CESNET Technical Report 9/2013

Nemea: Framework for stream-wise analysis of network traffic

Václav Bartoš, Martin Žádník, Tomáš Čejka

Received 13. 12. 2013

Abstract

Since network anomalies, attacks and other incidents are no longer exceptions network monitoring and analysis play important roles in network administration. This technical report deals with traffic analysis in high-speed network. In particular the report describes our framework Nemea which provides stream-wise analysis of data on network traffic.

Keywords: network, security, analysis

1 Introduction

Network services, applications as well as connected users are targets of increasing network security threats. Some of these threats are best detected when the network traffic is monitored and the observed data are analyzed in an automated way. A common approach is to collect and to store the monitored data into a storage system (*e.g.* multiple hard drives set up in one of the RAID modes). Subsequently, the analysis operates over the stored data. The analysis looks for suspicious behaviour or particular communication pattern to reveal anomalies, attacks or other non-compliant events and states. As the networks and the threats evolve so must the analysis. To this end, we research and develop streaming data handling framework as an alternative for a tradional store-and-ex-post analysis approach.

In the case of a large network it is necessary to collect and analyze a large amount of data. In 2013, our metering points generated approximately 10 billion of flow records per day (that is approximately 110 000 flows/s on average with 160 000 flows/s during business hours and peaks up to 400 000 flows/s). The metering points are deployed at the perimeter of CESNET2 network and each monitors one or more 10 Gbps lines. The pilot deployment of the metering points is described in [1].

The store-and-ex-post approach poses certain limits. The amount of monitored data as well as the complex methods renders the access to data stored on the hard drive unfeasible. Emerging threats as well as a need to achieve better detection precision require to analyze new type of data and to analyze data as soon as the data are available not to delay a potential response. Moreover the data are usually stored in chunks and each chunk is analyzed separately. This separation renders the detection blind to some threats spanning the chunks.

This technical report summarizes our goal to enable online and seamless network traffic analysis. The report describes a distributed and modular framework for network flow analysis (Nemea) as well as our TRaffic Analysis Platform (TRAP) for fast data transfers among network analysis processes instantiated in Nemea. Nemea

© CESNET, 2014

system consists of a set of modules. Each module is an independent process which utilizes TRAP to communicate with other processes. Typically the modules are connected in a unidirectional tree where the input of the root is typically data on flows and the outputs of the leaves are the results of the analysis.

Nemea data access differs from the traditional approach significantly. The data handling in Nemea follows two concepts: stream-wise processing and no data storage on the hard drive. This enables prompt data analysis and fast data access. Moreover TRAP allows for communication among processes hosted on different machines transparently to the processes themselves.

The report is organized as follows. First, Nemea framework is described from a high-level point of view in Section 2. Then the Traffic Analysis Platform, which is a low-level basis of Nemea, and a flexible data format used by Nemea modules called UniRec are described in Section 3. Results of performance tests of the platform are presented in Section 4. Section 5 briefly presents selected Nemea modules. Section 6 concludes this technical report and outlines our future work.

2 Nemea

Network Measurements Analysis (Nemea) is a framework which allows for an assembly of a system for automated analysis of flow records gathered from network monitoring processes in real time. The system consists of separate building blocks called modules. The modules are interconnected by interfaces.

The module is an independent working unit which generally receives a stream of data records on its input(s), processes or analyzes the data records and sends another stream of records to its output(s). The module may, for example, compute some statistics based on the received flow records and detect a network attack. The attack is described by a data record, which is sent to the output of the module. Other modules may detect anomalies utilizing statistics computed previously by another module, or may aggregate detection results and report the most important ones to a human operator or to other reporting systems. A complex system for real-time analysis of network traffic is assembled by interconnecting several such modules.

Nemea is also compatible with store-and-ex-post approach as demonstrated on a minimal example of a Nemea instance on Figure 1. There are two modules interconnected by one interface. The first module reads flow records from a file and sends them to the second module. The second module counts total number of flows, packets and bytes. Figure 2 shows an example of more complex system, where flow data are received from a network in real time via a plugin for our IPFIX collector [4]. The data are preprocessed, analyzed by several attack/anomaly detection algorithms, detection results are aggregated and then reported. Each particular task is implemented by an independent module. This allows to instantiate certain modules at various points of the system as well as to share the results of one module to multiple other modules.

One of the main advantages of Nemea modular design is the possibility to share common tasks by several algorithms. For example, there exists a lot of algorithms for detection of anomalies in time series of a traffic volume, an entropy of packet header fields or other statistics. As shown in the example on Figure 2, there is a

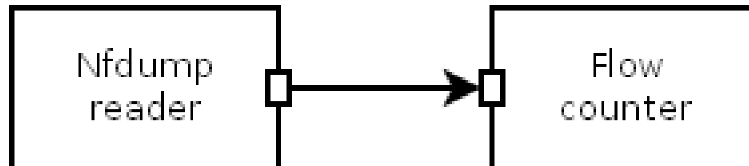

Figure 1. A minimal example of Nemea system.

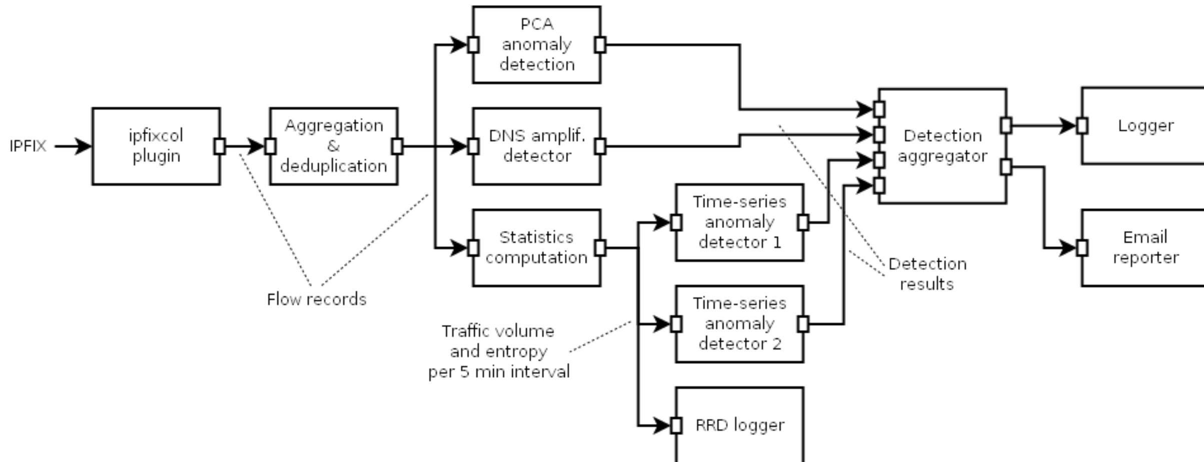

Figure 2. An example of more complex Nemea system. There are modules for data acquisition, preprocessing, attack and anomaly detection, postprocessing, storage and reporting.

module which computes the statistics from the incoming flow records and sends them to several time-series modules regularly. Thus the common preprocessing task is shared by all the algorithms. Other examples of the shared preprocessing include an aggregation&deduplication module or prefix filtering modules connected before analysis modules themselves.

2.1 Modules

Each module is a standalone application, *i.e.* it runs as a separate system process. While this may be a little less efficient than other possible solutions, such as implementing modules just as separate pieces of code all linked into one binary or compiling modules into shared libraries, which are later dynamically loaded into one process, this solution offers several advantages.

First, it is easy to start and stop processes independently on each other and to monitor their resource usage by means provided by operating system. It is also possible to develop modules in various programming languages, even the interpreted ones (*e.g.* Python), which would be much harder if all the modules ran as a single process. Last but not least, when each module runs as a separate process, a malfunctioning module does not crash the whole system (*e.g.* by a segmentation fault). Quotas on memory and other system resources can also be set up to avoid a malfunctioning module disabling the system by consuming all resources of a host.

The Nemea modules can be added to or removed from the system dynamically. When a new module is started, it tries to connect to specified interfaces of other modules. If a connection is not possible at the time (*e.g.* because the other module

has not been started yet), it tries to connect periodically until the connection is established. Similarly, when some connection is lost, the module tries to reconnect automatically. Due to this behaviour, addition or removal of a module to/from the system is as easy as running or stopping an application. Such an ability of "hot plugging" and "hot swapping" of the modules is very useful for adding new analysis algorithms or replacing existing modules with their newer versions on the fly with no or little impact on the rest of the system.

2.2 Interfaces

All interfaces are one-way only and they transfer data in the form of individual records (*e.g.* flow records or records describing detected anomalies). All records sent through one interface must be of the same format, *i.e.* they must contain the same set of fields, but different interfaces may use different formats.

The format used by the given interface is specified dynamically when the module is being connected into the system. The dynamic format specification renders it possible, for example, to add a new field to the flow record without a need to change the code of the modules processing these records. A protocol which specifies how to define these formats and how to create and to use the records is called UniRec. See Section 3.2 for more information.

There are different types of interfaces distinguished by an underlying method of communication. There are reliable interfaces for communication within the same host as well as between different hosts, all with the unified and simple API. Parameters of the connection (its type and the other site of the connection) are given at the time of the module start. The module is completely indepedent on the particular interface type since the interconnection mechanisms are hidden to the module. More information about interfaces is presented in Section 3.1.

Nemea utilizes Traffic Analysis Platform (TRAP) - a library (shared library called libtrap) which implements these interfaces. The TRAP is described in Section 3.

2.3 Streaming concept

The modules in Nemea framework process data stream-wise, *i.e.* data arrives to a module as individual records one by one. The records are not grouped into packs or bursts of any kind (unless it is done by the module explicitly).

The stream-wise and on-the-fly processing is one of the main advantages over the existing network traffic analysis frameworks. For example, a state-of-the-art flow collector NfSen supports plugins for analysis of collected data. But the data are first stored into files, each file containing data from a 5 minute interval. The file is processed by the plugins only after it has been completed, so there may be up to 5 minute delay between the time a flow record arrives and the time it is analyzed. There is no such delay in Nemea which results in the prompt analysis.

The proposed concept is also very friendly to a storage subsystem of a host as there is no need to read data from a hard drive. A collector can pass data directly to Nemea system so all automated processing is done entirely in primary memory (RAM). Of course, the data may also be stored to hard drive for later analysis.

2.4 Versatility

Nemea allows for versatile deployment.

For example, if it is necessary to analyze some offline data by the same methods as the online data, all that is required is to replace the collector by a module which reads the data from a file or a database, the rest remains untouched (or may be duplicated, if the online analysis must keep running).

Another interesting Nemea usecase is a possibility to test detection algorithms by online injection of synthesized attack traces into real data. This is shown on Figure 3.

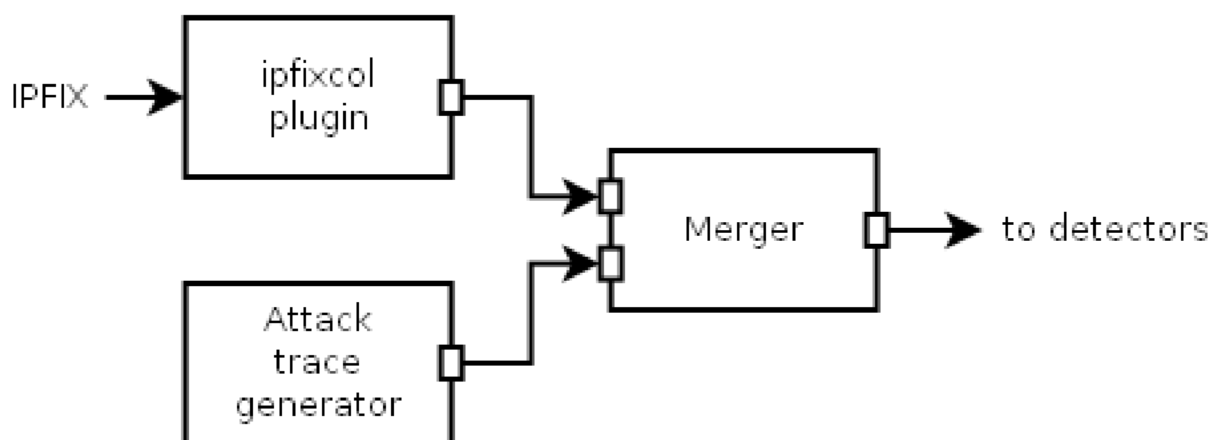

Figure 3. An example in which a module generating attack traces and a module merging two data streams are used to inject synthesized attacks into real data.

Since there are interface types allowing communication over a network, it is easy to distribute Nemea system onto several physical machines. All that is required is to run the modules on different hosts and specify appropriate types of interfaces.

Nemea was designed to be easy to use and to allow rapid implementation of analysis algorithms and other tasks. It offers easy-to-use API with functions to transfer data and it also includes various common data processing functions.

2.5 Nemea setup

It is important to note, that Nemea is a framework and not the final system itself. Nemea defines a high-level architecture and interfaces including TRAP library, UniRec format and several basic modules. A particular analysis system, *i.e.* a set of modules and their interconnection, should always be designed specifically to fit a particular network, available data sources and analysis requirements. Moreover, while Nemea already includes many modules, it is expected that users will also implement some modules themselves to meet their special needs.

Nemea was designed to render the implementation of a module as easy as possible and thus to allow rapid implementation of analysis algorithms and related tasks. Nemea solves data acquisition, parsing of the input data or formatting and reporting/logging the outputs. The framework also provides means to do some other

common tasks easily, such as processing of timestamps and IP addresses or computation of various hash functions. Moreover, the modular nature of Nemea allows to fully utilize the power of modern multi-core processors and/or to distribute the system onto several hosts without any source code modification.

Nemea control script allows to specify the particular Nemea setup. The user specifies its own Nemea configuration in the setup file. The configuration defines the list of instaniated modules, their starting parameters and definition of interconnection among the modules. The control script allows to start and stop all modules in the given configuration or to start/stop any module independently.

3 Traffic Analysis Platform

We have developed Traffic Analysis Platform to stream data in Nemea efficiently.

3.1 Communication Layer (libtrap)

The Nemea modules utilize a common communication interface. The communication interface is represented by a shared object called libtrap, which is linked by every Nemea module. The concept of the communication between two modules is shown in Figure 4.

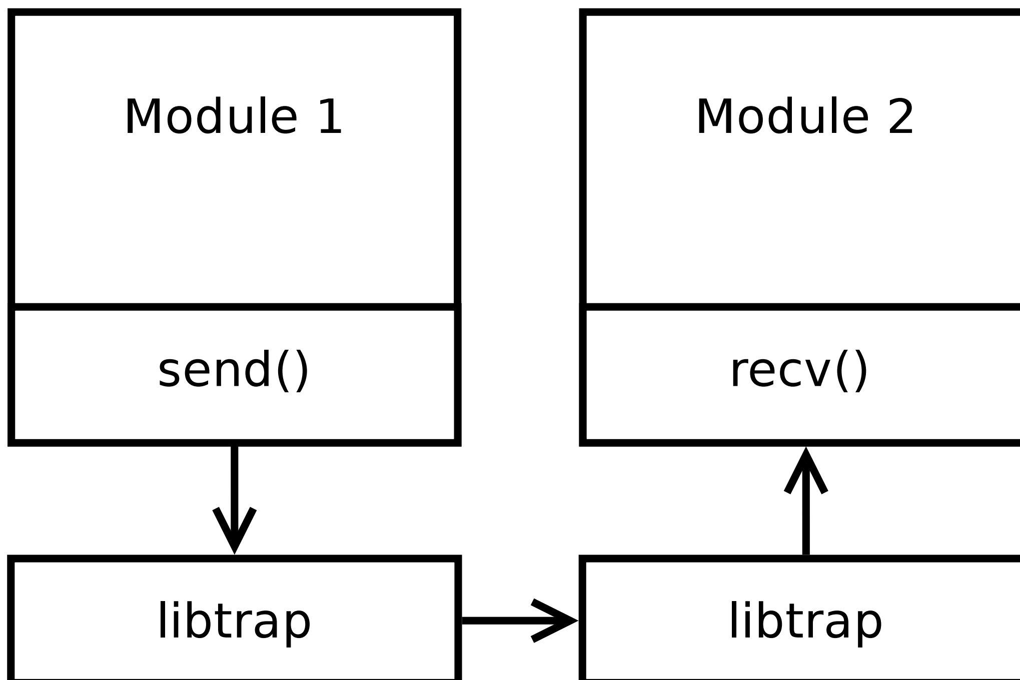

Figure 4. A typical inter-module communication in the Nemea system.

Libtrap abstracts the module from the actual underlying interface and its specifics. The source module sends the data out of the output interface as soon as the data are available. The send operation may be non-blocking as well as blocking based on configuration parameters. Libtrap takes care of the buffering as well as of the

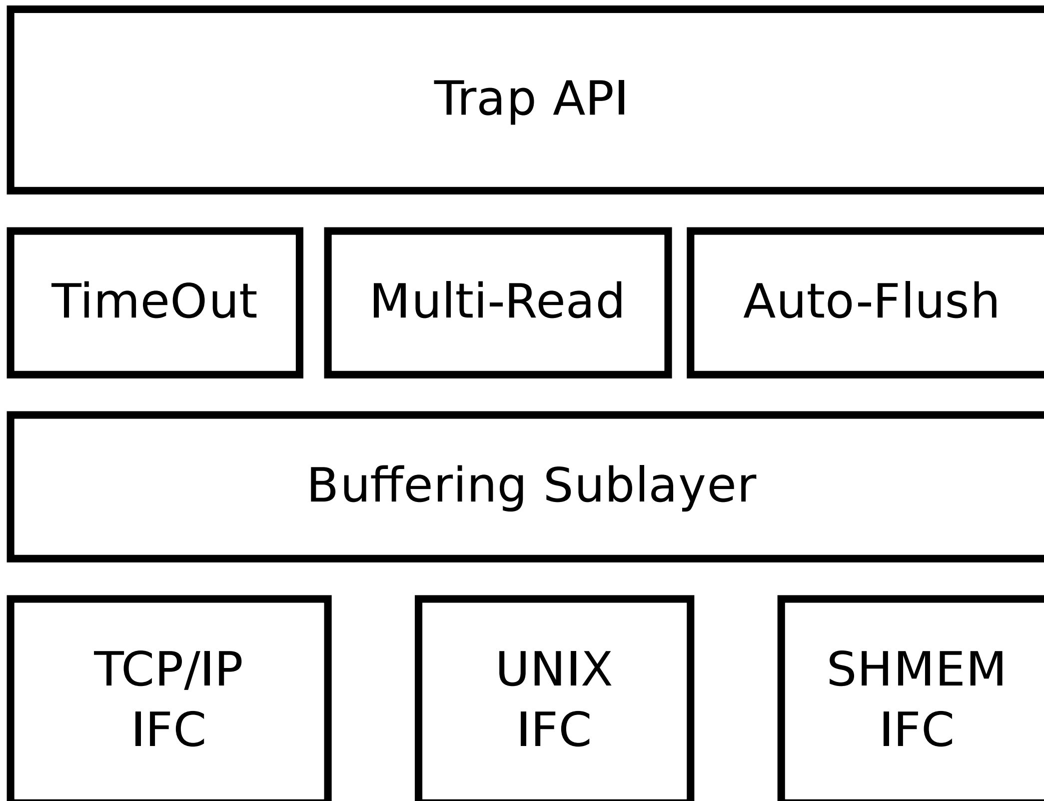

Figure 5. The architecture of the libtrap library: library supplies API easy to use for rapid development of new Nemea modules.

controlled data drop according to the configured behaviour. The destination module reads the data from the input in a loop but the read is blocked if no data are available (this behaviour is dependent on the configured parameters as well).

The lowest layer is composed of two interface types – a TCP socket (TCP/IP IFC) and a Unix socket (UNIX IFC). There is a Buffering sublayer that decreases the number of time-consuming transfer requests, *i.e.* it aggregates multiple transfer requests. Libtrap contains various high level functionalities such as TimeOut, Multi-Read, Auto-Flush. The layers are described in detail further on.

Libtrap supplies modules with an API to communicate via input and output communication interfaces. Current version of libtrap supports three main libtrap interface types that can be seen at the lowest row in Figure 5. Mostly used interface types are the UNIX socket (for local communication) and the TCP/IP socket (for remote communication between modules on separate machines). The last available communication interface uses the shared memory. The memory interface supports one to one communication only whereas the UNIX socket and the TCP/IP socket allows a user to connect N destination modules to a single output interface.

The types and parameters of the communication interfaces per each module are chosen by the user at the time a module is started. This way of interface initialization provides the possibility to adapt the module for the communication with other modules that are already running.

When the Nemea module is started successfully (including processing of the interface parameters), libtrap automatically establishes connections of the input in-

terfaces to the output interfaces of other modules and waits for new connections to the output interfaces. The connection is fully abstracted, the modules do not need to take care of the connection attempts. When the connection between two modules fails or if one of the module fails, libtrap tries to recover the connection automatically. The reconnection is done independently on the chosen interface type. The recovery not only tolerates failures of the physical communication channel but it enables to restart any module without inflicting the rest of the system.

The Nemea modules are meant to process high volume of data with high throughput. Therefore, the communication layer has to provide high throughput interface for the modules. The developed buffering sublayer aggregates requests, thus effectively reducing the overhead related to each transfer. The buffering increases the number of messages that can be transferred from one module to another. Based on the experiments, we set the size of the buffer to achieve optimal throughput for the size of the messages with basic network flow information.

Buffering sublayer is available for the input or output interface of any type. Buffering can be disabled by the user to minimize the latency of an urgent single message delivery. It can be useful to disable buffering, for example, for the modules that produce alert messages, *i.e.* the messages which are not very frequent but need to be delivered as soon as possible.

There is a set of feature blocks which allow the user to control the behaviour of the lower layers. The currently provided features are Auto-Flush, TimeOut handling, and Multi-Read.

Buffering sublayer supports a mechanism of automatic transmission of the buffer content after a specified interval for modules generating a lot of messages irregularly. This mechanism is called Auto-Flush. It prevents messages stored in the buffer from aging, because the buffering sublayer without Auto-Flush does not start the transmission until the buffer is full. Auto-Flush feature is supported only for the output interfaces. The time interval is configurable and Auto-Flush can be disabled by the user.

There are various requirements on time delays concerning send and receive functions. In this context, time delay means the maximal amount of time between the module calls the function and obtains the result of the operation. Libtrap has three special timeouts defined for this purposes: TRAP_WAIT, TRAP_NO_WAIT, TRAP_HALFWAIT. These defined timeouts are used to control the time the module wants to wait for the transfer. TRAP_WAIT is an equivalent of blocking system call. It waits until a message is sent by the output interface or the message is fully received by the input interface. As a result, when timeout of the interface is set to TRAP_WAIT and no connection to another module is established on that interface, it blocks the module until some other module is connected. TRAP_NO_WAIT is an opposite to TRAP_WAIT. It acts as a non-blocking function and returns error when no message was transferred. TRAP_HALFWAIT is a compromise between the two timeouts. It behaves as TRAP_WAIT, however it does not block the module when no module is connected on the other site of the connection. Besides these specific timeouts, TRAP also supports timeout interval, *i.e.* in the case of the stucked operation the function waits for the specified number of microseconds and then returns.

The last feature is called Multi-Read. It can be used in modules that utilize more than one input interface. Multi-Read allows to read messages from all interfaces enabled by a mask in parallel. The read operation is done by a set of process threads which wait for incoming data. The result of the Multi-Read operation is available in form of an array, where each element of the array contains the resulting code of the operation, the received message size and a pointer to the received message data.

3.2 Unified Record

Unified Record (UniRec) is a specific data format of messages that are sent over TRAP interfaces. The modules need to interchange various types of data and the amount of data may be very large. This implies three requirements which are reflected in UniRec:

1) The format should be flexible, *i.e.* it should allow to dynamically create new message structures.
2) It also needs to be memory efficient, *i.e.* it should achieve efficiency close to a simple C structure to minimize overhead during message transfer.
3) Manipulation with data in the message should be fast, *i.e.* an access to an element of the message should not pose a bottleneck of the module.

Each UniRec record consists of several *fields*, each field has a name and a type. The set of fields in the record is called a *template*. The template is specified by enumerating names of all its fields. A list of possible field names, together with their types and meaning are given in a global configuration file (which can be easily extended when a new field is needed). Each TRAP interface uses exactly one template, *i.e.* all messages sent over a particular interface are in the same format. Any two modules interconnected by an interface must use the same UniRec format on that common interface. The template of each interface of a module is defined during an initialization of the module. While the format of the output interface is usally given by the function of the module, many modules allow to specify the format expected on their input by command-line parameters.

The record, when stored in a memory or sent via the TRAP interface, looks similarly to a simple C structure - individual fields are stored one after another, without additional data or padding. The record consists of a *static part*, in which all fields have static size, and a *dynamic part*, in which the fields with dynamic size are stored, see Figure 6. The dynamic fields are stored at the end of the record. Offsets pointing the the ends[1] of these fields are stored at the end of the static part.

The template is (usually) specified during an initialization of the module by providing a string with names of all fields in the record. It is stored as a structure containg information about which fields are used, a table of their offsets in a record and some other auxiliary information. The table of offsets enables fast access to individual fields in the record.

[1] Offsets are stored instead of length because it allows fast access to dynamic fields regardless of the number of such fields. Offsets of the ends are used because the beginning of the first field is equal to the size of the static part, which can be precomputed (so it does not need to be stored in records), and the offset of the last dynamic field can be used to get the total length of the record.

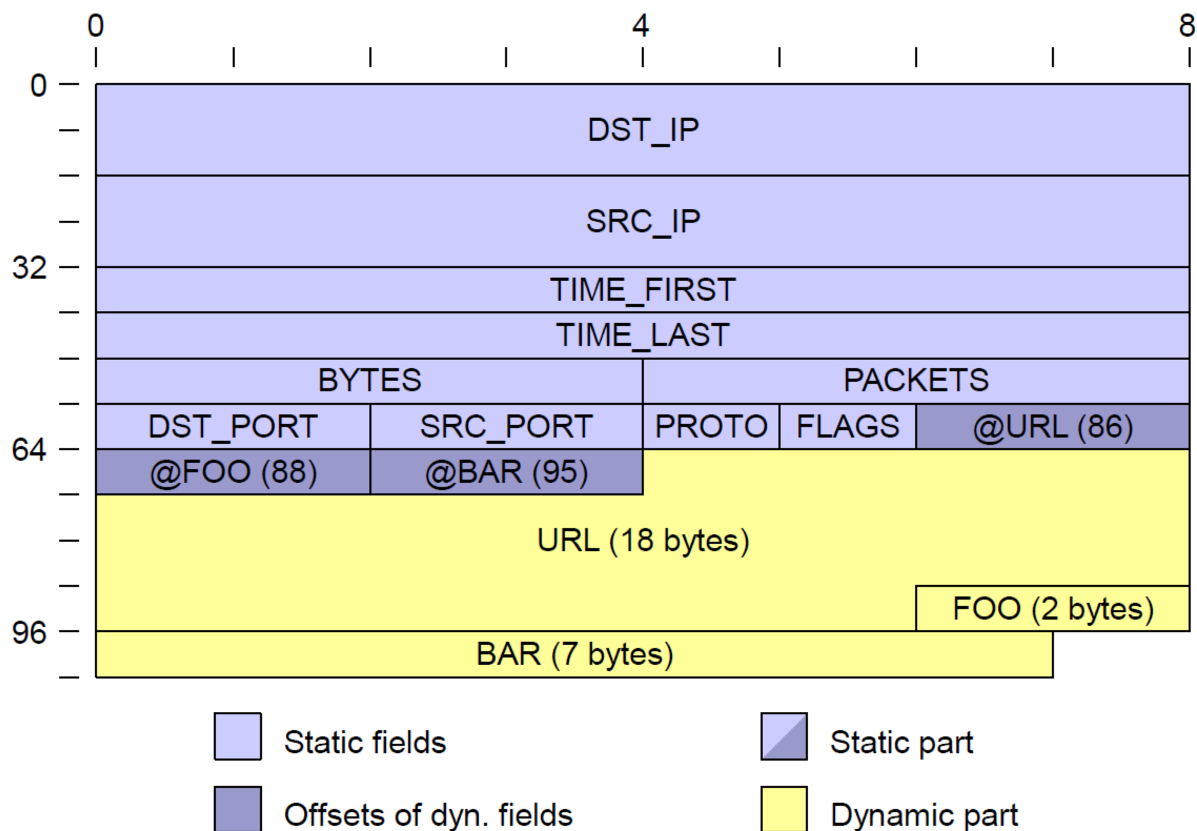

Figure 6. An example of UniRec record representing a basic IP flow record extended with several fields with the dynamic size.

An order of field names in the template specification is not important. The order of the fields in the record is always determined by the following rules:

1. First, there are the static fields, then the offsets of the dynamic fields and then the values of the dynamic fields.
2. The fields in the static part are sorted by their length in a decreasing order (so all fields are on aligned positions in memory if the beginning of the record is aligned).
3. The fields with the same length are sorted alphabetically by their name.

The fields are accessed by using simple macros, which take a pointer to the template structure, a pointer to the beginning of the record and an identifier of the field.

The access to the static field requires just one additional CPU instruction in comparison to the ordinary C structure. The additional instruction respresents the access to the small template table (easily fits into L1 cache) so the overhead is minimal. Nevertheless, the UniRec enables to create new structures (templates) at runtime and it also supports fields with dynamic size.

The UniRec fields may be of any basic type supported by the C language (signed/unsigned integers and floats of various widths), a string of chars, or one of the two special types - a type for storage of timestamps with a sub-second precision and a universal structure to handle both IPv4 and IPv6 addresses.

For performance reasons, all numeric fields in UniRec use the native endianness of a host machine.[2] IP addresses are considered to be a set of bytes rather than a number, so the addresses are stored in big endian (network order).

4 Platform Evaluation

We measured performance using two connected modules with libtrap interfaces. One module is used as a generator of messages with the given size and the second module is a receiver counting the number of received messages. All tests were performed repeatedly for the list of different message sizes. The tests are described in Section 4.1 (buffering was enabled) and Section 4.2 (buffering was disabled). The tests were performed on a single machine with following configuration: Intel(R) Xeon(R) CPU E5335 @ 2.00GHz, 10GB RAM, Scientific Linux release 6.3 (Carbon), Linux kernel 2.6.32-131.0.15.el6.x86_64 #1 SMP.

4.1 Performance of Libtrap with Buffering Enabled

The performance of commonly used libtrap interface types (TCP and UNIX IFC) is measured and depicted in Figure 7 and Figure 8. The figures display the number of transferred messages per second and the measured average transfer speed in Mbps respectively. The results for the UNIX interface are depicted in Figure 9 and Figure 10. Our goal was to optimize throughput for short messages to enable fast processing of majority of UniRec messages. A typical UniRec is of size 66 B and corresponds to a basic flow record. The results show that TRAP is able to transfer more than 7 million of these messages per second which is more than enough since the message peaks reach less than half a million.

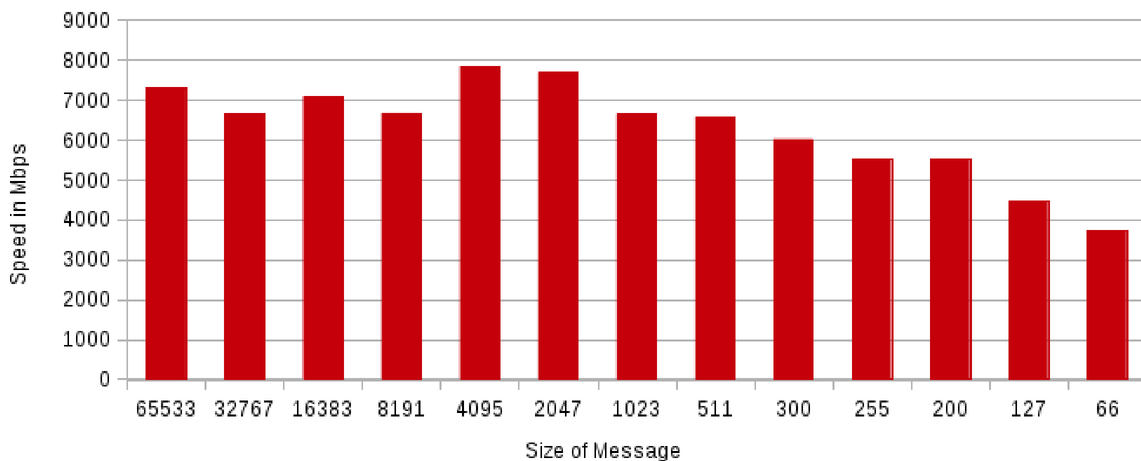

Figure 7. Throughput of the TCP libtrap interface in Mbps.

[2] In a rare case when a Nemea system would be distributed on hosts with different endianness, it is possible to implement a module which switch endianness of all relevant fields of all records.

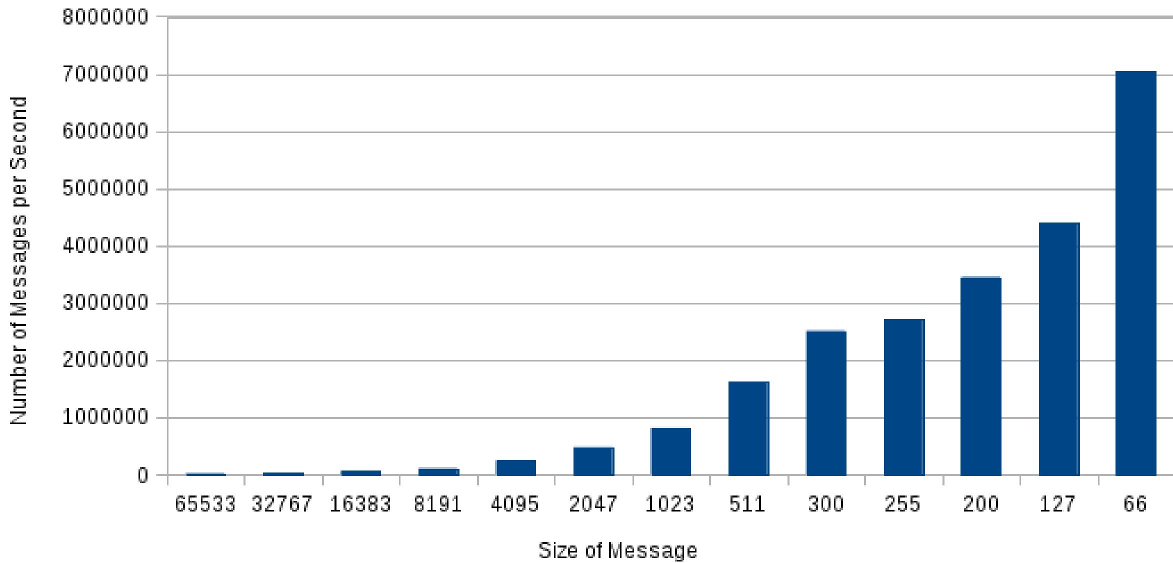

Figure 8. Number of transferred messages per second using the TCP libtrap interface.

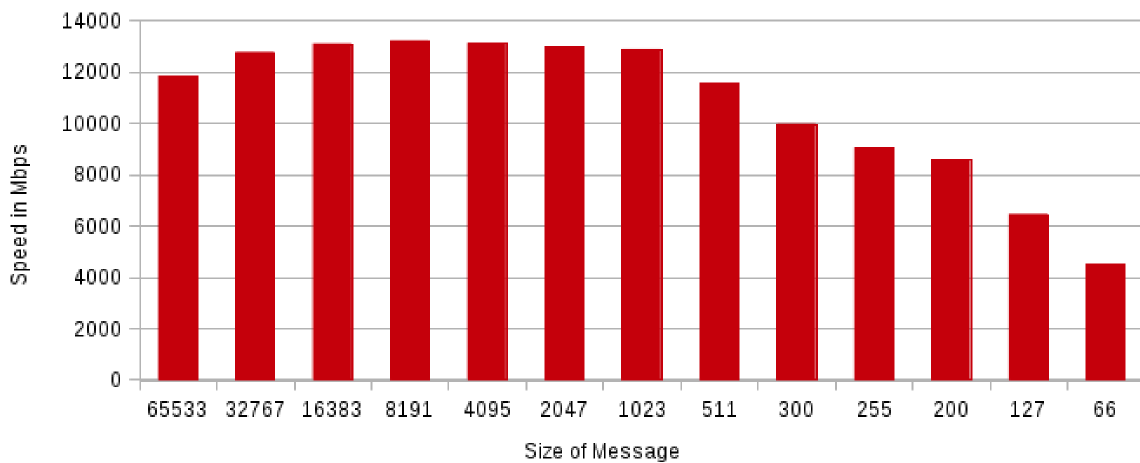

Figure 9. Throughput of the UNIX libtrap interface in Mbps.

4.2 Performance of Libtrap with Buffering Disabled

When the Buffering Sublayer is disabled, the latency of the message delivery is minimized. However, the efficiency of the libtrap library is much lower than with the buffering. The results of performance measurement with the buffering disabled are depicted in Figure 11, Figure 12, Figure 13 and Figure 14.

Although it is clear that the performance of un-buffered communication scheme must be significantly lower (due to processing and transfer overhead) it is still far behind any expected required limit because this communication scheme is typically used for exporting aggregated information about detected anomalies (which statistically represents only fraction of input messages, e.g. 10000 of input message count in worst case of massive DDoS attacks).

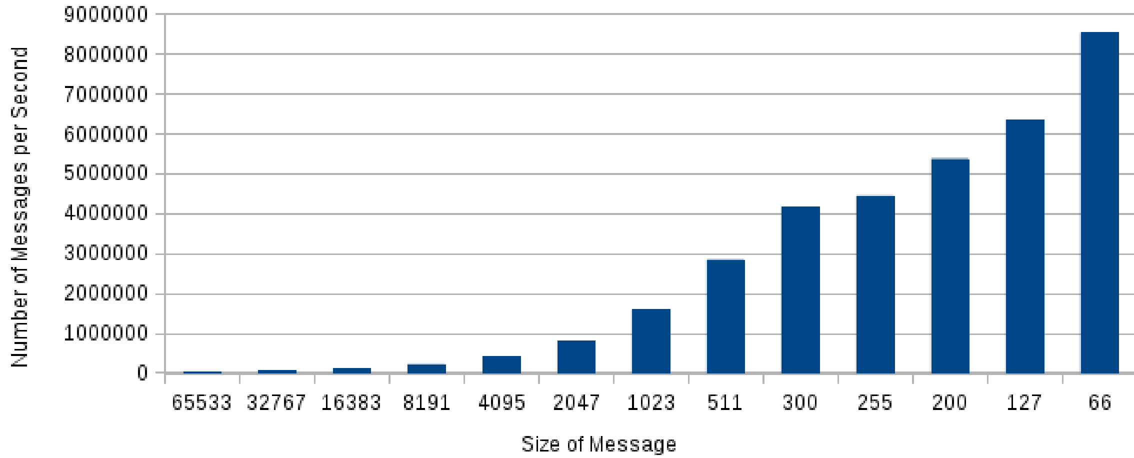

Figure 10. Number of transferred messages per second using the UNIX libtrap interface.

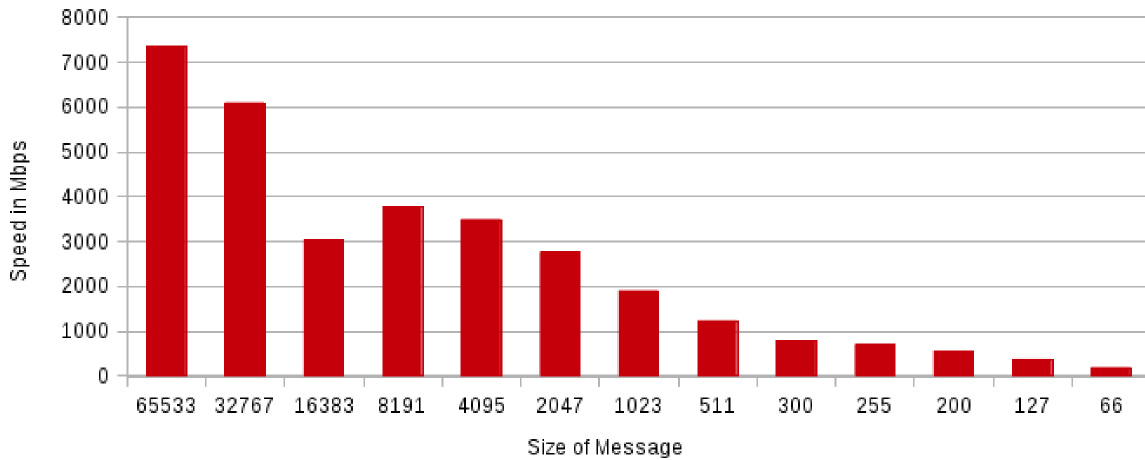

Figure 11. Throughput of the TCP libtrap interface in Mbps with Buffering Sub-layer disabled.

5 Nemea Modules

At the end of year 2013, there are more than 20 Nemea modules implemented. Many of them are released together with the framework [6]. Several other modules are currently in development. Some of the released modules are described briefly in this section. Nemea modules usually implement either basic message processing or analytical tasks.

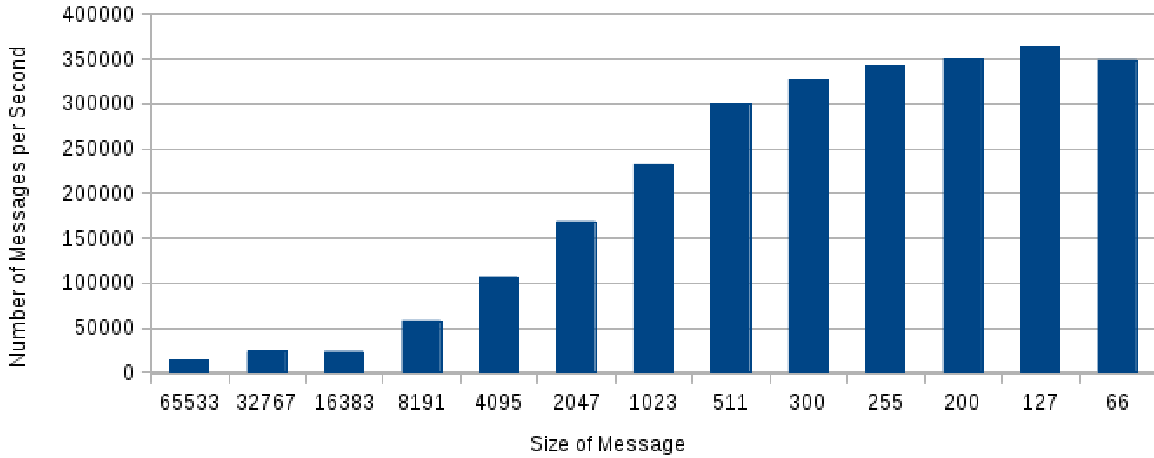

Figure 12. Number of transferred messages per second using the TCP libtrap interface with Buffering Sublayer disabled.

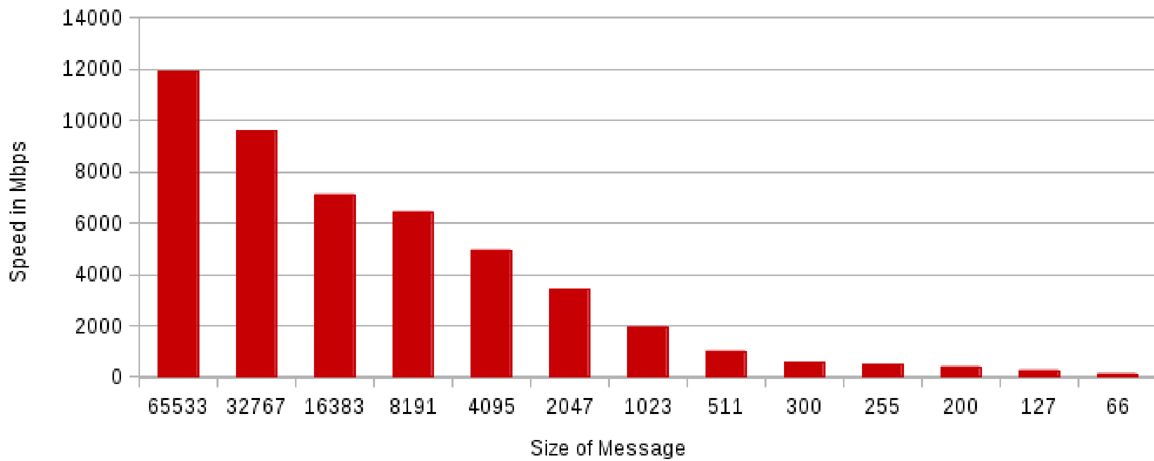

Figure 13. Throughput of the UNIX libtrap interface in Mbps with Buffering Sublayer disabled.

5.1 Processing Modules

5.1.1 Nfdump Reader

This module is one of possible data sources in the Nemea system. The module reads files with flow records stored in Nfdump format. These flow records are converted into the corresponding UniRec records and sent to the output interface. The module is able to read several files in a sequence. The records are normally read and sent as fast as possible, but they can optionally be sent in a slower pace given by a user. There is also an option to send the records in a pace following the end timestamps of the flow records and to modify the timestamps according to current time.

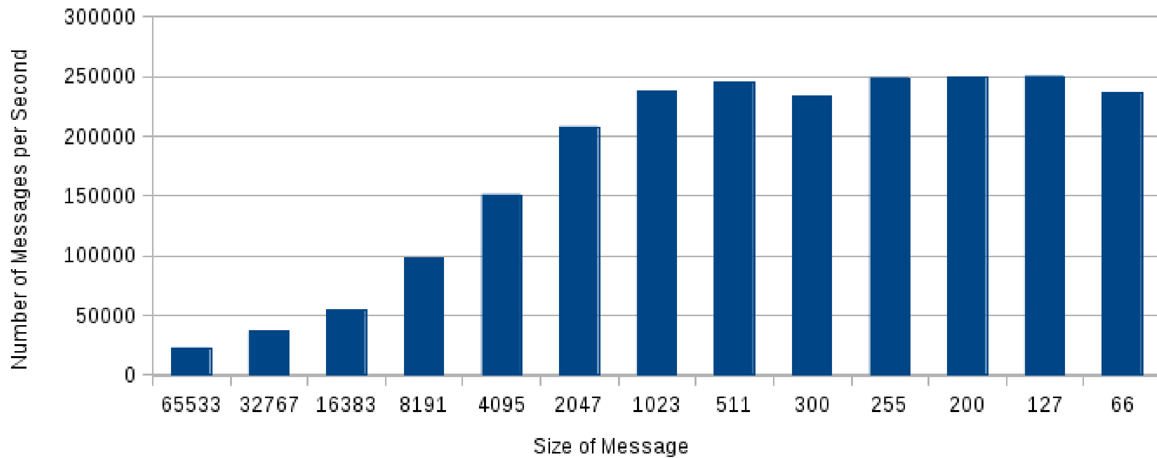

Figure 14. Number of transferred messages per second using the UNIX libtrap interface with Buffering Sublayer disabled.

5.1.2 Flow Counter

This simple module has one input interface on which it receives flow records. It counts the number of flows and the total number of packets and bytes in these flows. The counts are printed out when the module is terminated. It has also an option to print a dot every N records. It is used mostly for testing.

5.1.3 Logger

This module prints all received records to a standard output or into a file in a human readable format. The module may have any number of inputs and any input may use any UniRec format. It allows to select which fields of incoming records should be logged. By default, union of all input formats is used. This module can be used for testing or to log results of some detector, for example.

5.1.4 TrapDump/TrapReplay

These modules allow to store any stream of data sent over a TRAP interface into a file and replay it later, regardless of data format. The modules are used mostly for debugging or for testing of detectors. For example, it is possible to store some data which contain an attack, and push them into a detector module repeatedly, every time with different parameters of the tested detection algorithm, to see if and when the attack is detected.

5.1.5 Anonymizer

This module performs on-the-fly anonymization of a stream of flow records. For every flow record received on the input interface, source and destination IP addresses of the flow are anonymized and the record is passed to the output interface. Anonymization is done using CryptoPAN method with configurable key.

5.1.6 Entropy

This module computes statistics about traffic volume and entropy of packet header fields per time interval. By default, number of flows, packets and bytes and entropy of source and destination addresses and ports are computed for every 5 minute interval. Optionally, entropy of combinations of various header fields may be computed as well. If information about a link (or more generally about an observation point) at which a flow was captured is available, the statistics are computed separately for every link. The output of this module is used as an input of various anomaly detectors based on time-series analysis.

5.1.7 Transit Filter

This module detects transit traffic, *i.e.* flows which do not originate or end in the monitored network. Of course, such module is meaningful only in networks with more than one entry point. The module needs a list of all IP address ranges inside the monitored network. It has two outputs, it forwards non-transit traffic on the first one and transit traffic on the second one.

5.1.8 Plugin for IPFIXcol - IPFIX2UniRec

We utilize plugin into our IPFIX collector to feed our Nemea system with real-life data collected from CESNET metering points. IPFIX2UniRec translates IPFIX records to the UniRec and sends them on the output interfaces. Each output interface delivers a different UniRec template to allow for various level of detail. A basic UniRec template describes the flow record up to the transport layer. Whereas application-level templates describes the flow records belonging to HTTP and DNS protocols. This plugin is used as the main source of data in the Nemea system.

5.1.9 Email Reporter

This module converts incoming UniRec records into email messages using a template and send them to an SMTP server. The template is a text of email body with references to UniRec fields which are replaced by corresponding values of the incoming records. This module can be used for reporting outputs of detection modules to a human operator.

All previous modules are implemented in C or C++. Email Reporter is an example of a module implemented in Python using a Python wrapper around libtrap. Writing modules in Python is much easier, but such module has much lower performance so it is not appropriate for modules which have to process large amounts of data.

5.2 Analytical Modules

5.2.1 PCA

This module detects anomalies in time series of traffic volume and/or entropy of traffic features on different links. The detection method is based on Principal Component Analysis and is derived from a well known method proposed by Lakhina *et. al.* [2]. For every detected anomaly a record describing the anomaly is sent to the module output. The record contains a timestamp the anomaly happened, a link identifier and dimension, (*i.e.* volume metric or entropy field) the anomaly occurred on. Suitable input data can be obtained using entropy module described above.

5.2.2 Astute

This module performs ASTUTE detection method as described in [3]. The method looks for correlated changes in flow volumes between two consecutive time windows. It computes a so called Astute Assessment Value. This value should be close to zero in normal traffic but it deviates from zero significantly when a lot of flows changes its volume or (dis)appears at the same time. The modules output this value for various levels of flow aggregation at the end of each time window (5 minutes by default). It is responsibility of a subsequent module to threshold these values and report anomalies.

5.2.3 IP Spoofing Detector

The module detects spoofed source IP addresses using a method described in [5]. The method is designed for large networks with several entry points. The method learns which IP prefixes usually comes through which entry points. It reports spoofing if there are too many flows which comes into the network through unusual entry points. It also look for so called bogon addresses - the addresses which should not exist in public Internet, such as addresses from private or reserved ranges and ranges which has not been allocated yet. The module needs a list of all IP address ranges inside the monitored network and a list of bogon ranges. Also, the input should not contain transit traffic. This can be achieved using the Transit Filter module described above.

5.2.4 Blacklist Filters

This is a set of modules which allow to filter flow records using given blacklists. IP addresses in every incoming flow record are compared to one or more blacklists and if a match is found, number of the matching blacklist is added to the record and the record is sent to output. Blacklist of domain names or URLs may be used instead of IP addresses if flow records contain information extracted from DNS or HTTP protocol.

5.2.5 Botnet Detector

The module implements our method for detecting active botnets. Detection is based on observing communication of the potential bots and calculating the probability value. The module receives records from blacklist detector. The blacklist detector marks all records that contain IP addresses of a blacklisted Command & Control server. To minimize false alarms the algorithm observes if the marked communication complies with the characteristics of active bot and C&C server communication. As a result the module outputs IP address together with the probability of the address being a bot.

5.2.6 HostStats

HostStatsNemea module calculates statistics about network traffic of each individual host (IP address) in the network. The statistics are searched for suspicious behaviour using a simple ruleset. This suspicious behaviour is stored in a log file. Currently there are rules for detection of hosts scanning network and for detection of DoS attackers and victims.

Several other modules, mostly for detection of various types of attacks and anomalies, are in development. Also, a module for smart aggregation of detection results is being developed. This module should receive results of all detectors, compare them between each other and with history, evaluate credibility and importance of each event and report selected ones to a human operator.

6 Conclusion

This technical report summarizes our work on the network traffic analysis. Our effort results in implementation and deployment of two frameworks, TRAP and Nemea. TRAP is utilized as a mean to transfer heterogeneous data in a streaming way among processes instantiated in Nemea. Our practical experience shows that this concept is viable and serves well for on-the-fly processing of data collected from a large network. Nemea constitutes various modules for processing of the streaming data as well as the data analysis. Some of the modules are ready-to-use and have been deployed to analyze real-life network issues with promising results. Other modules are in various phases of development.

Our future work includes developing more modules to address the detection of various attacks and anomalies. The goal is to aggregate and correlate the detected events to verify the attack as well as to monitor the attack history. We also plan to elaborate further the correlation of the behavioural analysis with available blacklists. The detection results will be fed to Warden [7], CESNET early-warning system. An extra effort will be focused on the analysis of data extracted from the application layer. A supervisor is being developed to manage running Nemea modules.

References

[1] Václav Bartoš, Pavel Čeleda, Tomáš Kreuzwieser, Viktor Puš, Petr Velan, Martin Žádník: Pilot Deployment of Metering Points at CESNET Border Links, CESNET technical report 5/2012.

[2] Anukool Lakhina, Mark Crovella, Christophe Diot: Mining anomalies using traffic feature distributions. In: ACM SIGCOMM Computer Communication Review, vol. 35, no. 4, 2005.

[3] Fernando Silveira, Christophe Diot, Nina Taft, Ramesh Govindan: ASTUTE: Detecting a Different Class of Traffic Anomalies. In: ACM SIGCOMM Computer Communication Review, vol. 40, no. 4, 2010.

[4] Petr Velan, Radek Krejčí: Flow Information Storage Assessment Using IPFIXcol. In: Dependable Networks and Services, Lecture Notes in Computer Science, 2012.

[5] Michal Kováčik, Michal Kajan, Martin Žádník: Detecting IP-spoofing by modelling history of IP address entry points. In: Emerging Management Mechanisms for the Future Internet, Barcelona, ES, Springer, 2013.

[6] Nemea framework. *http://www.liberouter.org/nemea/*

[7] Warden. *https://csirt.cesnet.cz/Warden/The%20Project*