer.apple.com/xcode/`
[2] `https://developer.apple.com/xcode/`
[3] Brew is a packet manager for Mac OS X that can be found here: `http://mxcl.github.com/homebrew/`

1.1.4 PF_RING

Under certain circumstances e.g. large quantities of small packets, it the kernel might drop packets. This happens due to the normal kernel dispatching which is known to be inefficient for packet capture operations. The capturing process can be devised more efficiently by changing the kernel as in packet_mmap, but then a patched libpcap is required which is not available yet.[4] Another option is pf_ring. Its kernel module passes the incoming packets in a different way to the user process.[5]

Requirements

- Kernel version prior to 3.10. [6]
- All packages needed for building a kernel module, names are distribution-dependent
- A network interface which supports NAPI polling by its driver.
- optional: A network card which supports Direct Network Interface Card (NIC) access (DNA).[7]

Quick setup

Download PF_RING from a stable tar ball or development source at http://www.ntop.org/get-started/download/. In order to build the code the following commands have to executed in a bash window:

```
cd PF_RING/kernel
make && sudo make install
modprobe pf_ring
```

Figure 1: *building kernel module*

Tranalyzer2 requires at least libpfring and libpcap-ring whic can be installed the following way:

```
cd PF_RING/userland
cd lib
make && sudo make install
cd ..
cd libpcap
make && sudo make install
```

Figure 2: *basic userland*

You may like to install other tools such as tcpdump. Just install it the same way as descirbed above.
NOTE: The pf_ring.ko is loaded having the transparent_mode=0 by default which enables NAPI polling. If you use a card with special driver support for DNA you may want to compile the driver and load pf_ring.ko in a different mode.[8]

Load on boot

Since this seems to be difficult for many users the load procedure is described in the following.
Depending on your distribution or to be more specific, the init system your distribution uses at boot time may be somewhere different. In systemd [9] create a file with a .conf ending at /etc/modules-load.d/ which contains just the text `pf_ring`,

[4]See https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt for more information
[5]See http://www.ntop.org/products/pf_ring/
[6]Presently when composing this document there is no patch for the depreciation of create_proc_read_entry() function. See: https://lkml.org/lkml/2013/4/11/215
[7]documentation: http://www.ntop.org/products/pf_ring/DNA/
[8]See: man modprobe.d
[9]More info: http://www.freedesktop.org/wiki/Software/systemd/

the module name without the .ko ending.[10]
Ubuntu uses /etc/modules as a single file where you can add a line with the module name.[11]

```
systemd
echo pf_ring > /etc/modules-load.d/pfring.conf
OR
ubuntu
echo pf_ring >> /etc/modules
```

Figure 3: *on-boot kernel module load examples*

New kernel

Once in a while there is indeed a new kernel available. If you want to use pf_ring afterwards do not forget to recompile the kernel module, or set up `dkms`.

1.1.5 Supporting Scripts

The directory tranalyzer2/trunk/scripts contains useful scripts for post processing of tranalyzer output or compile support:

- convO3; Converts all plugins to optimization O3, default O2
- scnclean; Cleans a plugin directory of standard compiler files
- svncleanall; Cleans all compiler files under tranalyzer2/trunk
- svncleanrall; Cleans all files even your own plugins under tranalyzer/trunk
- fpsGplt; Transforms the packet signal output of nFirstPacketsStats plugin in flow file to gplot format
- protStat; Sorts proto file of protocolStatistics plugin according to biggest talkers
- statGplt; Transforms packet length , IAT statistics of packetSizeInterArrivalTimeHisto in flow file to gplot forma
- netwadd; Extracts from a standard flow file the ip Adresses and invokes whois requests (see warning in script)
- netw_1; Transforms list of ip adresses in cidr representation to whois answer record: for subnet file

[10]For more info: man modules-load.d
[11]See: man modules

2 Using Tranalyzer2 in a productive environment

Tranalyzer2 is designed in a modular way. Thus, the packet flow aggregation and the flow statistics are separated. While the main program performs the header dissection and flow organisation, the plugins produce specialized output such as packet statistics, mathematical transformations, signal analysis and result file generation.

2.1 Enabling/Disabling Plugins

If Tranalyzer2 is installed under */usr/local/bin*, it is globally executable by all users. The plugins, however, are stored in the user-dependent home folder and therefore individually tailored by the user. Upon program start, Tranalyzer2 checks the directory *.tranalyzer/plugins* and invokes every existing plugin. In order to integrate a new plugin it only has to be moved to the plugin folder. Removal from the folder deactivates a plugin[12].

2.2 man page

You either use the *install* option when building Tranalyzer or `gzip ./tranalyzer.1` under the Tranalyzer man directory, e.g., */tranalyzer2/trunk/tranalyzer2/man* and store the `.gz` file under */usr/share/man/man1*. Then the man page will be globally accessible. You also can invoke `man ./tranalyzer.1` in the Tranalyzer man directory.

2.3 Invoking Tranalyzer

As being stated earlier Tranalyzer2 either operates on Ethernet/DAG interfaces or pcap files. It may be invoked using a BPF if only certain flows are interesting. The required arguments are listed below. Note that the `-i`, `-r` and `-R` options cannot be used at the same time.

2.3.1 -i INTERFACE

Capture data from an Ethernet interface *INTERFACE*. *root* privileges are required otherwise Tranalyzer will exit.

2.3.2 -r FILE

Capture data from a pcap file *FILE*.

2.3.3 -R FILE

Process the list of pcap files in textfile: *FILE*. All files are being treated as one large file. The life time of a flow can extend over many files. The processing order is defined by the location of the filenames in the text file. The absolute path has to be specified. A comment line is defined by a '#' character.

2.3.4 -w PREFIX

Use a *PREFIX* for all output file types. The number of files being produced vary with the number of activated plugins. The file suffixes are defined in the file `tranalyzer.h` (see Section 2.6.11) or in the header files for the plugins. If you forget to specify an output file, Tranalyzer will use the input interface name or the file name as file prefix and print the flows to stdout. Thus, tranalyzer output can be piped into other command line tools, such as netcat in order to produce centralized logging to another host or an AWK script for further post processing without intermediate writing to a slow disk storage.

2.3.5 -p FOLDER

Changes the plugin folder from standard *.tranalyzer/plugins* to *FOLDER*.

[12]Hint: A new folder inside the plugins directory is suitable to store unused plugins.

2.3.6 -s [BPF Filter]

Initiates the packet mode, where a file with the suffix *_packets* is created when the **tcpFlags** plugin is loaded. This mode allows a deeper inspection of a specific flow in practise without using wire/t-shark or tcpdump. Thus a BPF filter extracting the flow of interest is preferable. Generally a BPF filter is applied to extract the flow of interest from the flow file. Each packet info is preceded by a Unix time stamp and the four tuple followed by the IPID, sequence/acknowledgement numbers, packet length, flag info, window size and TCP options. The output format is formated in decimal or hex notation for efficient application of Unix command line operators during the post-processing step. The timestamp is controlled by the `PKTFILE_DATE_TIME` variable. If set to the default value 1, a human readable Date Time format is selected namely for visual inspection. The standard Unix timestamp 0 is appropriate if extensive script postprocessing is necessary. A tab separated header description line is printed only at the beginning of the packet file. The first two lines then read as follows:

```
Time    pktInterdis    Flow_Index    EtherSrc_EtherDst    srcIP    srcPort  dstIP    dstport  proto    ipTOS    ipID    ipIDDiff    ipFrag  ipTTL
        ipHdrChkSum    ipCalChkSum    L4HdrChkSum    L4CalChkSum    ipFlags  pktLen   ipOptLen    ipOpts   seq    ack    seqDiff  ackDiff  SeqPktLen
        ackPktLen    tcpFlags    SpecialFlags    tcpWin   tcpOptLen    tcpOpts
14:24:49.751950  0.000000    1    00:13:D4:F3:AB:DD_00:13:D4:F3:A9:53    192.168.201.243  56151    192.168.201.242  51523    6    0x00    0x2E65   0    0
        x4000   64    0xF71F   0xF71F   0x8FAD   0x0000   0x1840   0    0    0xB7EBA608    0x00000000    0    0    0    0    0x02    0
        x0000   5840    20    0x02  0x04  0x05  0xB4  0x04  0x02  0x08  0x0A  0x01  0x53  0x88  0x6D  0x00  0x00  0x51  0x00  0x00  0x00  0x00  0x00
```

2.3.7 -e PCAPFILE

Denotes the filename and path of the esomfile when the pcapd plugin is loaded. The path and name of the pcapfile depends on PCAPFILE. If omitted the default names for the PCAP file are defined in **pcapd.h**.

2.3.8 BPF Filter

A Berkeley Packet Filter (BPF) can be specified at any time in order to reduce the amount of flows being produced and to increase speed during life capture ops. All rules of pcap BPF apply.

2.4 Description of networkHeaders.h

All header definitions are residing here. An important switch is also located here because every plugin and core program has to import it. The switch `IPV6_ACTIVATE == 1` or `0` controls the mode of operation on IP data. Either IPv6 or IPv4. Later versions will have a dual mode.

2.5 Description of packetCapture.h

The config file `packetCapture.h` provides control about the packet capture and packet structure process of Tranalyzer2. The most important fields are described below. Please note that after changing any value in define statements a rebuild is required.

2.5.1 PACKETLENGTH

The `PACKETLENGTH` variable controls the packet length processing of all plugins. Valid values are described in the table below:

Value	Meaning
0	Packet length including L2, L3 and L4 header
1	Packet length including L3 and L4 header
2	Packet length including L4 header
3	Packet length == L7 payload **(default)**

2.5.2 FRGIPPKTLENVIEW

If fragmentation is present and `FRGIPPKTLENVIEW` is set to 1 the L3 headers of all packets following the first packet are subtracted from the packet length. Hence, the total packet length of the whole fragmented packet stream is equal to the reassembled packet length. See table below:

Value	Meaning
0	IP header stays with 2nd++ fragmented packets
1	IP header stripped from 2nd++ fragmented packet

2.5.3 NOLAYER2

`NOLAYER2` controls the automatic protocol dissection engine. If set to 1, manual mode is active and the user has to position the pointer for the L3 header start by setting the `NOL2_L3HDROFFSET` switch to the appropriate offset value, e.g., 14. This is useful if artificial traffic produced by tools with invalid header has to be analyzed. The possible values are described in the table below:

Value	Meaning
0	artificial traffic: 0: Automatic L3 header discovery on , 1: Manual L3 header positioning on
1	artificial traffic: if (NOLAYER2 == 1) offset of L3 header

2.5.4 MAXHDRCNT

`MAXHDRCNT` defines the maximal headers in IPv6 to be dissected. A later version will also have a value for IPv4. Default is 5 but the minimal value has to be 3.

2.6 Description of tranalyzer.h

The config file `tranalyzer.h` provides abundant control over Tranalyzer2's functionality. The most important fields are described below. Please note that after changing any value in define statements a rebuild is required.

2.6.1 DEBUG

Activates debug output at console level for development purposes.

2.6.2 VERBOSE

Enables the verbose level if not equal to zero. Each level provides more elaborate information.

2.6.3 ENABLE_IO_BUFFERING

ENABLE_IO_BUFFERING = 1 activates an internal core plugin, a threaded buffer of size IO_BUFFER_SIZE . It buffers the peak when capturing data from the interface. Default is 0. IO_BUFFER_QUEUE_FULL_WAIT_MICROSEC handles the backpressure mechanism. More information s. section Plugins.

2.6.4 PACKETS_PER_BURST

Legacy functionality of Tranalyzer2 specifying the number of packets being processed by the *pcap_dispatch* routine before the flow termination check is initiated. For a detailed explanation see the pcap man page. A value greater than 1 prevents the program from immediately checking for timed-out flows after each incoming packet. This is not a problem, as long as there is no plugin activated that defines dependencies between flows. Nevertheless, this parameter should remain at 1 and will be eventually removed.

2.6.5 FORCE_MODE

A 1 enables the force mode which enables any plugin to force the output of flows independent of the timeout value. Hence, cisco netflow similar periodic output can be produced or overflows of counters can produce a flow and restart a new one.

2.6.6 ALARM_MODE

A 1 enables the alarm mode which differs from the default flow mode by the plugin based control of the tranalyzer core flow output. It is useful for classification plugins generating alarms, thus emulating alarm based SW such as Snort, etc. The default value is 0. The plugin sets the global output supress variable supOut = 1 in the onFlowTermination() function before any output is generated. The core resets supOut so that if the classifier standard flow mode is established again. This mode also allows multiple classification plugins producing an ored operation. A sampl code at the beginning of the onFlowTermination() function is shown below:

```
#if ALARM\_MODE == 1
        if (!Alarm) {
                supOut = 1;
                return;
        }
#endif // ALARM\_MODE == 1
```

Figure 4: *A sample code in the onFlowTermination routine*

2.6.7 L2TP

A 1 activates the L2TP processing of the Tranalyzer2 core. All L2TP headers either encapsulated in MPLS or not will be processed and followed down via PPP headers to the IP header and then passed to the IP processing. The default value of the variable is 0. Then the stack will be parsed until the first IP header is detected. So all L2TP UDP headers having src and dest port 1701 will be processed as normal UDP packets.

2.6.8 GRE

A 1 activates the L3 General Routing Encapsulation (L4proto=47) processing of the Tranalyzer2 core. All GRE headers either encapsulated in MPLS or not will be processed and followed down via PPP headers to the IP header and then passed to the IP processing. The default value of the variable is 0. Then the stack will be parsed until the first IP header is detected. If the following content is not existing or compressed the flow will contain only L4proto = 47 information.

2.6.9 FRAGMENTATION

A 1 activates the fragmentation processing of the Tranalyzer2 core. All packets following the header packet will be assembled in the same flow. The core and the plugin tcpFLags will provide special flags for fragmentation anomalies. Please refer to tcpFlags plugin documentation below. If `FRAGMENTATION` is set to 0 only the initial fragment will be processed; all later fragments will be ignored.

2.6.10 FRAG_HLST_CRFT

A 1 enables crafted packet processing even when the lead fragment is missing or packets contain senseless flags as being used in attacks or equipment failure. The default value is 0.

2.6.11 FRAG_ERROR_DUMP

A 1 activates the dump of packet information on the command line for time based identification of ill-fated or crafted fragments in tcpdump or wireshark. It provides the Unix timestamp, the six tuple, IPID and fragID as outlined in figure below. The default value is 0.

WARNING: If `FRAG_HLST_CRFT == 1` then every fragmented headerless packet will be reported!

```
1.frag not found @ 1291753225.639627 20 86.51.18.243 17664 92.105.108.208 54 17
   - 0x0DAE 0x00AC
1.frag not found @ 1291753225.655378 20 92.104.181.154 17664 93.144.66.3 150 17
   - 0x1941 0x00A0
1.frag not found @ 1291753225.825724 20 86.51.18.243 17664 92.105.108.208 54 17
   - 0x0DC1 0x00AC
1.frag not found @ 1291753225.850076 20 86.51.18.243 17664 92.105.108.208 54 17
   - 0x0DC2 0x00AC
```

Figure 5: *A sample report on stdout for packets with an elusive first fragment*

2.6.12 *_SUFFIX

These fields define the suffix of all plugin output files. For example if you specify the output *foo.foo* (with the `-w` option), the generated file for the per-packet output will be in the default setting *foo.foo_packets*.

2.6.13 FLOW_TIMEOUT

This field specifies the default time in seconds after which a flow will be considered as terminated since the last packet is captured. Note: Plugins are able to change the timeout values of a flow. For example the tcpStates plugin adjusts the timeout of a flow according to the TCP state machine.

2.6.14 HASHTABLE_SIZE

The number of buckets in the hash table. As a *separate chaining* hashing method is used, this value does not denote the amount of elements the hash table is able to manage! The current default value is 240,000.

2.6.15 HASHCHAINTABLE_SIZE

Specifies the amount of flows the main hash table is able to manage. Detailed explanation of the hashing function is available under `http://en.wikipedia.org/wiki/Hash_tables#Separate_chaining`. The default value is 120,000.

2.6.16 L2PROTO

Specifies the OSI layer 2 protocol of the network to be analyzed. The available protocols are listed below:

L2_ETHERNET	Ethernet (Default)

2.6.17 Aggregation Mode

The aggregation mode enables the user to confine certain IP, port or protocol ranges into a single flow. The variable `AGGREGATIONFLAG` in *tranalyzer.h* defines a bit field which enables specific aggregation modes according to the six tuple values listed below.

Aggregation Flag: 8 Bit switch	Hex Value
VLANID	0x20
SRCIP	0x10
DSTIP	0x08
SRCPORT	0x04
DSTPORT	0x02
L4PROT	0x01

if a certain aggregation mode is enabled the following variables in *tranalyzer.h* define the aggregation range.

Aggregation Flag: 8 Bit	Var type	Meaning
SRCIPMASKn	uint32_t	src IP aggregtion hex bit mask network order
DSTIPMASKn	uint32_t	dst IP aggregation hex bit mask network order
SRCPORTLW	uint16_t	src port lower bound
SRCPORTHW	uint16_t	src port upper bound
DSTPORTLW	uint16_t	dst port lower bound
DSTPORTHW	uint16_t	dst port upper bound

2.7 Tranalyzer2 output

As stated before, the functionality and output of Tranalyzer2 is defined by the activated plugins. Basically, there are two ways a plugin can generate output. First, it can generate its own output file and write any arbitrary content into any stream. The plugin is notified by an event as being described in Section 3.
The second way is called standard output or per-flow output. After flow termination Tranalyzer2 provides an output buffer and appends the direction of the flow to it. For example, in case of textual output, an “A” flow is normally followed by a “B” flow or if the “B” flow does not exist it is followed by the next “A” flow. Then, the output buffer is passed to the plugins providing their per-flow output. Finally the buffer is sent to the activated output plugins. This process repeats itself for the “B” flow. For detailed explanation about the functionality of the output plugins refer to the section plugins.

2.7.1 Hierarchical ordering of numerical or text output

Tranalyzer2 provides a hierarchical ordering of each output. Each plugin controls the:

- volume of its output
- number of values or bins
- hierarchical ordering of the data
- repetition of data substructures

Thus, complex structures such as lists or matrices can be presented in a single line.
The following sample of text output shows the hierarchical ordering for four data outputs, separated by tabulators:

```
A:	0.3	2.0_3.4_2.1	2;4;2;1	(1_2_9)_(1_3_1)_(7_5_3)_(2_3_7)
```

The `A:` indicates the direction of the flow; in this case it is the initial flow. The next number denotes a singular descriptive statistical result. Output number two consists of three values separated by “_” characters. Output number three consists of one value, that can be repeated, indicated by the character “;”. Output number four is a more complex example: It consists of four values containing three subvalues indicated by the braces. This could be interpreted as a matrix of size 4x3.

2.8 Final Report

Standard configuration of Tranalyzer2 produces a statistical report to stdout about timing, packets, protocol encapsulation type, average bandwidth, dump length etc. A sample report including some current protocol relevant warnings is depicted in the figure below. Warnings are not fatal hence are listed at the end of the statistical report when Tranalyzer2 terminates naturally. The *Average total Bandwidth* estimation refers to the processed bandwidth during the data acquisition process. It is only equivalent to the actual bandwidth if the total packet length including all encapsulations is not truncated and all traffic is IP. The *Average IP Traffic Bandwidth* is an estimate comprising all IP traffic actually present on the wire.

```
================================================================================
Tranalyzer 0.5.8 (Anteater), beta. PID: 13199
================================================================================

Preparing sniffing process...
Active plugins:
  00: protocolStatistics, version 0.5.8
  01: basicFlowOutput, version 0.5.8
  02: macRecorder, version 0.5.8
  03: basicLayer4CalcStatistics, version 0.5.8
  04: tcpFlags, version 0.5.8
  05: tcpStates, version 0.5.8
  06: icmpDecode, version 0.5.8
  07: connectionCounter, version 0.5.8
  08: textFileSink, version 0.5.8
Finished preparing sniffing.
Start processing file.
Dump start: 1291719781.988476 sec : Tue 07 Dec 2010 12:03:01.988476
Warning: Snaplength: 1550 - SnapIpLength: 1484 - IpLength in Header: 1492
ERROR: Hashtable full! Increase size of HASHTABLE_BASE_SIZE &|
    HASHCHAINTABLE_BASE_SIZE in tranalyzer.h.
Shutting down Tranalyzer 0.5.8...
Dump stop: 1291720134.902460 sec : Tue 07 Dec 2010 12:08:54.902460
Total dump duration: 352.913984 sec
Number of processed packets: 18816351
Number of processed traffic bytes: 15037860212
Number of IPv4 fragmented packets: 34977
Number of IPv6 fragmented packets: 0
Number of IPv4 packets: 18815132
Number of IPv6 packets: 0
Number of IPv4 flows: 610883
Average total Bandwidth: 333119.913 KBit/s
Average IP Traffic Bandwidth: 713131.423 KBit/s
Average full raw Bandwidth: 732531.261 KBit/s
Warning: Reduced snap length
Warning: IPv4 Fragmentation header packet missing, trailing packets ignored
Warning: VLAN encapsulation
Warning: MPLS encapsulation
Warning: L2TP encapsulation
```

Figure 6: *A sample Tranalyzer2 final report including encapsulation warning and Hash table full error*

Fatal errors regarding the invocation, configuration and operation of Tranalyzer2 are printed to stdout after the plugins are loaded, thus before the processing is activated, see the *Hash table error* example in figure ? above. These errors terminate Tranalyzer2 immediately and are located before the final statistical report as being indicated by the *"Shutting down ..."* key phrase. If the final report is to be used in a following script a pipe can be appended and certain lines can be filtered using grep or awk.

2.9 Monitoring during runtime

If debugging is deactivated or the verbose level is zero (see 2.6.1 and 2.6.2) Tranalyzer2 prints no status information. Therefore an interrupt signal has been introduced to force intermediate status information to stdout. The appropriate Unix command is:

```
kill -USR1 PID
```

It sends the signal `USR1` to the process with the process ID *PID*. The ID of the current Tranalyzer2 process is printed on the console at startup. Tranalyzer2 then supplies the following status information:

- Total number of processed packets
- Total number of processed flows
- Total number of bytes processed

If Tranalyzer2 is sniffing from an interface it also supplies the following additional information:

- Total number of received packets
- Total number of dropped packets by kernel and interface

If Tranalyzer2 processes a pcap file the following additional information is supplied:

- Total number of bytes to process
- Percentage completed

A verbose level not equal to zero enables the output of the number of free chains in the main hash map. An example of a typical signal requested report is shown below:

```
================================================================================
Showing progress for Tranalyzer 0.5.8 (Anteater), beta. PID: 27903
Current processed packets: 1197777
Current processed flows: 50081
Total bytes to process: 1660496154
Total bytes processed so far: 1026276888
Percentage completed: 61.81
Warning: Reduced snap length
Warning: IPv4 Fragmentation
Warning: IPv4 Fragmentation head packet missing, trailing packets ignored
================================================================================
```

Figure 7: *A sample Tranalyzer2 report after a kill signal*

2.10 Cancellation of the sniffing process

Processing of a pcap file stops upon end of file. In case of life capture from an interface Tranalyzer2 stops upon `CTRL+C` interrupt or a `kill -9 PID` signal. The disconnection of the interface cable will stop Tranalyzer2 also after a timeout of 182 seconds. The console based `CTRL+C` interrupt does not immediately terminate the program to avoid corrupted entries in the output files. It stops creating additional flows and finishes only currently active flows. Note that waiting the termination of active flow depends on the activity or the lifetime of a connection and can take a very long time. In order to mitigate that problem the user can issue the `CTRL+C` for `PROCESS_SIGKILL_LEVEL_THRESHOLD` times to immediately terminate the program.

3 Plugins

This section gives a brief overview about every plugin being supplied. First both output plugins, namely standard file sink and text file sink, are introduced and then the statistical plugins. Plugins marked with * are only build if the *autogen_all.sh* script is executed (see 1.1.2). If a plugin provides options, they are defined in their specific header file. Note that after changing an option the plugin needs to be recompiled with the custom autogen script.

3.1 Standard File Sink

3.1.1 Description

The standard file sink plugin is the basic output plugin for Tranalyzer2. It uses the output prefix to generate the standard flow file with suffix `_flows`. All standard output from every plugin is stored in binary format in this file. For text conversion **tranalyzer-b2t** (see 3.2) or **Traviz**, a graphical inspector `http://traviz.sf.net`) has to be utilized. A compiled version of <em tranalyzer-b2t is placed in the plugins folder only if the Text File Sink plugin is compiled (see 3.2).

3.2 Text File Sink

3.2.1 Description

If only human readable text output is required the *Text File Sink Plugin* can be compiled instead of the Standard binary. The text file sink prints a textual representation of all plugins results into a file with suffix `_flows_txt`. Each line in the file represents one flow. The different output statistics of the plugins are separated by a tab character to provide better post-processing with command line scripts or statistical toolsets.

3.2.2 Options

TEXT_FILE_SINK_PRINT_HEADER If non-zero, the plugin enables the printing of a header row at the beginning of the output file. The row contains the short names of each output statistic separated by tabs.

TEXT_FILE_SINK_PRINT_HEADER_FILE If non-zero, the plugin generates a separate header file with suffix `_headers` s. fig below. Each line in the file contains a detailed description of the output statistics: A consecutive index number, the type and detailed description of the values.

3.3 binaryToText.h

`binarytoText.h` controls the conversion from internal binary format to standard text output. Historically Tranalyzer only supported text output in a row based format. The Header file describes the columns of the flow file and data type being used as indicated below for a case without special protocol output.

Due to the fact that most of the post analysis was conducted by the naked eye or with tools such as Bash AWK, Perl, SPSS, Excel or LibreOffice, IP addresses were displayed in a normalized format: `172.029.222.085` facilitating row and column based post processing. Nevertheless practice showed that for post processing with SQL host databases, IDS correlation, nslookup or whois the standard IP format `172.29.222.85` is preferable. In the *textFileSink* directory the headerfile *binaryToText.h* contains the variable `IP_PRINT_NORMALIZE` which controls the format of the IP address. It is set to 0 by default due to a final practitioner vote.
When interpreting hex numbers, capital letters proved to be beneficial during our practical work. Nevertheless, there are lingering questions by several parties whether small letter representation is more effective when searching and typing, etc. In order to abolish future discussions about that matter the variable `HEX_CAPITAL` was introduced. Default value is 1, thus capital representation.

3.3.1 tranalyzer-b2t

The plugin also supplies a program: **tranalyzer-b2t** (short for binary to text). It is a small console program that transforms binary Tranalyzer files into column oriented text files. The text file is the same as being printed by the Text File Sink plugin.

```
# Header file for flow file
# /home/hartwurstadapter/tranalyzer_output/skype_flows_txt
# Col No.       Type     Name
1       24:NR   Flow direction
2       25:NR   System time of first packet
3       25:NR   System time of last packet
4       26:NR   Flow duration
5       8:NR    VLAN ID
6       28:NR   Source IP address
7       8:NR    Subnet number of source IP
8       8:NR    Source port
9       28:NR   Destination IP address
10       8:NR    Subnet number of destination IP
11      8:NR    Destination port
12      7:NR    Layer 4 protocol
13      27,27,10:R      Source MAC address, destination MAC address, number of
    packets seen with this MAC address combination
14      23:NR   Port based classification of the destination port
15              ...
```

Figure 8: *A sample header file showing the columns of the plugins* Basic Flow Output, MAC Recorder *and* Port-based Classifier. *The first column shows the column number of the statistic, the second the type of values inside the statistic and if these values can be repeated (R) or not (NR). The third column contains the description of the column.*

The program is stored under the plugins folder. For more information about its operation invoke the program without any arguments.

3.4 Basic Flow Output

3.4.1 Description

The basic flow output plugin writes host identification fields and timing information into the standard flow output file. All compiler switches controlling the output of the *BasicFlowOutput* plugin are defined in the file `basicFlowOutput.h`. In detail, the following fields are supplied to the flow output file if all compiler switches are activated:

Column	Meaning
1	Flow: Flow Direction A: / B:
2	Flow: Flow Index (Optional)
3	Flow: Flow Status Bit Field (Optional)
4	Flow: Time stamp of packet being captured first [Date-time or unix time]
5	Flow: Time stamp of packet being captured last [Date-time or unix time]
6	Flow: Duration of the flow [unix time]
7	Flow: VLAN number [inner VLAN or all headers in HEX] (Optional)
8	Flow: MPLS header [HEX or Detail view] (Optional)
9	Flow: PPP header (Optional)
10	Flow: GRE header
11	Flow: GRE source IP address
12	Flow: GRE Src IP(4/6), several output modes controlled by textFileSink.h (Optional)
13	Flow: GRE Src IP Subnet Label [Decimal or Hex] (Optional)
14	Flow: GRE Dst IPv(4/6), several output modes controlled by textFileSink.h (Optional)
15	Flow: GRE Dst IP Subnet Label [Decimal or Hex] (Optional)
16	Flow: L2TP header (Optional)
17	Flow: L2TP TID (Optional)
18	Flow: L2TP SID (Optional)
19	Flow: L2TP Src IP(4/6), several output modes controlled by textFileSink.h (Optional)
20	Flow: L2TP Src IP Subnet Label [Decimal or Hex] (Optional)
21	Flow: L2TP Dst IPv(4/6), several output modes controlled by textFileSink.h (Optional)
22	Flow: L2TP Dst IP Subnet Label [Decimal or Hex] (Optional)
23	Flow: Src IPv(4/6), several output modes controlled by textFileSink.h
24	Flow: Src IP Subnet Label [Decimal or Hex] (Optional)
25	Flow: Src Port
26	Flow: Dst IPv(4/6), several output modes controlled by textFileSink.h
27	Flow: Dst IP Subnet Label [Decimal or Hex] (Optional)
28	Flow: Dst Port
29	Flow: Layer 4 protocol

All fields which do not exist in a flow will be displayed as a 0 in decimal or hex respectively. Note: subnet labeling is currently only available in IPv4 mode.

3.4.2 Time Stamps

Time stamps are controlled by the flag `BASICFLOWOUTPUT_DATE_TIME`. If set to 1 a human readable date time format is selected, being also the default value, because visual inspection of flow files seems to be the preferred method of our users. Nevertheless, I am using the standard Unix Timestamp mode = 0, because it is more appropriate for the script based postprocessing of large files.

3.4.3 Flow Index Output

If the flag `BASICFLOWOUTPUT_FLOWINDEX` is set to 1 a unique flow identification column is added. It is useful to identify flows when post processing operations, such as sort or filters are applied to a flow file and only a `B:` or an `A:` flow is selected. Moreover a packet file generated with the `-s` option supplies the flow index which simplifies the mapping of singular packets to the appropriate flow. The default value is 1.

3.4.4 Flow Status Output

If the flag `BASICFLOWOUTPUT_FLOW_STATUS` is set to 1 a 32 bit long status word is supplied in the 2nd column of each flow. The default value is 0. The bit position is outlined below:

Bitfield	Meaning
2^0 (=0x00000001)	Dump: Warning Flag: L2 Snaplength too short; Flow: Invert Flow, not client flow
2^1 (=0x00000002)	Dump/flow: L3 Snaplength too short
2^2 (=0x00000004)	Dump/flow: L2 header length too short
2^3 (=0x00000008)	Dump/flow: L3 header length too short
2^4 (=0x00000010)	Dump: Warning: IP Fragmentation Detected
2^5 (=0x00000020)	Flow: ERROR: Severe Fragmentation Error
2^6 (=0x00000040)	Flow: ERROR: Fragmentation Header Sequence Error
2^7 (=0x00000080)	Flow ERROR: Fragmentation Pending at end of flow
2^8 (=0x00000100)	Flow/Dump: Warning: PPPoE_D detected
2^9 (=0x00000200)	Flow/Dump: Warning: PPPoE_S detected
2^{10} (=0x00000400)	Flow/Dump: Warning: LLDP detected
2^{11} (=0x00000800)	Flow/Dump: Warning: ARP detected
2^{12} (=0x00001000)	Flow/Dump: Warning: reverse ARP detected
2^{13} (=0x00002000)	Flow/Dump: Warning: VLAN(s) detected
2^{14} (=0x00004000)	Flow/Dump: Warning: MPLS unicast detected
2^{15} (=0x00008000)	Flow/Dump: Warning: MPLS multicast detected
2^{16} (=0x00010000)	Flow/Dump: Warning: L2TP detected
2^{17} (=0x00020000)	Flow/Dump: Warning: GRE detected
2^{18} (=0x00040000)	Flow/Dump: Warning: PPP detected
2^{19} (=0x00080000)	Flow/Dump: 0/1: IPv4/IPv6 detected
2^{20} (=0x00100000)	Flow/Dump: -
2^{21} (=0x00200000)	Flow/Dump: -
2^{22} (=0x00400000)	Flow/Dump: -
2^{23} (=0x00800000)	Flow/Dump: -
2^{24} (=0x01000000)	Flow/Dump: -
2^{25} (=0x02000000)	Flow/Dump: -
2^{26} (=0x04000000)	Flow/Dump: -
2^{27} (=0x08000000)	Flow/Dump: -
2^{28} (=0x10000000)	Flow/Dump: PPPL3 header not readable, compressed
2^{29} (=0x20000000)	Flow/Dump: -
2^{30} (=0x40000000)	Flow/Dump: Warning: Land Attack detected
2^{31} (=0x80000000)	Flow/Dump: Warning: Time Jump

3.4.5 Subnet detection

If the flag `BASICFLOWOUTPUT_SUBNET_TEST` or `BASICFLOWOUTPUT_SUBNET_TEST_L2TP` is enabled source and destination IP of the IP packet or the encapsulating L2TP packet stack will be marked according to their membership to a defined set of subnets. This definition is supplied in a user defined file `subnet.txt` residing in the plugin folder. If no subnet file exists in the *.tranalyzer/plugin* folder a sample file will be automatically created. It can be edited where each line contains the subnet and network mask as being outlined below in case of decimal labeling:

```
#Subnet/msk      User defined member net
# Private Address Space 1-3
10.0.0.0/8       1
172.16.0.0/12    1
192.168.0.0/16   1
# extern 4 - 5
62.202.0.0/15    2
81.62.0.0/15     2
```

If the variable `NETWORKLABELLING` equals 0 in the `subnet.h` file, the subnet number in the flow file is defined by the position of the subnet in the subnet file starting with one. A zero value in the flow file denotes the fact that the IP does not match any subnet. In case of `NETWORKLABELLING = 1` the user defined membership numbers are assigned to the subnet number in the flow file. If the variable `NETWORK_HEXLABEL` equals 1 the user defined membership numbers are interpreted as a 32 bit hex number, thus a bit wise meaning can be assigned to address spaces, such as country, city, industry or degree of evil. The default was originally 0, but was changed to default 1 as we use it that way. If you are a decimal guy just switch it back to 0 and run `./autogen.sh` under the plugin directory.

3.4.6 Script netwadd

The script **netwadd** in the trunk/scripts folder extracts from a *flows_txt* file all networks in CIDR notation by invoking with the following parameters:

```
   ./netwadd path/filename_flow_txt > waurich
```

By applying the following command sequence a draft subnet file is created:

```
sort -k 2 waurich | uniq > subnet.txt
```

All networks are now sorted according to the network owner. This output can be used to build a customized subnet file with a user defined labeling.

3.4.7 Ether Type

If the flag `BASICFLOWOUTPUT_ETHERTYPE` is activated a column is added to the flow file describing all relevant Ether Type charactersitics of the L2 part of the packet, such as 8021Q VLAN encapsulation including VLAN nesting. The variable `BASICFLOWOUTPUT_ETHERTYPE_HEX` controls the output format. A 1 prints the plain hex 32 bit Ether Types, while 0 represents the default case, decimal VLAN Identifiers separated by semicolons if nesting is present.

3.4.8 MPLS

If the flag `BASICFLOWOUTPUT_MPLS` is set to 1, a column is added to the flow file describing the MPLS headers. The `BASICFLOWOUTPUT_MPLS_DETAIL` controls the output format. If set to 1 the MPLS headers will be decoded and displayed as decimals separated by underlines:
`Label1_ToS1_S1_TTL1`;textbfLabel2_ToS2_S2_TTL2;...
otherwise the raw 32 bit MPLS headers are printed, also separated by semicolons. Default is 0 for both compiler switches, as most of the common users do not use MPLS, unless Tranalyzer2 issues a warning in its final report. If you are working for an operator you might need it.

3.4.9 L2TP

If the flag `BASICFLOWOUTPUT_L2TP` is set to 1 a column is added to the flow file describing relevant L2TP and preceding IP header contents as being listed below:

L2TP Variable	**Description**
Header	Flags and version header bit field
Tunnel ID	Tunnel ID of 1. packet
Session ID	Session ID of 1. packet
Source IP address	Source Addr of preceding IP header
Destination IP address	Destination Addr of preceding IP header

Operator encapsuation. In normal inhouse traffic not needed, unless Tranalyzer2 issues a warning in its final report. Default is set to 0.

3.4.10 GRE

If the flag `BASICFLOWOUTPUT_GRE` is set to 1 a column is added to the flow file describing relevant GRE and preceding IP header contents as being described below:

GRE Variable	**Description**
GRE Header	32Bit, Flags, version and following protocol
Source IP address	Source Addr of preceding IP header
Destination IP address	Destination Addr of preceding IP header

Operator encapsulation. In normal in-house traffic not needed, unless Tranalyzer2 issues a warning in its final report. Default is set to 0.

3.4.11 PPP

If the switch `BASICFLOWOUTPUT_PPP` is set to 1 a PPP header column is added to the flow file. Default is 0. If you are working for an operator it comes in handy.

3.5 Basic Layer 4 Statistics

3.5.1 Description

It supplies basic layer four statistics for each flow. The following fields are written to the `PREFIX_flows` file:

Column	Meaning
1	Flow: Number of packets being transmitted
2	Flow: Number of packets being received
3	Flow: Number of bytes being transmitted
4	Flow: Number of bytes being received
5	Flow: Minimum packet length (optional)
6	Flow: Maximum packet length (optional)
7	Flow: Average packet size (optional)
8	Flow: Packets per second (optional)
9	Flow: Bytes per second (optional)
10	Flow: Packet Asymmetry (optional)
11	Flow: Byte Asymmetry (optional)

The variable **BASIC_PKT_STATS** in *basicLayer4Statistics.h* controls the calculation and output of the basic statistics features 5 - 11. If set to 0 a rapid assessments of dumps is possible without statistics info. The default is set to 1.

3.6 TCP Flags

3.6.1 Description

This plugin contains IP and TCP Header information encountered during the lifetime of a flow. All features are a result of practical troubleshooting experience in the field. Features can be selected when setting the following key values in *tcpFlags.h*:

Variable	Default	Description
`SPKTFILE_KEY_VALUE`	0	-s option: 0: description in 1. line of flow file, 1: Key value pairs
`RTT_ESTIMATE`	1	1: Round trip time estimation
`IPCHECKSUM`	2	1: Calculation of L3 (IP) Header Checksum,
		2: L3/L4 (TCP,UDP,ICMP,IGMP, ...) Checksum
`WINDOWSIZE`	0	1: Calculation of tcp window size parameters
`SEQ_ACK_NUM`	0:	1: Sequence/Acknowledge Number features
`FRAG_ANALYZE`	1	1: Fragmentation analysis enabled, 0: Fragmentation analysis off

Using the default setting above the following columns are provided below. The round trip measurements are stored in their respective flow direction `A:` or `B:`, also in order to save memory. If all swiches are engaged then the following columns are added to the flow file.

Column	Meaning
1	IP: IPv4: Minimum Delta IPID
2	IP: IPv4: Maximum Delta IPID
3	IP: Minimum TTL
4	IP: Maximum TTL
5	IP: TTL Min/Max Change Count
6	IP IPv4:TOS Byte, IPv6: Traffic Class
7	IP: Header flags
8	IP: IPv4/6: 8 and 32 Bit Options Field [$2^{\text{Copy-Class}}$]_[2^{Number}]
9	IP: Options count
10	TCP: Packet Sequence Count
12	TCP: Sent Bytes, Agg. Difference of Sequence Numbers
13	TCP: Sequence Number retry count
14	TCP: Packet Ack Count
15	TCP: Sent Bytes, Aggregated difference of Ack Numbers
16	TCP: Ack Number retry count
17	TCP: Initial tcp effective window size
18	TCP: Average effective window size
19	TCP: Minimum effective window size
20	TCP: Maximum effective window size
21	TCP: Effective window size change down count
22	TCP: Effective window size change up count
23	TCP: Effective window size directional change count
24	TCP: Aggregated TCP protocol flags
25	TCP: Aggregated header anomaly flags
26	TCP: 32 Bit Options Field [2^{Type}], s.
27	TCP: Maximum Segment Size
28	TCP: Windows Scale
29	TCP: Options count
30	TCP: Packet Initial TCP Trip Time A: Syn, Syn-Ack | B: Syn-Ack, Ack
31	TCP: Packet Initial TCP Round Trip Time A: Syn, Syn-Ack, Ack | B: RTT Average|
32	TCP: Packet TCP Ack Trip Minimum A: | B:
33	TCP: Packet TCP Ack Trip Maximum A: | B:
34	TCP: Packet TCP Ack Trip Average A: | B:

3.6.2 Window Size Features

The window size features are experimental. Practice showed that some of them were useful as anomaly detection for several congestion problems especially where a host is the cause. The effective window size is calculated as follows:
$\text{Windowsize} \cdot 2^{\text{WS}}$
The average window size is calculated by a first order IIR butterworth filter giving an indication about the potential throughput of a connection. Minimum and maximum window size in combination with average window size indicate also potential throughput problems. In combination with the Window Size up down counts type the quality of the regulation algorithm and problems on the host side can be assessed. The directional change count detects erratic behavior of a channel. More experiments and practical troubleshooting cases will indeed reveal an improvement of the listed features. Proposals are more than welcome, please submit to the Sourceforge Feature Requests mailing list: http://sourceforge.net/projects/tranalyzer/support.

3.6.3 Aggregated IP header flags

The meaning of the aggregated IP header flags bitfield is the following:

Bitfield	Meaning
2^0 (=0x0001)	IP Options present, s. IP Options Type Bit field
2^1 (=0x0002)	IPID out of order
2^2 (=0x0004)	IPID rollover
2^3 (=0x0008)	Fragmentation: Below expected RFC minimum fragment size: 576
2^4 (=0x0010)	Fragmentation: Warning fragments out of range (Possible tear drop attack)
2^5 (=0x0020)	Fragmentation: MF Flag
2^6 (=0x0040)	Fragmentation: DF Flag
2^7 (=0x0080)	Fragmentation: x Reserved flag bit from IP Header
2^8 (=0x0100)	Fragmentation: Unexpected position of fragment
2^9 (=0x0200)	Fragmentation: Unexpected sequence of fragment
2^{10} (=0x0400)	L3 Checksum Error
2^{11} (=0x0800)	L4 Checksum Error
2^{12} (=0x1000)	SnapLength Warning: IP Packet truncated, L4 Checksums invalid
2^{13} (=0x2000)	Packet Interdistance == 0
2^{14} (=0x4000)	Packet Interdistance < 0
2^{15} (=0x8000)	State Bit for interdistance assessment

3.6.4 IP Options Type Bit Field

The aggregated IP options are coded as a bit field in hexadecimal notation where the bit position denotes the IP options type according to following format: [$2^{\text{Copy-Class}}$]_[2^{Number}]. If the field reads: `0x10_0x00100000` in an ICMP message it is a 0x94 = 148 router alert.
Refer to RFC for decoding the bit field : http://www.iana.org/assignments/ip-parameters.

3.6.5 Aggregated TCP protocol flags

The meaning of the aggregated TCP protocol flags bitfield is the following:

Bitfield	Flag	Meaning
2^0 (=0x01)	FIN	No more data, finish connection
2^1 (=0x02)	SYN	Synchronize sequence numbers
2^2 (=0x04)	RST	Reset connection
2^3 (=0x08)	PSH	Push data
2^4 (=0x10)	ACK	Acknowledgement field value valid
2^5 (=0x20)	URG	Urgent pointer valid
2^6 (=0x40)	ECE	ECN-Echo
2^7 (=0x80)	CWR	Congestion Window Reduced flag is set

3.6.6 Example

If the TCP Flags column contains the value `0x1B` all flags except the urgent- and the push flag where recognized in the flow.

3.7 Aggregated TCP Anomaly Flags

Description: The TCP Flags Plugin supplies a 16 Bit word containing information about anomalous Layer 3/4 behavior of a TCP flow. It proved to be useful in practical network security or troubleshooting cases of large IT infrastructures. In combination with the TCP Flag the correct initiation or completion of a TCP Handshake is also retracable. In combination with packets and bytes transmitted or received key values routing problems can easily be detected.

Bitfield	Meaning
2^0 (=0x0001)	Fin-Ack Flag
2^1 (=0x0002)	Syn-Ack Flag
2^2 (=0x0004)	Rst-Ack Flag
2^3 (=0x0008)	Syn-Fin Flag, Scan or malicious packet
2^4 (=0x0010)	Syn-Fin-Rst Flag, potential malicious scan packet or malicious channel
2^4 (=0x0020)	Fin-Rst Flag, abnormal flow termination
2^5 (=0x0040)	Null Flag, potential NULL scan packet, or malicious channel
2^6 (=0x0080)	XMas Flag, potential Xmas scan packet, or malicious channel
2^8 (=0x0100)	Due to packet loss, Sequence Number Retry, retransmit
2^9 (=0x0200)	Sequence Number out of order
2^{10} (=0x0400)	Sequence mess in flow order due to pcap pkt loss
2^{11} (=0x0800)	Warning: L4 Option field corrupt or not acquired
2^{12} (=0x1000)	Syn retransmission
2^{13} (=0x2000)	Ack number out of order
2^{14} (=0x4000)	Ack Packet loss, probably on the sniffing interface
2^{15} (=0x8000)	Last state of TCP Window State Machine

3.7.1 TCP Options Type Bit Field

The aggregated TCP Options are coded as a bit field in a 32 bit hexadecimal notation where the bit position denotes the TCP options type according to following format: [$2^{\text{TCP Options Type}}$]. So if option type 0, 2 and 8 is encountered the resulting value will read `0x00000105`. Refer to RFC for decoding the bit field: `http://www.iana.org/assignments/tcp-parameters/tcp-parameters.xml`.

3.7.2 Example

A prominent example is the routing problem by misconfiguration: Anomaly flag shows `0xXX03` with Flags `0x1A` indicating perfect data exchange but the received byte count and packet count are zero. Either the return traffic is not captured and/or a routing anomaly exists, such as the traffic returns via an unknown gateway. This was an actual case resolving a firewall misconfiguration combined with unexpected OSPF actions in a large company network.

3.8 TCP States Analyzer

3.8.1 Description

This plugin tracks the actual state of a TCP connection. To do so, it analyzes the flags being set in the packet header. If the plugin recognizes non-compliant behavior, it sets a bit in a so-called *bogus bitfield*. The definition of the bogus bit field is defined below:

Bitfield	Meaning
2^0 (=0x01)	Malformed connection establishment
2^1 (=0x02)	Malformed teardown
2^2 (=0x04)	Malformed flags during established connection
2^3 (=0x08)	Packets detected after teardown
2^4 (=0x10)	NA
2^5 (=0x20)	NA
2^6 (=0x40)	Reset from sender
2^7 (=0x80)	Potential evil behavior

The TCP States Analyzer plugin also changes the timeout values of a flow according to its recognized state:

State	Meaning	Timeout in seconds
New	Three way handshake is encountered	120
Established	Connection established	610
Closing	Hosts are about to close the connection	120
Closed	Connection closed	10
Reset	Connection reset encountered by one of hosts	0.1

3.8.2 Differences to the host TCP state machines

The plugin state machine and the state machines usually implemented in hosts differ in some cases. Major differences are caused by the benevolence of the plugin. For example, if a connection has not been established in a correct way, the plugin treats the connection as established, but sets the *malformed connection establishment* flag. The reasons for this benevolence are the following:

- A flow might have been started before invocation of Tranalyzer2.
- A flow did not finish before Tranalyzer2 terminated.
- Tranalyzer2 did not detect every packet of a connection, for example due to a router misconfiguration.
- Flows from malicious programs may show suspicious behavior.
- Packets may be lost **after** being captured by Tranalyzer2 but **before** they reached the opposite host.

3.9 ICMP Decoder

3.9.1 Description

It analyzes ICMP traffic and provides absolute and relative statistics to the `PREFIX_icmpStats` file. In detail, as an example for IPv4 ICMP traffic, fields with the following meaning will be supplied:

Type	Subtype	Meaning
ICMP_ECHOREQUEST	-	Echo request
ICMP_ECHOREPLY	-	Echo reply to an echo request
ICMP_SOURCE_QUENCH	-	Source quenches
ICMP_TRACEROUTE	-	Host trace route
ICMP_DEST_UNREACH	ICMP_NET_UNREACH	Network unreachable
ICMP_DEST_UNREACH	ICMP_HOST_UNREACH	Host unreachable
ICMP_DEST_UNREACH	ICMP_PROT_UNREACH	Protocol unreachable
ICMP_DEST_UNREACH	ICMP_PORT_UNREACH	Port unreachable
ICMP_DEST_UNREACH	ICMP_FRAG_NEEDED	Fragmentation needed
ICMP_DEST_UNREACH	ICMP_SR_FAILED	Source route failed
ICMP_DEST_UNREACH	ICMP_NET_UNKNOWN	Network unknown
ICMP_DEST_UNREACH	ICMP_HOST_UNKNOWN	Host unknown
ICMP_DEST_UNREACH	ICMP_HOST_ISOLATED	Host is isolated
ICMP_DEST_UNREACH	ICMP_NET_ANO	Network annotation
ICMP_DEST_UNREACH	ICMP_HOST_ANO	Host annotation
ICMP_DEST_UNREACH	ICMP_NET_UNR_TOS	Unreachable type of network service
ICMP_DEST_UNREACH	ICMP_HOST_UNR_TOS	Unreachable type of host service
ICMP_DEST_UNREACH	ICMP_PKT_FILTERED	Dropped by a filtering device
ICMP_DEST_UNREACH	ICMP_PREC_VIOLATION	Precedence violation
ICMP_DEST_UNREACH	ICMP_PREC_CUTOFF	Precedence cut off
ICMP_REDIRECT	ICMP_REDIR_NET	Network redirection
ICMP_REDIRECT	ICMP_REDIR_HOST	Host redirection
ICMP_REDIRECT	ICMP_REDIR_NETTOS	Network type of service
ICMP_REDIRECT	ICMP_REDIR_HOSTTOS	Host type of service
ICMP_TRACEROUTE	-	Traceroute packets

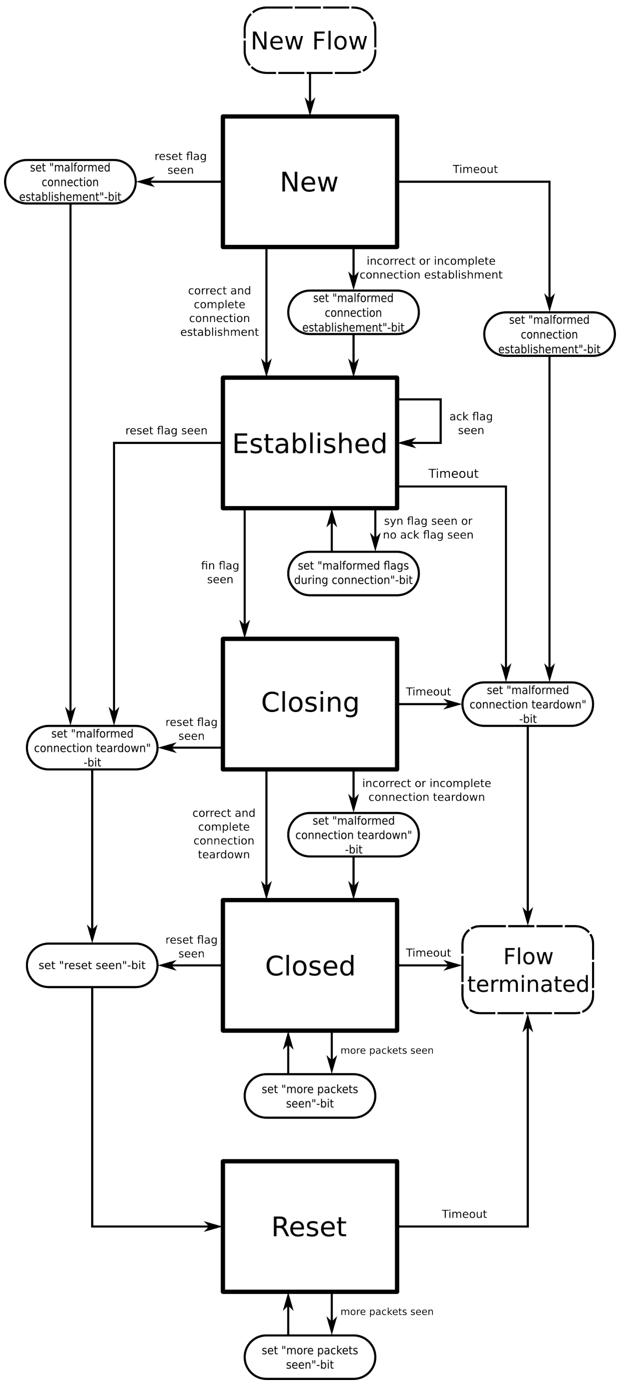

New Flow
New
reset flag seen
set "malformed connection establishement"-bit
Timeout
incorrect or incomplete connection establishment
correct and complete connection establishment
set "malformed connection establishement"-bit
set "malformed connection establishement"-bit
Established
ack flag seen
reset flag seen
Timeout
syn flag seen or no ack flag seen
fin flag seen
set "malformed flags during connection"-bit
Closing
Timeout
set "malformed connection teardown" -bit
reset flag seen
set "malformed connection teardown" -bit
incorrect or incomplete connection teardown
correct and complete connection teardown
set "malformed connection teardown" -bit
Closed
reset flag seen
set "reset seen"-bit
Timeout
Flow terminated
more packets seen
set "more packets seen"-bit
Reset
Timeout
more packets seen
set "more packets seen"-bit

Figure 9: *The state machine of the TCP State Analyzer plugin. For better readability the connection establishment and teardown recognition are substituted each by a summarizing arrow. Also the recognition of malicious behavior is not being shown.*

ICMPv6 provides a similar output when `IPV6_ACTIVATE == 1` is selected. For each flow two columns called *Aggregated ICMP Type & Code bit Field* and *icmpERSuccRate* are appended in the `PREFIX_flows` file output of this plugin. The *icmpERSuccRate* indicates the success of an ICMP echo reply process and is defined by the following values:

icmpERSuccRate	Meaning
Negative value	No ICMP traffic in flow
Positive value	Percentage of ICMP echo replies to echo requests
Zero (Null)	ICMP requests being captured but no replies detected

The aggregated ICMP Code and Type Bit field is supplied in 16 and 32 bit hexadecimal notation using the following format: $[2^{\text{ICMP Code}}]\_[2^{\text{ICMP Type}}]$. The bit position denotes the ICMP type or code information.

3.10 Connection Counter

Description: Records the number of connections at the termination of a flow:
The first value is the number of connections from the source IP to **different** destination hosts, the second denotes the same for the destination host. The third value describes the number of connections between source and destination host.

3.11 Descriptive Statistics *

3.11.1 Description

Calculates various statistics about a flow. If the packet length statistics output is enabled in the descriptiveStatistics.h file the following key values are supplied in the flow file:

Column	Statistics
01	Minimum packet length
02	Maximum packet length
03	Mean packet length
04	Lower quartile of packet lengths
05	Median of packet lengths
06	Upper quartile of packet lengths
07	Inter quartile distance (IQD) of packet lengths
08	Mode of packet lengths = The most occurring packet length
09	Standard deviation (stddev) of packet lengths
10	Robust standard deviation of packet lengths = minimum of stddev and 0.7413 * IQD
11	Skewness of packet lengths
12	Excess of packet lengths

If the inter-arrival time statistics output is enabled the following key values are supplied:

Column	Statistics
01	Minimum inter-arrival time
02	Maximum inter-arrival time
03	Mean inter-arrival time
04	Lower quartile of inter-arrival times
05	Median of inter-arrival times
06	Upper quartile of inter-arrival times
07	Inter quartile distance (IQD) of inter-arrival times
08	Mode of inter-arrival times = The most occurring inter-arrival time
09	Standard deviation (stddev) of inter-arrival times
10	Robust standard deviation of inter-arrival times = minimum of stddev and 0.7413 * IQD
11	Skewness of inter-arrival times
12	Excess of inter-arrival times

Because the inter-arrival time of the first packet is per definition always zero, it is removed from the statistics. Therefore the inter-arrival time statistics values for flows with only one packet is set to zero.

3.11.2 (De)Activation of flow based output

The output for the packet length and inter-arrival time statistics can be enabled and disabled via the switches `ENABLE_PACKETSIZE_CALCULATION` and `ENABLE_IAT_CALCULATION` in the *descriptiveStatistics.h* header file. If these values are changed, the user has to rebuild the plugin using the *autogen.sh* script in the root folder of the plugin.

3.11.3 Dependencies

The descriptive statistics plugin makes use of the structures in the packet length inter-arrival time histo plugin. Therefore, the packet length inter-arrival time histogram plugin has to be activated.

3.11.4 Known issues

Because the packet length and inter-arrival time plugin stores the inter-arrival times in statistical bins the original time information is lost. Therefore the calculation of the inter-arrival times statistics is due to its logarithmic binning only a rough approximation of the original timing information. Nevertheless, this representation has shown to be useful in practical cases of anomaly and application classification.

3.12 N First Packet Signal *

This plugin supplies the PL and IAT of the N first packets per flow as columns in the `PREFIX_flows` file. This representation has shown to be useful in fast and efficient application classification. Useful values of N are in the range of 8 to 30, the default value is 20. The output in the `PREFIX_protocols` file is the following:

```
[PL packet 1]_[IAT packet 1];[PL packet 2]_[IAT packet 2]; ...  ;[PL packet N]_[IAT packet N]
```

3.12.1 Gnuplot/Excel/SPSS support

By invoking the script `fpsGplt` under *trunk/scripts* files are generated for the packet signal in a Gnuplot/Excel/SPSS readable column oriented format. The format is shown below:

PL	IAT	absolute time

3.13 Packet Length and Inter-Arrival Time Histogram *

3.13.1 Description

This plugin records the PL and IAT of a flow. While the PL is precise the IAT is divided by default into 91 statistical bins:

Bin	Range of IAT
0 – 39	0 ms (incl.) – 200 ms (excl.), partitioned into bins of 5 ms
40 – 59	200 ms (incl.) – 400 ms (excl.), partitioned into bins of 10 ms
60 – 89	400 ms (incl.) – 1 sec. (excl.), partitioned into bins of 20 ms
90	for all IAT higher than 1 sec.

Classifying tasks may require other IAT binning. Then the bin limit `IATBINBu` and the binsize `IATBINWu` constants in *packetSizeInterArrivalTimeHisto.h* need to be adapted as being indicated below using 5 different classes of bins:

```
#define IATBINBu1     50          // bin boundary of section one: [0, 50)ms
#define IATBINBu2     200
#define IATBINBu3     1000
#define IATBINBu4     10000
#define IATBINBu5     100000

#define IATBINWu1     1           // bin width 1ms
#define IATBINWu2     5
#define IATBINWu3     10
#define IATBINWu4     20
#define IATBINWu5     50

#define IATBINNu1     IATBINBu1 / IATBINWu1                        // # of bins in section one
#define IATBINNu2     (IATBINBu2 - IATBINBu1) / IATBINWu2 + IATBINNu1
#define IATBINNu3     (IATBINBu3 - IATBINBu2) / IATBINWu3 + IATBINNu2
#define IATBINNu4     (IATBINBu4 - IATBINBu3) / IATBINWu4 + IATBINNu3
#define IATBINNu5     (IATBINBu5 - IATBINBu4) / IATBINWu5 + IATBINNu4

#define IATSECMAX 5                    // max # of section in statistics ,last section comprises all elements > IATBINBu4
```

```
// definition of bin count fields
const uint32_t IATBinBu[IATSECMAX+1] = { 0, IATBINBu1, IATBINBu2, IATBINBu3, IATBINBu4, IATBINBu5 };
const uint32_t IATBinWu[IATSECMAX] = { IATBINWu1, IATBINWu2, IATBINWu3, IATBINWu4, IATBINWu5 };
const uint32_t IATBinNu[IATSECMAX+1] = { 0, IATBINNu1, IATBINNu2, IATBINNu3, IATBINNu4, IATBINNu5 };
```

The number of bin sections is defined by `IATSECMAX`, default is 3. The static fields `IATBinBu` and `IATBinWu` need to be adapted when `IATSECMAX` is changed. The static definition in curly brackets of the constant fields `IATBinBu[]`, `IATBinBu[]` and `IATBinBu[]` must adapted as well to the maximal bin size. The constant `IATBINUMAX` including his two dimensional packet length, IAT statistics is being used by the descriptive statistics plugin and can suit as a raw input for subsequent statistical classifiers, such as Bayesian networks or C5.0 trees.

3.13.2 Output format

All PL-IAT bins greater than zero are appended for each flow in the `PREFIX_flows` file using the following format:

```
[ps]_[IAT]_[# packets]_[# of packets PL]_[# of packets IAT]
```

the PL-IAT bins are separated by semicolons. The IAT value is the lower bound of the IAT range of a bin.

3.13.3 Example for PL-IAT distributions

Consider a single flow with the following PL and IAT values:

Packet number	PL (bytes)	IAT (ms)	IAT bin
1	50	0	0
2	70	88.2	17
3	70	84.3	16
4	70	92.9	18
5	70	87.1	17
6	60	91.6	18

Packet number two and five have the same PL-IAT combination. Packets number two to five have the same PL and number two and five as well as the number four and six fall within the same IAT bin. Therefore the following sequence is generated:

```
50_0_1_1_1 ; 60_90_1_1_2 ; 70_80_1_4_1 ; 70_85_2_4_2 ; 70_90_1_4_2
```

Note that for better readability spaces are inserted around the semicolons which will not exist in the text based flow file!

3.13.4 Customizing the PL-IAT distribution output

The user is able to customize the output by changing several define statements in the header file *packetSizeInterArrival-TimeHisto.h*. Every change requires a recompilation of the plugin using the *autogen.sh* script.
`HISTO_PRINT_BIN == 0`, the default case, selects the number of the IAT bin, while 1 supplies the lower bound of the IAT bin's range.
As being outlined in the Descriptive Statistics plugin the output of the plugin can be suppressed by defining `PRINT_HISTO` to zero.
For specific applications in the AI regime, the distribution can be directed into a separate file if the value `PRINT_HISTO_IN _SEPARATE_FILE` is different from zero. The suffix for the distribution file is defined by the `HISTO_FILE_SUFFIX` define.

3.13.5 Gnuplot/Excel/SPSS support

By invoking the script `statGplt` under *trunk/scripts* files are generated for the 2/3 dim statistics in a Gnuplot/Excel/SPSS column oriented format. The format is shown below:

For the 3 D case:

PL	IAT	count

and for the 2 D case:

PL	count

3.14 MAC Recorder *

3.14.1 Description

The Mac Recorder plugin provides the source- and destination MAC address as well as the number of packets detected in the flow separated by an underscore ("_"). If there is more than one combination of MAC addresses e.g. due to load balancing or router misconfiguration, the plugin prints all recognized Mac addresses separated by semicolons.

3.14.2 Example

Consider a host with MAC address `AA:AA:AA:AA` in a local network requesting a website from a public server. Due to load balancing the opposite flow can be split and transmitted via two routers with MAC addresses `BB:BB:BB:BB` and `CC:CC:CC:CC`. The MAC Recorder plugin then produces the following output:

```
BB:BB:BB:BB_AA:AA:AA:AA_667;CC:CC:CC:CC_AA:AA:AA:AA_666
```

3.15 Port-based Classifier *

This plugin classifies the flow according to the destination port meaning. It accepts a default port list `portmap.txt` in */.tranalyzer/plugin* in the user's home directory. If tt ./autogen.sh is invoked in the plugin's root folder for the first time a default `port.txt` file is copied to the plugin folder.

3.16 Protocol Statistics *

Provides protocol/port sorted frequency statistics about the observed OSI layer 4 protocols and ports to the file named `PREFIX_protocols`. Protocols numbers are decoded via a `proto.txt` file. In order to identify the biggest talkers the script `protStat` is supplied which produces lists sorted according to the occurrence of the protocol and port feature. If `./autogen.sh` is invoked in the plugin's root folder for the first time a default `proto.txt` file is copied to the plugin folder.

3.17 I/O Buffer

This plugin buffers pcap data either from an interface or a file up to `IO_BUFFER_SIZE` elements, thus being able to hold `MAX_MTU` bytes per packet. It serves as an intermittent storage for traffic bursts in case of processor or bus overload. Although the I/O Buffer Plugin uses a different thread no multicore CPU is required to produce a higher performance because the program is I/O bound, thus file or interface operations are considerable longer than CPU time.
Note: The I/O Buffer Plugin is an internal plugin that can be enabled or disabled in `tranalyzer.h` by changing the define value: `ENABLE_IO_BUFFERING`.

4 Creating a custom plugin

A plugin is a shared library file comprising of special functionality. Tranalyzer2 dynamically loads these shared libraries at runtime from the *.tranalyzer/plugins* directory in the user's home folder. Therefore Tranalyzer2 is available for users if being installed in the */usr/local/bin* directory while the plugins are user dependent. To develop a plugin it is strongly recommended that the user utilizes our special "skeleton" plugin. It is available via SVN from the Tranalyzer repository. This skeleton contains a header and source file comprising of all mandatory and optional functions as well as a small `HOWTO` file and a script to build and move a shared library to the plugins folder.

4.1 Preparation

Usage of the skeleton plugin requires the main folder of the Tranalyzer source files to be located in the same folder as the main folder of the skeleton. Also the tools automake and libtool must be installed. Further directions concerning the `configure.ac` and the `Makefile.am` are located in the `HOWTO` file.

4.2 Accessible structures

Due to practical reasons all plugins are able to access every structure of the main program and the other plugins. This is indeed a security risk, but since Tranalyzer2 is a tool for practitioners and scientists in access limited environments the maximum possible freedom of the programmer is more important for us.

4.2.1 Accessible structures under Mac OS X

When you write a plugin which should run also under Mac OS X you will realize that it handles global variables in a different way than Linux. To access a global variable from the main program or the other plugins you have to write your own getter or use the existing once. For examples see basicFlowOutput, a for a plugin dependency example see **descriptiveStatistics**.

4.3 Important structures

A predominant structure in the main program is the flow table *flow* where the six tuple for the flow lookup timing information is stored as well as a pointer to a possible opposite flow. A plugin can access this structure by including the `packetCapture.h` header. For more information please refer to the header file.
Another important structure is the main output buffer `mainOutputBuffer`. This structure holds all standard output of activated plugins whenever a flow is terminated. The main output buffer is accessible if the plugin includes the header file `main.h`.

4.4 Generating output

As mentioned in Section 2.7 there are two ways to generate output. The first is the case where a plugin just writes its arbitrary output into its own file, the second is writing flow-based information to a standard output file. We are now discussing the later case.
The standard output file generated by the Standard File sink plugin consists of a header, a delimiter and values. The header is generated using header information provided by each plugin, that writes output into the standard output file. During the initialization phase of the sniffing process, the core calls the `printHeader()` functions of these plugins. These functions return a single structure or a list of structures of type `binary_value_t`. Each structure represents a statistic. To provide a mechanism for hierarchical ordering, the statistic itself may contain one ore more values and one or more substructures. The structure contains the following fields:

Field name	Field type	Explanation
num_values	uint32_t	Amount of values in the statistic
subval	binary_subvalue_t*	Type definition of the values
name_value_short	char[128]	Short definition of the statistic
name_value_long	char[1024]	Long definition of the statistic
is_repeating	uint32_t	one, if the statistic is repeating, zero otherwise
next	binary_value_t*	used if the plugin provides more than one statistics

The substructure `binary_subvalue_t` is used to describe the values of the statistic. For each value, one substructure is required. For example, if `num_values` is two, two substructures have to be allocated. The substructures must be implemented as a continuous array consisting of the following fields:

Field name	Field type	Explanation
value_type	uint32_t	Type of the value
num_values	uint32_t	Amount of values in the statistic
subval	binary_subvalue_t*	Definition of the values
is_repeating	uint32_t	one, Statstic is repeating, zero otherwise

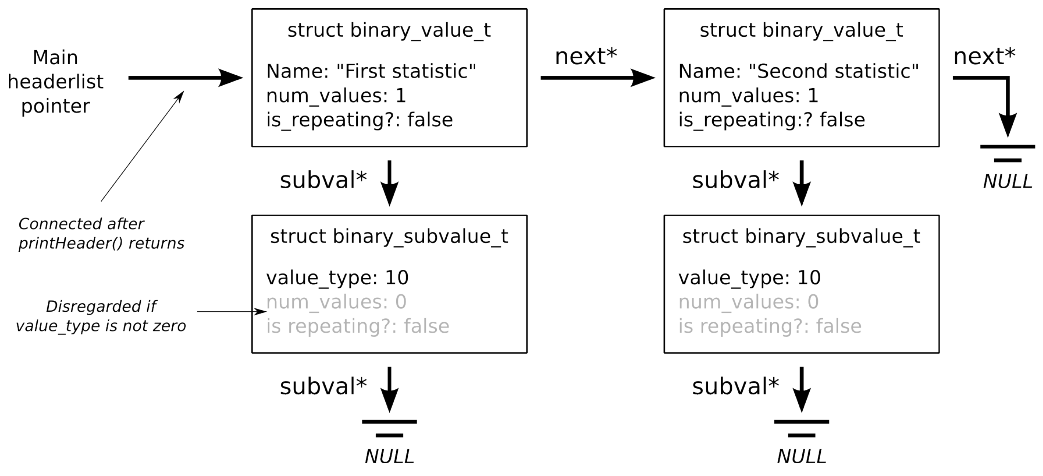
Compared to the `binary_value_t` representation two strings are omitted in the statistic's short and long description and the *next pointer but it contains a new field, the value type. Possible values for this new field are described in the enumeration `binary_types` defined in the header file `binaryValue.h`. If the field contains a value greater than zero the fields `num_values` and `subval` are ignored. They are needed if a subval contains itself subvalues. To indicate additional subvalues, the field `value_type` need to be set to zero. The mechanism is the same as for the `binary_value_t`.

The field `is_repeating` should be used if the number of values inside a statistic is variable; e.g. a statistic of a vector with variable length; see 3.12).

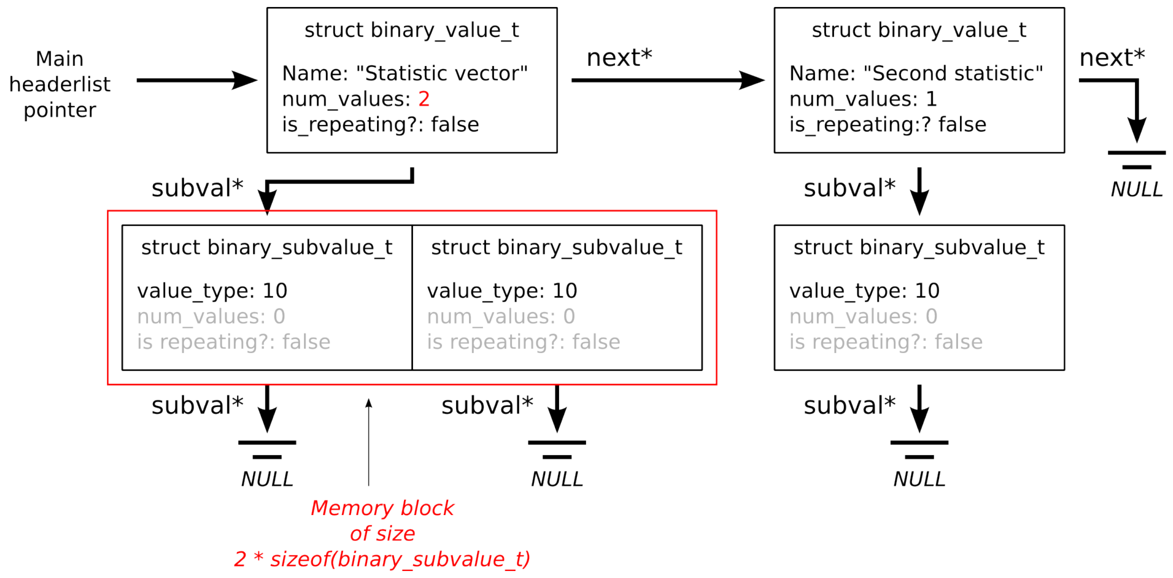
4.4.1 Examples

The following examples illustrate the usage of the said two structures:

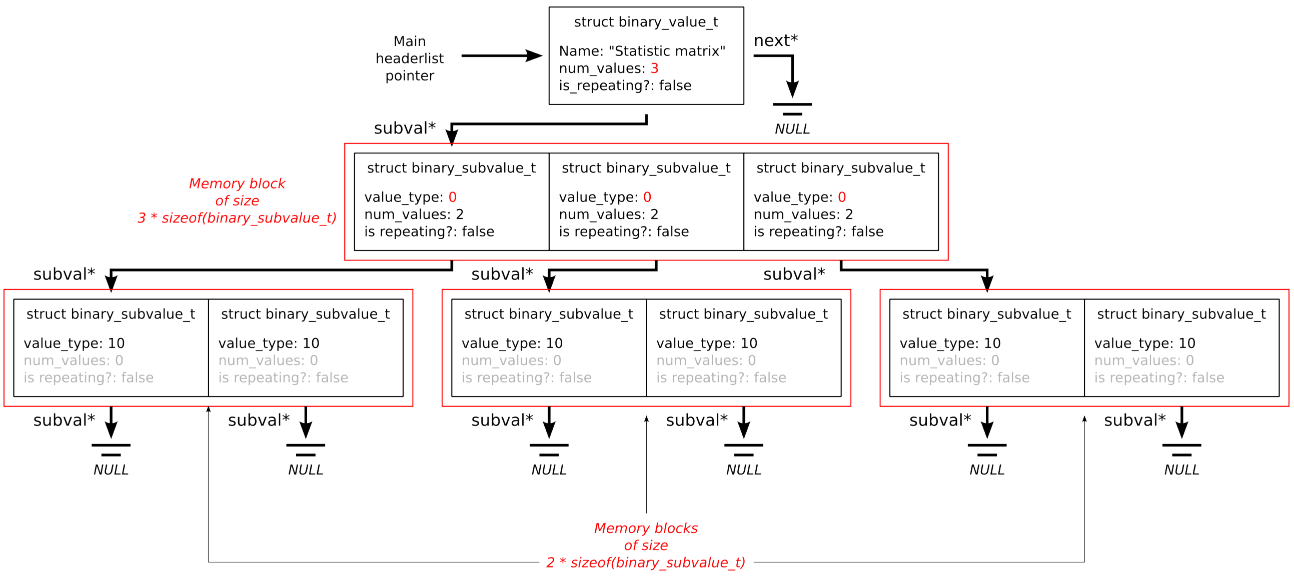
Example 1: Two Statistics each containing a single value If a plugin's output is consisting of two statistics each having a single value it needs to pass a list containing two structures of type `binary_value_t`. Both structures contain a substructure with the type of the single values. The following diagram shows the relationships between all four structures:



Example 2: A statistic composed of two values Now the output of the plugin is again two statistics, but the first statistic consists of two values; e.g. to describe a position on a grid. Therefore `num_values` is two and `subval*` points to a memory field of size two-times struct `binary_subvalue_t`. The subvalues themselves contain again the type of the statistic's values. Note: These values do not need to be identical.



Example 3: A statistic containing a complete matrix With the ability to define subvalues in subvalues it is possible to store multidimensional structures such as matrices. The following example illustrates the definition of a matrix of size three times two:



4.4.2 Helper functions

In order to avoid filling the structures by hand a small API is located in the header file `binaryValue.h` doing all the nity gritty work for the programmer. The therefor important four functions are described below.

`binary_value_t* bv_append_bv(binary_value_t* dest, binary_value_t* new)`
Appends a binary_value_t struct at the end of a list of binary_value_t structures and returns a pointer to the start of the list.

Arguments:

Type	Name	Explanation
binary_value_t*	dest	The pointer to the start of the list
binary_value_t*	new	The pointer to the new binary_value_t structure

binary_value_t* bv_new_bv (char* name_long, char* name_short, uint32_t is_repeating, uint32_t num_values...)

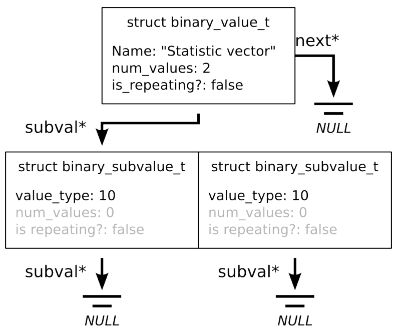
Generates a new structure of type binary_value_t and returns a pointer to it

Arguments:

Type	Name	Explanation
char*	name_long	a long name for the statistic
char*	name_short	a short name for the statistic
uint32_t	is_repeating	one, if the statistic is repeating, zero otherwise
uint32_t	num_values	the number of values for the statistic
int	...	the types of the statistical values, repated `num_values`-times

The function creates a `binary_value_t` structure and sets the values. In addition, it creates an array field with num_values binary_subvalue_t structures and fills the value types provided in the variable argument list.

Example: The call `bv_new_bv("Statistic vector", "stat_vec", 2, 0, bt_uint_64, bt_uint_64)` creates the following structures:



`binary_value_t* bv_add_sv_to_bv (binary_value_t* dest, uint32_t pos,`
`uint32_t is_repeating, uint32_t num_values, ...)`

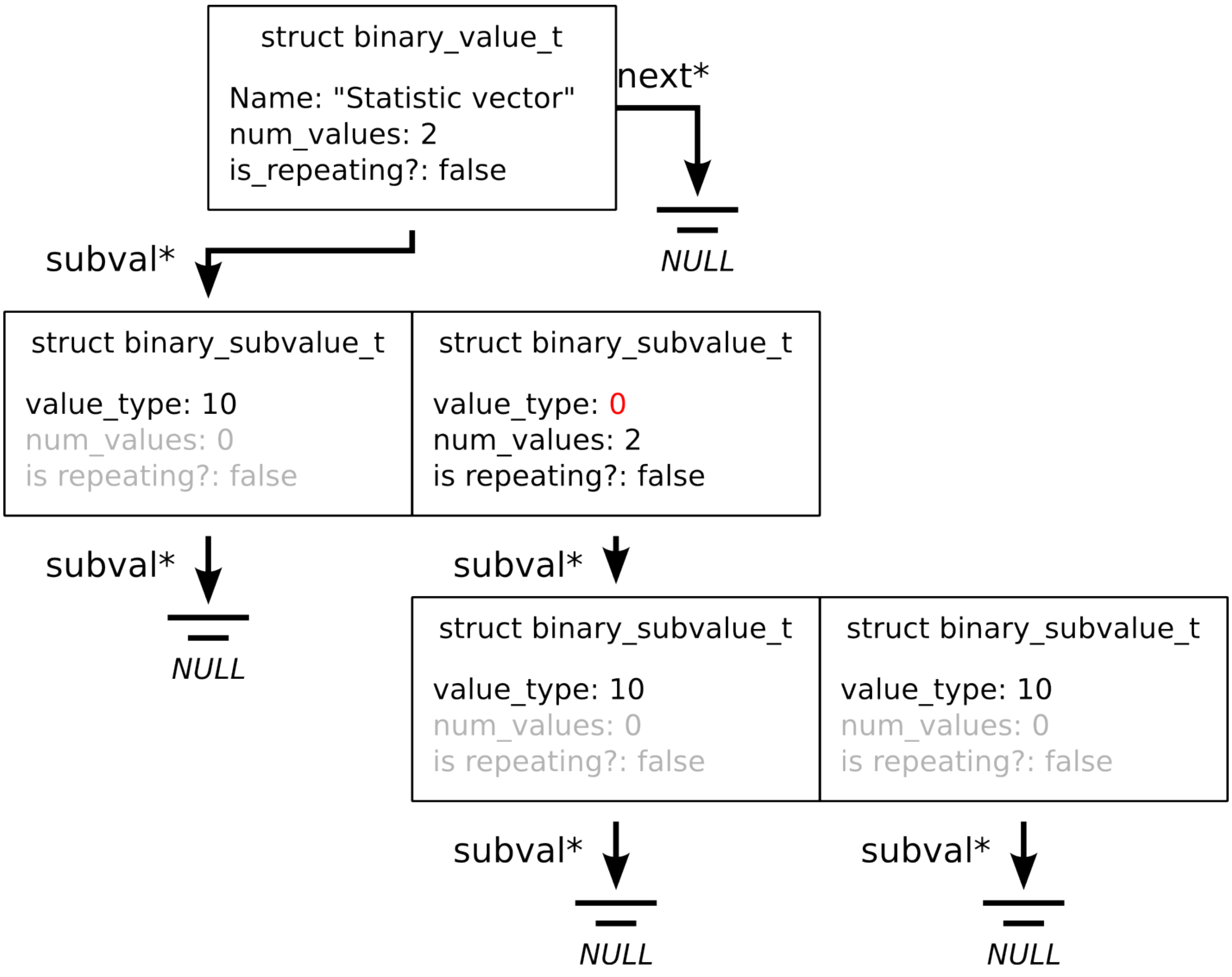
Replaces a subvalue in a `binary_value_t` *structure with a new substructure that contains additional substructures and returns a pointer to the parent binary value.*

Arguments:

Type	Name	Explanation
binary_value_t*	dest	the pointer to the parent binary value
uint32_t	pos	the position of the substructure to be replaced, starting at 0
uint32_t	is_repeating	one, if the subvalue is repeating, zero otherwise
uint32_t	num_values	the number of values in the subvalue
int	...	the types of the statistical values, repated *num_values*-times

This function is only valid if `dest` is already a complete statistic containing all necessary structures.

Example: Let *dest* be a pointer to the `binary_value_t` structure from the example above. A call to the function `bv_add_sv_to_bv(dest, 1, 0, 2, bt_uint_64, bt_uint_64)` replaces the second substructure with a new substructure containing two more substructures:



```
binary_value_t* bv_add_sv_to_sv (binary_subvalue_t* dest, uint32_t pos,
                                  uint32_t is_repeating, uint32_t num_values, ...)
```

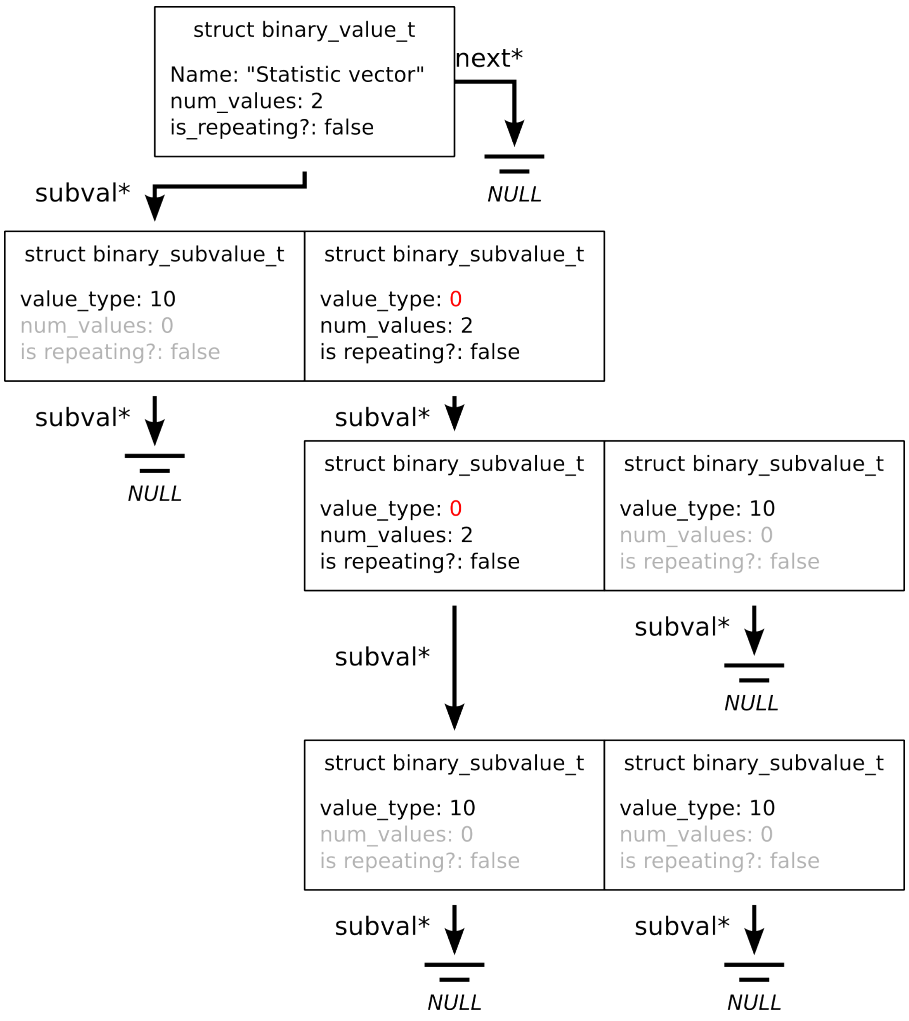
Replaces a subvalue in a `binary_subvalue_t` *structure with a new substructure that contains additional substructures and returns a pointer to the parent binary subvalue.*

Arguments:

Type	Name	Explanation
binary_subvalue_t*	dest	Pointer to the parent binary subvalue
uint32_t	pos	Position of the substructure to be replaced, starting at 0
uint32_t	is_repeating	one, if the subvalue is repeating, zero otherwise
uint32_t	num_values	Number of values in the subvalue
int	...	Types of the statistical values, repeated *num_values*-times

For all hierarchical deeper located structures than above the function described above is required.

Example: Let *dest* be a pointer to the subvalue structure being replaced in the example above. A call to the function `bv_add_sv_to_sv(dest, 0, 0, 2, bt_uint_64, bt_uint_64)` replaces *dest's* first the substructure with a new substructure containing two more substructures:



4.4.3 Writing into the standard output

Standard output is generated using a buffer structure. Upon the event `onFlowTerminate` (see 4.8.7) Plugins write all output into this buffer. It is strongly recommended using the function `outputBuffer_append(outputBuffer_t* buffer, char* output, size_t size_of_output)`.
Arguments:

Type	Name	Explanation
outputBuffer_t*	buffer	the pointer to the standard output buffer structure, for standard output, this is `main_output_buffer`
char*	output	a pointer to the output, currently of type char
size_t	size_of_output	the length of field *output* in single bytes

The output buffer is send to the *output sinks* after all plugins have stored their information.

Example: If a plugin wants to write two statistics each with a single value of type `uint64_t` it first has to commit its `binary_value_t` structure(s) (see section above). During the call of its `onFlowTerminate()` function the plugin writes both statistical values using the append function:

```
outputbuffer_append(main_output_buffer, (char*) value1, 4);
outputbuffer_append(main_output_buffer, (char*) value2, 4);
```

where `value1` and `value2` are two pointers to the statistical values.

4.5 Writing repeated output

If a statistic could be repeated (field `is_repeating` is one) the plugin has first to store the number of values as `uint32_t` value into the buffer. Afterwards, it appends the values.

Example: A plugin's output is a vector of variable length, the values are of type `uint16_t`. For the current flow, that is terminated in the function `onFlowTerminate()`, there are three values to write. The plugin first writes a field of type `uint32_t` with value three into the buffer, using the append function:

```
outputbuffer_append(main_output_buffer, (char*) numOfValues, sizeof(uint32_t));
```

Afterwards, it writes the tree values.

4.6 Important notes

- IP addresses (bt_ip4_addr or bt_ip6_addr) or MAC addresses (bt_mac_addr) are stored in network order.
- Strings are of variable length and need to be stored with a trailing zero bit ('\0').

4.7 Administrative functions

Every plugin has to provide five administrative functions. The first four are mandatory while the last is optional. Their existence is checked during the plugin initialization phase one and two:

Function name	Return type	Explanation
get_plugin_name()	char*	a unique name of the plugin, not necessarilythe filename. All characters except the comma is allowed.
get_plugin_version()	char*	a version number, usually a dot separated 3 tupel (x.y.z)
get_supported_tranalyzer_version_major()	unsigned int	The minimum major version number of the main program being supported by the plugin
get_supported_tranalyzer_version_minor()	unsigned int	The minimum minor version number in combination with the minimum major version number of the main program being supported by the plugin
get_dependencies()	char*	if exists, the plugin loader checks the avalability of the plugin names returned by this function. The plugin names have to be separated by a comma. White spaces, tabs or any other characters are **not** treated as name separators.

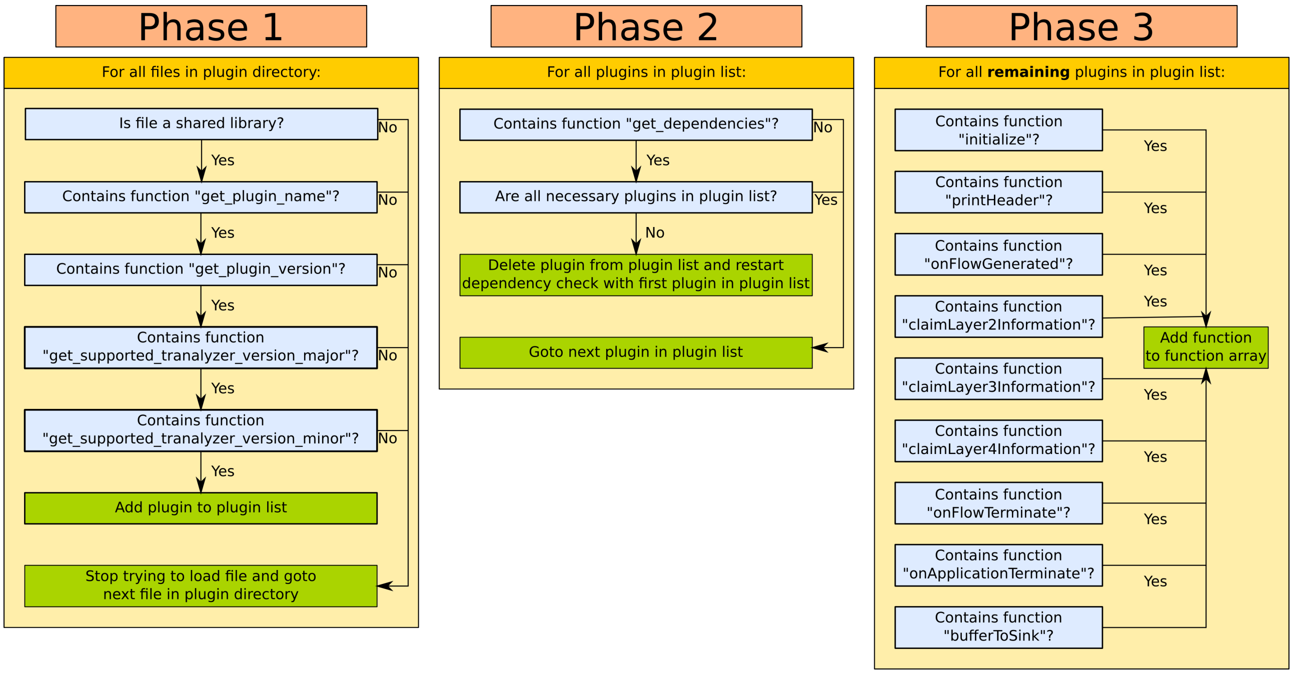

Phase 1
For all files in plugin directory:
Is file a shared library?
Contains function "get_plugin_name"?
Contains function "get_plugin_version"?
Contains function "get_supported_tranalyzer_version_major"?
Contains function "get_supported_tranalyzer_version_minor"?
Add plugin to plugin list
Stop trying to load file and goto next file in plugin directory
Yes
No
Phase 2
For all plugins in plugin list:
Contains function "get_dependencies"?
Are all necessary plugins in plugin list?
Delete plugin from plugin list and restart dependency check with first plugin in plugin list
Goto next plugin in plugin list
Phase 3
For all **remaining** plugins in plugin list:
Contains function "initialize"?
Contains function "printHeader"?
Contains function "onFlowGenerated"?
Contains function "claimLayer2Information"?
Contains function "claimLayer3Information"?
Contains function "claimLayer4Information"?
Contains function "onFlowTerminate"?
Contains function "onApplicationTerminate"?
Contains function "bufferToSink"?
Add function to function array

Figure 10: *Processing of the plugin loading mechanism*

4.8 Processing functions

During flow analysis Tranalyzer2 generates several *events* based on the status of the program, the inspected OSI layer of the current packet or the status of the current flow. These events consist of specific function calls provided by the plugins. The implementation of the event functions is dependent on the required action of a plugin to be carried out upon a certain event.

4.8.1 Event: initialize()

Event / function name	Return type	Parameters
initialize	void	—

The `initialize` event is generated before the program activates the packet capturing phase. After Tranalyzer2 has initialized its internal structures it grants the same phase to the plugins. Therefore temporary values should be allocated during that event by using a C `malloc`.

4.8.2 Event: printHeader()

Event / function name	Return type	Parameters
printHeader	binary_value_t*	—

This event is also generated during the initialization phase. With this event the plugin providing data to the standard output file signals the core what type of output they want to write (see 4.4). The function returns a pointer to the generated `binary_value_t` structure or to the start pointer of a list of generated `binary_value_t` structures.

4.8.3 Event: onFlowGenerated()

Event / function name	Return type	Parameters
onFlowGenerated	void	packet_t *packet, unsigned long flowIndex

This event is generated every time Tranalyzer2 recognizes a new flow not present in the flow table. The first parameter is the currently processed packet, the second denotes the new generated flow index. As long as the flow is not terminated the flow index is valid. After flow termination the flow number is reintegrated into a list for later reuse.

4.8.4 Event: claimLayer2Information()

Event / function name	Return type	Parameters
claimLayer2Information	void	packet_t *packet

This event is generated for every new packet comprising of a valid and supported layer two header, e.g. Ethernet as default. This is the first event generated after libpcap dispatches a packet and before a lookup in the flow table happened. At this very point in time no tests are conducted for higher layer headers. If a plugin tries to access higher layer structures it has to test itself if they are present or not. Otherwise, at non-presence of higher layers an unchecked access can result in a `NULL` pointer access and therefore in a possible segmentation fault! We recommend using the subsequent two events to access higher layers.

4.8.5 Event: claimLayer3Information()

Event / function name	Return type	Parameters
claimLayer3Information	void	packet_t *packet

This event is generated for every new packet comprising of a valid and supported layer three header. The currently supported layer three headers are IP and IP encapsulated in a variable number of VLAN headers. The event is generated after the `claimLayer2Information` event and before a lookup in the flow table is performed. Again, no tests are conducted for higher layer headers. If a plugin tries to access higher layer structures it has to test their existence. If not present an unchecked access can result in a `NULL` pointer access and therefore in a possible segmentation fault! We recommend using the subsequent event to access higher layers.

4.8.6 Event: claimLayer4Information()

Event / function name	Return type	Parameters
claimLayer4Information	void	packet_t *packet, unsigned long flowIndex

This event is generated for every new packet containing a valid and supported layer four header. The current supported layer four headers are TCP, UDP and ICMP. This event is called after Tranalyzer2 performs a lookup in its flow table and eventually generates an `onFlowGenerated` event. Implementation of other protocols such as IPsec or OSPF are planned.

4.8.7 Event: onFlowTerminate()

Event / function name	Return type	Parameters
onFlowTerminate	void	unsigned long flowIndex

This event is generated every time Tranalyzer2 removes a flow from its active status either due to timeout or protocol normal or abnormal termination. Only during this event, the plugins write output to the standard output.

4.8.8 Event: onApplicationTerminate()

Event / function name	Return type	Parameters
onFlowTerminate	void	—

This event is generated shortly before the program is terminated. At this time no more packets or flows are processed. This event enables the plugins to do memory housekeeping, stream buffer flushing or printing of final statistics.

4.8.9 Event: bufferToSink()

Event / function name	Return type	Parameters
bufferToSink	void	outputBuffer* buffer

The Tranalyzer core generates this event immediately after the `onFlowTerminate` event with the main output buffer as parameter. A plugin listening to this event is able to write this buffer to a data sink. For example the standard file sink plugin pushes the output into the `PREFIX_flows` file.

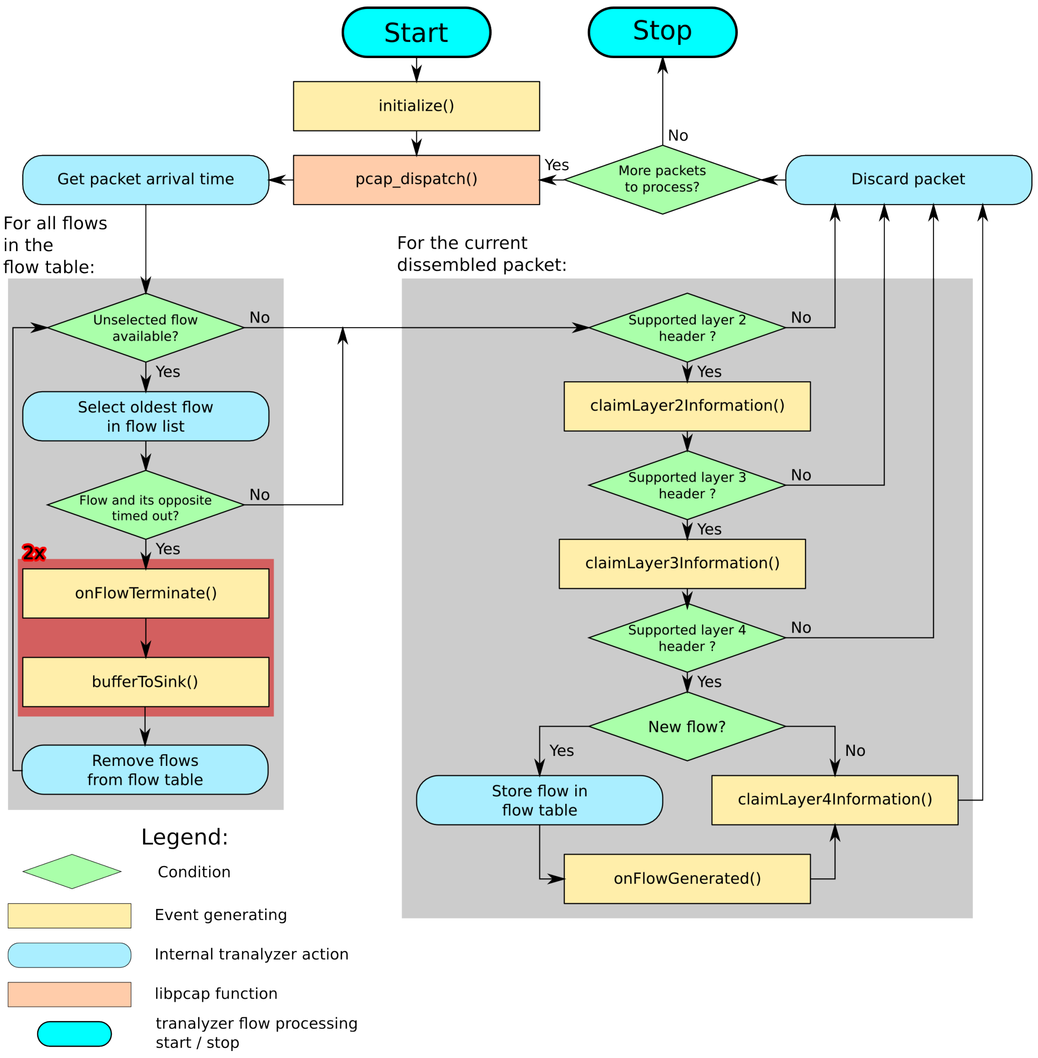


Figure 11: *Tranalyzer packet processing and event generation.*

4.9 Timeout handlers

A flow is terminated after a certain timeout being defined by so called *timeout handlers*. The default timeout value for a flow is 182 seconds. The plugins are able to access and change this value. For example, the TCP States plugin changes the value according to different connection states of a TCP flow.

4.9.1 Registering a new timeout handler

To register a new timeout handler, a plugin has to call the function `timeout_handler_add(float timeout_in_sec`. The argument is the new timeout value in seconds. Now the plugin is authorized by the core to change the timeout of a flow to the registered timeout value. Without registering a timeout handler the test is unreliable.

4.9.2 Programming convention and hints

- A call of `timeout_handler_add` should only happen during the initialization function of the plugin.
- Registering the same timeout value twice is no factor.
- Registering timeout values in fractions of seconds is possible, see TCP States plugin.

5 Code segments

5.1 main.c

The `main.c` file is the core file of Tranalyzer2. It initializes and prepares the program for execution, manages flow lists and data integrity.

5.1.1 Function main()

It is responsible for validating the application's parameter, signal capture and `mainLoop` start.

5.1.2 Function mainLoop()

The function `mainLoop()` contains the loop being executed as long as data is available or until the termination `KILL` signal is received. In either case the variable `running` will be set to zero in order to signal program termination.
All active flows are linked by a *last recently used list*, denoted as *LRU list*. The most recent flow is located at the beginning of the list after the `lruHead` position, whereas the oldest flow is located at the tail of the list before the `lruTail` position. Whenever a new packet is captured the corresponding flow will be moved to the beginning of the LRU list after the `lruHead` position. After having processed `PACKETS_PER_BURST` packets from the source, the LRU list is traversed twice: The list is processed from tail to head in order to terminate and remove flows being either in time-out or not being successfully terminated; e.g. due to a TCP Reset.
In both cases the traversal will be aborted when the specified case does not apply anymore.

5.1.3 Function removeFlow()

Removes the flow given as argument to the function as well as its opposite flow from the hashtable and from the LRU list.

5.1.4 Function lruFlowPrintout()

Takes the variable `lruPointer` as input and decides whether `lruPointer` or its opposite flow is the flow initiator. Eventually calls the appropriate output function.

5.1.5 Function printFlow()

Calls each plugin's function `pluginOnFlowTerminate` twice: The first time for the forward flow and the second time for its reverse flow.

5.1.6 Function prepareSniffing()

Initializes variables and calls the function `pluginInitialize` for each available plugin. It also opens files in write mode 'w' and initializes the pcap library.

5.1.7 Function copy_argv()

Parses the remaining arguments to the application as *BPF filter*, which then filters packets matching the defined pattern.

5.1.8 Function printUsage()

Prints the usage text to standard output.

5.1.9 Function terminate()

Terminates the application. Flushes all buffers and closes the pcap library and all files still being open.

5.1.10 Function catchInterrupt()

The function that is executed upon receiving either `SIGINT` or `SIGTERM`. Currently prepares the application for termination.

5.1.11 Function userInterrupt()

The function that is executed upon receiving either `SIGUSR1` or `SIGUSR2` prints various information such as file processing progress, run time, etc.

5.1.12 Function timevalSubtract()

Substracts the timeval y from the timeval x and returns the result by value.

5.2 packetCapture.c

5.2.1 Function perPacketCallback()

The function `perPacketCallback()` is called with each packet that does not match the specified BPF filter. The packet traverses all specified OSI layer analysis sections from one to seven. Currently only layer two, three and four are implemented being associated to either a new flow or an existing flow. During the traversal phase the flow is updated with every new information being gathered from the packet headers.

5.2.2 Function flowCreate()

Creates a new flow comprising of all available header information of the first packet. The generated flow is also inserted into the hashtable and LRU list. When an opposite flow is detected, a link will be created between both flows. For each active plugin the function `pluginOnFlowGenerated` is called.

5.2.3 Function updateLRUList()

This function moves a flow supplied as argument to the beginning of the LRU list, thus after the position `lruHead`.

6 Glossary

Name	Description
Flow	A distinct packet stream from a source host to a destination host