MMT User Manual

Montimage

Edgardo Montes de Oca
contact@montimage.com

MMT-DPI version 1.7.0
MMT-Probe version 1.4.4
MMT-Security version 1.2.11
MMT-Operator version 1.7.2

29/03/2021

This user manual presents and explains how to use Montimage monitoring tool (**MMT**) developed by Montimage's research team. The manual is suitable for both basic users to use MMT or developers to build their specific applications.

For more information please contact us at `contact@montimage.com`.

Contents

1 Introduction

Montimage Monitoring Tool (MMT) is a monitoring solution that combines: data capture; filtering and storage; events extraction and statistics collection[1]; and, traffic analysis and reporting. It provides network, application, flow and user level visibility. Through its real-time and historical views, MMT facilitates network security and performance monitoring and operation troubleshooting. MMT-Security engine can correlate network and application events in order to detect operational, security and performance incidents.

This document is divided into 6 sections:

- Section 1 includes this introduction, the tool usage context, a description of the tool's global architecture and its main features.
- Section 2 explains how to install MMT.
- Section 3 explains how to configure MMT.
- Section 4 explains how to use MMT-Security.
- Section 5 is dedicated to developers who want to use the libraries for building other applications with specific objectives.
- Section 6 gives a roadmap for future development of the tool.

1.1 Network and Business Activity Monitoring

Scope: New and more critical vulnerabilities are constantly being introduced by the evolution of the Internet and mobile communications where more and more critical infrastructures are becoming open and corporate IT is being de-materialized (e.g., using cloud services). This is pushing towards the need for more proactive and automated mechanisms for detecting and preventing anomalies due to attacks or misbehaviour.

In this context, Deep Packet Inspection (DPI) is considered as a key element in the shift towards advanced monitoring. DPI is the process of capturing network traffic, analysing and inspecting it in detail to determine accurately what is really happening in the network and applications that communicate. This technology feeds the different monitoring applications with high added value information.

Requirements: Starting from this perception, the requirements of a network/application monitoring system can be summarized as follows:

1. *High capturing performance.* It must be able to capture traffic at high speeds and under high traffic volume. This depends on what is to be monitored and where.
2. *Extensibility.* If new services are integrated in the network, it must be possible to deploy effortlessly new monitoring mechanisms for these specific services. In addition, if new analysis techniques are needed, it should be possible to integrate them as effortlessly as possible.
3. *Scalability.* It must be able to handle the increase of traffic data that needs to be analysed without performance degradation; due, for instance, to the increase of network link speeds and the number

[1] You can try MMT without any installation at `http://mmt-cloud.montimage.com`

of probes in the network. Scalability can be achieved by reducing the traffic information collected using efficient packet capturing mechanisms, load-balancing and traffic pre-processing.

4. *Near real time functioning.* It must implement near real time mechanisms in order to quickly detect network security/performance problems and allow timely execution of automated or manual countermeasures.
5. *Granularity.* It must be able to track the security and performance of each service by capturing and analysing the traffic belonging to the application of interest.
6. *Diversity.* It must support the network's diversity, taking into account different types of network devices from multiple vendors, protocols stacks, and applications providing services to the users.
7. *Low cost.* It should not use excessive amount of computing, storage, and communication resources so the cost of deploying and operating the monitoring infrastructure remains low for service providers.
8. *Secure.* It should not add vulnerabilities to the network, or disturb normal network operation.

1.2 MMT Architecture

Global architecture: A high level architecture of the MMT's frameworks is represented by Figure 1.

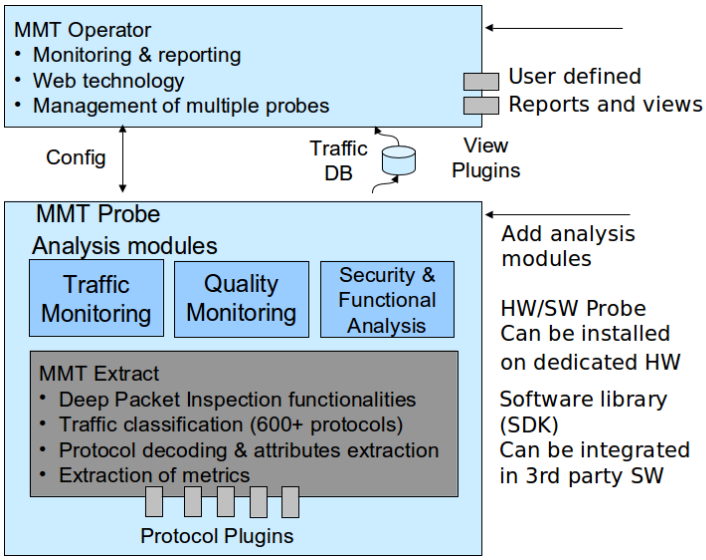

Figure 1: MMT Architecture

The integrated MMT tool is composed of three complementary, but independent, modules as depicted in this figure.

MMT-DPI is the core packet[2] processing module. It is a C library that analyses network traffic using Deep Packet/Flow Inspection (DPI/DFI)

[2] Note that the terms: "packet", "message", "log entry" and "event" can be considered as equivalent in this document. Here we will use either the term "packet" or "event" to denote an event in time that is perceived as structured data that can be extracted, classified and analysed.

techniques in order to extract hundreds of network and application based events, including: protocols field values, data in messages and logs, network and application QoS parameters and KPIs. MMT-DPI incorporates a plugin architecture for the addition of new protocols and data structures; and, a public API for integrating third party probes (more details given in Section 5).

MMT-Security is a security analysis engine based on MMT-Security properties described in this document. MMT-Security analyses and correlates network and application events to detect operational and security incidents. For each occurrence of a security property, MMT-Security allows to detect whether it was respected or violated. The MMT-Security engine has been fully implemented in the context of Montimage's research projects. It is the result of previous research work in the network monitoring field and relies on the multi-domain security requirements identified in the context of diverse case studies (more details given in section 4.4).

MMT-Probe is a main application that uses MMT-DPI and MMT-Security. It captures packets, makes them available to MMT-DPI and MMT-Security, and receives resulting values (extracted meta data or statistics) to be used for creating reports forwarded to MMT-Operator.

MMT-Operator is a visualization Web application. It allows collecting and aggregating meta data and reports provided by MMT-Probe, and presents them via a graphical user interface (e.g., alarms, line charts). MMT-Operator is customisable: the user is able to define new views or customise the large list of predefined ones. With its generic connector, MMT-Operator can also be integrated with third party traffic probes.

1.3 Features

Main features: The main features provided by MMT are:

1. *Granular traffic analysis capabilities*: MMT allows parsing a wide range of protocols and applications and to extract various network and application based traffic performance indicators. The extraction is powered by a plugin architecture for the addition of new protocols and applications.
2. *Application classification*: Prior to extracting protocol or application attributes, MMT uses DPI techniques for application identification and classification. This is essential when analysing applications that use non-standard port numbers (e.g., P2P, Skype).
3. *Rule engine*: MMT-Security introduces a rule engine that allows the detection of complex sequence of events that conventional monitoring does not detect. This is used to monitor: i) access control policies (e.g., that authorized users are authenticated prior to using a critical business application); ii) anomaly or attacks (e.g., excessive login attempts on the application server); iii) performance (e.g., identification of VoIP calls with QoS parameters exceeding acceptable quality thresholds); and, much more.
4. *Configurable reports*: MMT traffic reports and charts can be configured by the user. The user can edit pre-configured reports and create new ones. Different chart types and graphs can be used including: pie, histograms, time charts, stacked area charts, sequence charts, tables, hierarchical tables, etc.

5. *Multi-platform solution*: MMT is available and tested on Linux based distributions but portable on Windows. It can be installed as software on commodity hardware or optimized for integration with dedicated probes.
6. *Modular solution*: MMT is a modular solution composed of four components: MMT-DPI library for the traffic processing and data decoding; MMT-Security engine for property analysis; MMT-Probe application for integrating MMT-DPI and MMT-Security; and MMT-Operator for data aggregation, correlation, reporting, and distributed probe management. It is possible to integrate MMT-DPI and MMT-Security in third party traffic probes and to connect MMT-Operator with existing monitoring systems.

2 Installation

2.1 Pre-Requirement

2.1.1 Hardware

MMT can run easily on a (personal) computer to process low network traffic, i. e., less than 100 Mbps (mega bit per second). The minimal requirement is that the computer has at least 2 cores of CPU, 2 GB of RAM and 4GB of free hard drive disk. The computer must possess also one NIC (Network Interface Cards) on which MMT captures network traffic. Higher network bandwidth requires stronger computer.

To be able to exploit fully capacity of MMT to process very high network bandwidth, one need suitable hardware configurations. MMT can be deployed on a single server or split on separate servers. This section specifies the hardware requirements for the former. For those of the latter, please contact us.

Network Interface Cards: MMT needs at least 2 NICs: one for capturing network traffic and another for administrating. If the probe is to be an active network element (i.e., receives, processes and re-transmits the communication packets) then at least 3 NICs are necessary.

→ To achieve the best performance, the capturing NIC should be either Intel X710 or Intel X520 card. These are recommended because they support DPDK that will considerably improve the packet processing. For other hardware architectures, adaptations and tests would need to be performed.

CPU: For the best performance, the use of Intel Xeon class server system is recommended, such as Ivy Bridge, Haswell or newer. The larger the CPU cache, the better the performance obtained.

→ We recommend having at least 16 cores to process 1 Gbps of traffic. For higher bandwidths other server setups are needed, please contact us for more information.

RAM: We recommend using the fastest memory one can buy; and, having one DIMM per channel. One should avoid having 2 or more DIMMs per channel to make sure all memory channels are used to the fullest. It is more expensive to buy 2x16GB than 4x8GB, but with the later, memory access latency increases and the frequency and throughput decreases.

→ We recommend having at least 32 GB of RAM to process 1Gps traffic.

Hard Disk Drive: If MMT is to write meta data on a hard disk drive, it will do it using a database system (e.g., MongoDB) or files.

→ We recommend using a Solid State Drive with at least 20 GB of free space.

BIOS Settings: The following are some recommendations on BIOS settings. Different platforms will have different BIOS naming so the following is mainly for reference:

1. Before starting, consider resetting all BIOS settings to their default.
2. Disable all power saving options such as: Power performance tuning, CPU P-State, CPU C3 Report and CPU C6 Report.
3. Select Performance as the CPU Power and Performance policy.

4. Disable Turbo Boost to ensure the performance scaling increases with the number of cores.
5. Set memory frequency to the highest available number, NOT auto.
6. Disable all virtualization options when you test the physical function of the NIC, and turn on `VT-d` if VFIO is required.

2.1.2 Software

Ubuntu LTS 16.04: This Ubuntu version is recommended to run MMT as MMT has been carefully tested on it. Other Ubuntu version can also run MMT. To run MMT on the other Linux distros and Windows, please contact us. The example terminal commands used in this manual to prepare MMT running environment is suitable for a machine running Ubuntu LTS 16.04.

MongoDB >= v3.6 MMT stores meta data using MongoDB. For the use of other database systems please contact us. To install MongoDB follow steps described in `https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/`. To resume, for Ubuntu 16.04 LTS, one needs to do the following:

1. Import the public key used by the package management system:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com
    :80 --recv 2930ADAE8CAF5059EE73BB4B58712A2291FA4AD5
```

2. Create a list file for MongoDB and reload the local package database:

```
echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/
    apt/ubuntu xenial/mongodb-org/3.6 multiverse" | sudo
    tee /etc/apt/sources.list.d/mongodb-org-3.6.list
sudo apt-get update
```

3. Install the MongoDB package:

```
sudo apt-get install -y mongodb-org
```

4. Start MongoDB:

```
sudo service mongod start
```

5. Verify that the `mongod` process has started successfully by checking that the log file `/var/log/mongodb/mongod.log` contains the line:

```
[initandlisten] waiting for connections on port 27017
```

NodeJS >= v8.9.1 The MMT-Operator runs using NodeJS. To install NodeJS, follow the steps described in `https://nodejs.org/en/download/package-manager/`. To resume, for Ubuntu 16.04 LTS, one needs to do the following:

1. Create a list file for NodeJS and reload the local package database:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E
    bash -
```

2. Install NodeJS:

```
sudo apt-get install -y nodejs
```

3. Verify that NodeJS was successfully installed:

```
node -v
```

Option: One might need to install the REDIS and/or KAFKA message bus servers if the MMT-Operator is to receive meta data from the MMT-Probe via this type of publish/subscribe systems.

MMT supports the use of the Data Plane Development Kit (DPDK) for high performance capturing of network traffic, e.g., until 40Gbps. To be able to use MMT with other capturing libraries, e. g., netmap, native socket, etc., please contact us.

Before running MMT with DPDK, we need to set the "huge pages" option. The commands below will reserve 16 GB for MMT-Probe. Note that this configuration needs to be done after each reboot. For doing it permanently one needs to modify the arguments of the kernel command line.

```
# Each huge page has 2 MB so we need 8192 pages.
# For a single-node system:
echo 8192 > /sys/kernel/mm/hugepages/hugepages-2048kB/
    nr_hugepages

# For a NUMA system:
echo 4096 > /sys/devices/system/node/node0/hugepages/
    hugepages-2048kB/nr_hugepages
echo 4096 > /sys/devices/system/node/node1/hugepages/
    hugepages-2048kB/nr_hugepages
```

For this, follow the steps described in `http://dpdk.org/doc/guides/linux_gsg/build_dpdk.html` to install DPDK. To resume, one needs to do the following:

1. Install compiler tools:

   ```
   sudo apt-get update
   sudo apt-get install build-essential gcc make python3
   ```

2. Download DPDK (`http://fast.dpdk.org/rel/dpdk-16.11.1.tar.xz`):

   ```
   wget http://fast.dpdk.org/rel/dpdk-16.11.1.tar.xz
   ```

3. Decompress the archive and go to the uncompressed DPDK source directory:

   ```
   tar xJf dpdk-16.11.1.tar.xz
   cd dpdk
   ```

4. Install DPDK:

   ```
   make install T=x86_64-native-linuxapp-gcc
   ```

5. Load the DPDK driver:

   ```
   sudo modprobe uio
   sudo insmod x86_64-native-linuxapp-gcc/kmod/igb_uio.ko
   ```

6. Bind DPDK to the capturing NIC:

   ```
   # list the status of all NICs on the server:
   sudo python3 tools/dpdk-devbind.py --status
   # bind capturing NIC eth1 to the igb_uio driver:
   sudo python3 tools/dpdk-devbind.py --bind=igb_uio eth1
   ```

2.2 Installation

Installing MMT: To install MMT do the following steps:

1. Install MMT:

```
# Install MMT by order:
# sudo dpkg -i mmt-dpi-<version>.deb
# sudo dpkg -i mmt-security-<version>.deb
# sudo dpkg -i mmt-probe-<version>.deb
# sudo dpkg -i mmt-operator-<version>.deb
# For example:
sudo dpkg -i mmt-dpi_1.6.7.3_Linux_x86_64.deb
sudo dpkg -i mmt-security_1.1.2_7d30208_Linux_x86_64.deb
sudo dpkg -i mmt-probe_1.2.0_7a73b6b_Linux_x86_64.deb
sudo dpkg -i mmt-operator_1.6.2_aa3d383.deb
# Update library path
sudo ldconfig
```

2. Verify that MMT was correctly installed:

```
ls -R /opt/mmt
#=> there must be 6 folders: dpi, examples, operator,
    plugins, probe, security
```

3. Refer to Section 3 to configure MMT as need. The configuration files of MMT-Operator and MMT-Probe are located at `/opt/mmt/operator/config.json` and `/opt/mmt/probe/mmt-probe.conf` respectively.

   For example to use 2 threads of MMT-Probe to capture traffic, one should update in `/opt/mmt/probe/mmt-probe.conf` the following parameters:

```
# Enable output results to files
file-output {
    enable = true
    ...
}
# Using 2 threads of MMT-Probe
thread-nb    = 2
# Enable MMT-Security
security {
    enable = true
    ...
}
session-report {
    enable = true
    ...
}
```

Start MMT: Execute the following commands to launch the MMT-Operator and the MMT-Probe.

```
# Start MMT-Operator
sudo mmt-operator

# Start MMT-Probe to monitor traffic on NIC eth0
# run this command on a new terminal
sudo mmt-probe -i eth0

# they can be executed as system services, e.g.,
# sudo service mmt-operator start
# sudo service mmt-probe start
```

View MMT graphical Statistics: Open a Web browser, then goto to address `http://IP_of_server`, in which, `IP_of_server` is IP of administrator NIC of the server installing MMT-Operator.

Log in to MMT-Operator Web interface on the browser using `admin`/`mmt2nm` as username/password. We recommend changing this default password once logged in.

Uninstalling MMT: Execute the following commands to completely remove MMT:

```
sudo dpkg -r mmt-operator mmt-probe mmt-security mmt-sdk
# Do the following for completely removing all of MMT's
    execution logs as well as their reports output:
sudo rm -rf /opt/mmt
```

3 Configuration

3.1 MMT-Probe

MMT-Probe requires a configuration file for setting different options. It can operate in two modes PCAP and DPDK. The PCAP mode uses libpcap library, whereas, the DPDK mode uses dpdk library, for packet capturing.

The execution log of Probe is stored in `syslog` using id `mmt-probe`. The log can be viewed using `journalctl -t mmt-probe`.

3.1.1 Configuration File

The different available configuration options are given to Probe via a configuration file. By default, Probe will try to load the configuration from `./mmt-probe.conf`, or, `/opt/mmt/probe/mmt-probe.conf` by order of priority. A configuration file can be given to Probe by using `-c` parameter, for example:

```
mmt-probe -c /home/tata/probe.conf
```

→ Note: An attribute in the configuration file can be overrided by `-X` parameter. Multiple `-X` parameters are accepted. For example, the following command will override `probe-id` and `source` attribute of `input` block to the coresponding values given after `=` sign.

```
mmt-probe -c /home/tata/probe.conf -Xprobe-id=10 -Xinput.
    source=/tmp/a.pcap
```

To list the attributes that can be overriden, run:

```
mmt-probe -x
```

A comment line inside a configuration starts by `#` sign.

The options are listed in the following:

`probe-id`: The `probe-id` indicates the identifier of the MMT-Probe. All output report formats contain this identifier. Its value is an integer of 32 bits.

`license`: The `license` indicates the path to a file where the license information is present.

`stack-type`: The `stack-type` indicates protocol suite being used. For now, 1 to monitor Ethernet-based networks, 800 to monitor IEEE802154-based networks, 624 to analyse Linux-cooked-capture pcap files.

`enable-proto-without-session-report`: The option indicates whether Probe performs the statistics of the protocols that do not belong to any IP session, such as, ARP, PPP, etc.

`enable-ip-fragmentation-report`: The option indicates whether Probe performs the statistics of fragmentation of IP packets.

`enable-ip-defragmentation`: The option indicates whether Probe defragments IP segments before performing any statistics.

`enable-tcp-reassembly`: The option indicates whether Probe reassemble TCP segments before performing any statistics.

`stats-period`: The option indicates the period of sampling in seconds. That is Probe must perform statistics every x seconds.

`dyamic-config`: The option indicates whether Probe can be updated its configuration at

runtime. If it is enabled, Probe will open an UNIX socket at the given `descriptor` to receive the new configuration parameters.

`thread-nb`: The option indicates the number of threads the Probe will use for processing packets. It must be a positive number. Use 0 to have only one thread to read then analyze packets. Use x to have one thread to read packets and x threads to analyze the packets.

`thread-queue`: The option indicates the maximum number of packets that can be queued between the reading thread and an analyzing thread, thus, total number of packets will be enqueued are `(thread-nb * thread-queues)`. It has effect only when `thread-nb > 0`.

In case of `input.mode=ONLINE`, If a packet is dispatched for a thread having a full queue, then the packet will be dropped.

`input`: This block configures the input of MMT-Probe.

- `mode` can be either `ONLINE` or `OFFLINE` to indicate that MMT-Probe will analyze respectively either traffic in realtime from a NIC or the traffic being stocked in a pcap file. For DPDK, it accepts online `ONLINE`
- `source` indicates the source of traffic to be analyze
- `snap-len` indicates maximal size of an IP packet, by default 65355 bytes.
- `dpdk-option` reserves only for DPDK mode.

The options `mode` and `source` need to be specified according to the requirements.

- `mode = ONLINE` allows *near* real-time analysis of network traffic. In PCAP mode, the NIC's network interface name needs to be identified, whereas in DPDK mode, the interface port number needs to be identified using the `source` option. For example:

```
input{
    mode = ONLINE
    # For PCAP it is interface name
    source = "eth0"
    # For DPDK it is  interface port number
    # source = "0"
}
```

- `mode = OFFLINE` allows analysis of a PCAP trace file. The `source` identifies the name of the trace file. The offline analysis is only available for the PCAP mode. For example:

```
input{
    mode    = OFFLINE
    source = "/home/tata/wow.PCAP"
}
```

`output`: This block configures general output parameters.

- `format` is either `JSON` or `CSV` to indicate format of the reports.
- Probe maintains a cache of reports to send them by burst tot output channels, such as, to files, to mongoDB, etc. The cache will be flushed when its size is over than `cache-max` or periodically designed by `cache-period`.

- In case of output to files, `cache-period=5` indicates that a new file will be created each 5 seconds if `file-output.samle-file = true`

`file-output`: This block indicates outputting reports to files.

- `enable`: set to `false` to disable, `true` to enable
- `file-name`: name of output files
- `output-dir`: path to the folder where output files are written
- `sample-file` is either `true` or `false` to output to multiple sample files or a single file. If `true` then a new sample file is created each x seconds given by `output.cache-period`
- `retain-files` indicates number of the latest sample files to be retained if `sample-file = true`. Set to 0 to retain all files. Not that the value of `retain-files` must be greater than `(thread-nb + 1)`

For example:

```
file-output {
    enable       = true
    output-file  = "data.csv"
    output-dir   = "/opt/mmt/probe/result/report/online"
    sample-file  = true
    retain-files = 80
}
```

The names of report files are in format `<timestamp>_<thread-id>_<name>`, for example, `1552302610.832958_0_data.csv`, in which:

- `timestamp` is the moment, in second.microsecond format, of creating the file
- `thread-id` is the id of the analyzing thread that generates the file, starting from 0 to `thread-nb`. Thread 0 reports status of MMT-Probe, such as, starting time, license information, CPU and memory usages, etc. Threads from 1 to `thread-nb` report analysis of packets.
- `name` is given by `file-output.output-file`

Once Probe finished writing reports to a file, it will create a semaphore that is an empty file having the same name with the report file but ending by `.sem`, for example, `1552302610.832958_0_data.csv.sem`.

`redis-output`: This configuration bloc indicates that Probe should use REDIS to publish the output. The `hostname` and the `port` indicate the address, that can be also an IP, and the port of the redis-server respectively. The `channel` indicates the name of channel to which the reports will be published. The redis-server needs to be started beforehand.

For example:

```
redis-output
{
    enabled  = true
    hostname = "localhost"
    port     = 6379
    channel  = "report"
}
```

`kafka-output`: This block indicates that Probe should output reports to a topic of kafka bus. The kafka-server needs to be started beforehand. For example:

```
kafka-output
{
    enabled  = true
    hostname = "localhost"
    port     = 9092
    topic    = "report"
}
```

`mongo-output:` This block indicates that Probe should output reports to a collection of mongo database. The database server needs to be started beforehand. For example:

```
mongodb-output {
    enable     = true
    hostname   = "localhost"
    port       = 27017
    database   = "mmt-data"
    collection = "reports" # name of collection to store
        reports

    # limit size (megabytes = 1000*1000 bytes) of the
        collection
    # - set 0 to unlimit
    # - if size of the collection reaches the limit, the
        oldest reports will be removed to maintain the limit
    limit-size  = 0
}
```

`socket-output:` This block indicates that Probe should output report to either a UNIX or internet sockets or both. The socket server must be started beforehand.

The value of `socket-output.type` can be either `UNIX` for UNIX domain socket type, or `INTERNET` for internet domain socket type, or `BOTH` for using both of two types above.

For example:

```
socket-output
{
    enable = true
    type = BOTH
    descriptor = "/tmp/mmt-probe-output.sock" # required for
        UNIX domain.
    port = 5000             # Required for Internet domain.
    hostname = "127.0.0.1" # required for Internet domain.
}
```

`dump-pcap:` This block indicates that Probe should dump the analyzed packets to files. For example:

```
dump-pcap
{
    enable       = true
    output-dir   = "/opt/mmt/probe/pcap/"

    # List of protocols, separated by comma, must appear in a
         packet which will be dumped
    protocols    = {"unknown", "http"}
    period       = 60    # new pcap file is created every x
        seconds. 0 means default value 3600 seconds
    retain-files = 50    # retains the last x files,
    snap-len     = 65355
}
```

`security`: The security output configuration blocs allow specifying the security reporting.

- `thread-nb` indicates the number of security threads for each analyzing thread of Probe.

  For example, if we have 16 Probe analyzing threads and `security.thread-nb = x` , then `(x*16)` security threads will be used .

  If set to zero this means that the security analysis will be performed in the analyzing threads of the probe .

- `exclude-rules` indicates the range of rules to be excluded from the verification.

  The range of rules is in BNF format: `exclude-rules = "x,y-z"`, in which `x,y, z` are positive numbers.

  For example, `exclude-rules = "1,3-5,7,50-100"` will exclude the rules having id $1, 3, 4, 5, 7, 50, 51, \ldots, 100$.

- `rules-mask` indicates the range of rules should be distributed to a specific security thread.

  By default, the rules will be distributed increasingly to each thread. For example, given five security rules having id 1, 5, 6, 7, 8 and two security threads, then, the first thread will analyzes rules 1, 5 while the second one analyzes rules 6, 7, 8. This can be represented by:

  ```
  rules-mask = "(1:1,5)(2:6-8)"
  ```

  Generally, the `rules-mask` uses the following BNF format:

  ```
  rules-mask = (thread-index:rule-range)
  ```

  in which, `thread-index` is a positive integer; `rule-range` is either a positive integer, or a range of numbers (see `exclude-rules`).

  For example, if we have `thread-nb = 3` and `rules-mask = "(1:1,4-6)(2:3)"`, then, thread 1 verifies rules 1,4,5,6; thread 2 verifies only rule 3; and thread 3 verifies the rest.

→ Note: if we have `thread-nb = 2` and `rules-mask = "(1:1)(2:3)"`, then only rules 1 and 3 are verified, the other rules are not.

- `output-channel` indicates the destinations of reports. It can be `redis`, `kafka`, `file`, `mongodb`, or, `socket` to write reports to a redis channel, a kafka topic, files, a mongodb collection, or socket respectively.

  We can combination of these channels to output reports to different channels, for example, `output-channel = {file, mongodb, socket}`.

  If nothing is specified, the default value is `file`.

→ Note: The reports can be output a channel if the channel is enabled. For example, the security reports will be written to file if and only if we have `security.output-channel = file` and `file-output.enable = true`.

- `report-rule-description` indicates whether include rule's description into the alert reports. If set to `false` , the descriptions will be an empty string in the reports. Excluding descriptions will reduce the size of reports.

- `ignore-remain-flow` indicates whether ignore the security verification on the rest of an IP flow when an alert has been detected.

- The 3 following parameters are specific for MMT-Security to override its default configuration:
  - `input.max-message-size = 60000`: size of a message, in bytes, to encapsulate data sending from MMT-Probe to MMT-Security
  - `security.max-instances = 100000`: maximum number of instances of a rule
  - `security.smp.ring-size = 1000`: maximum number of messages that will be stored in a ring buffer
- `ip-encapsulation-index` this option selects which IP layer will be analyzed in case there exist IP encapsulation. Its value can be one of the followings:
  - `FIRST`: first IP in the protocol hierarchy
  - `LAST`: last IP in the protocol hierarchy
  - i: i^{th} IP in the protocol hierarchy.

  For example, given ETH.IP.UDP.GTP.IP.TCP.VPN.IP.SSL,
  - `FIRST`, or 1, indicates IP after ETH and before UDP
  - `LAST`, or any number $>= 3$, indicates IP after VPN
  - 2 indicates IP after GTP and before TCP

For example,

```
security
{
    enable = true
    thread-nb       = 1
    exclude-rules   = "(50-100)"
    rules-mask      = ""
    output-channel = {file, socket}
    report-rule-description = true
    ignore-remain-flow      = true
    input.max-message-size  = 60000
    security.max-instances  = 100000
    security.smp.ring-size  = 1000
    ip-encapsulation-index  = LAST
}
```

`session-timeout`: The configuration bloc specifies the session timeout in seconds for different types of applications. For example:

```
session-timeout
{
    # Default timeout (in seconds) for sessions.
    default-session-timeout = 60
    short-session-timeout =   15
    # This is reasonable for Web and SSL connections
        especially when long polling is used.
    long-session-timeout =  6000
    # Usually applications have a long polling period of
        about 3~5 minutes. This option is for persistent
        connections like messaging applications and so on
    live-session-timeout  = 1500
}
```

`session-report`: The configuration bloc enables or disables the reporting of protocols that belong to an IP session. The output reports can be reported to a file,

redis, kafka or a combination of these channels as being specified in the `security.output-channel` field. For example:

```
session-report
{
    enable = true
    output-channel = {}
    # enable/disable specific reports for specific protocol
        applications
    ftp  = true
    http = true
    rtp  = false
    ssl  = true
    gtp  = false #specific reports for LTE eNodeB network
    rtt-base = CAPTOR # Order of timestamp's origin that is
        used to calculate RTT in QOS reports
        # - SENDER: timestamp being marked in packet by its
            sender, e.g., tcp option TSVal, TSerc
        # This timestamp is available only on certain
            monitoring protocol, e.g., TCP
        # - CAPTOR: timestamp being marked in packet at the
            captured moment by its captor (e.g., tcpdump,
            wireshark)
        # This timestamp is always available.
        # The value of this rtt setting can be one of the
            followings:
        # - CAPTOR (by default): use only CAPTOR.
        # - SENDER: use only SENDER. Ignore if it is not
            available
        # - PREFER_SENDER:  use SENDER if it is available,
            otherwise CAPTOR
}
```

→ Note: Currently if `gtp` is enable then the other specific reports must be disable.

`micro-flows`: The configuration bloc specifies the criteria to use to determine if an IP flow is considered as a micro flow.

IF an IP flow has some characters, such as, number of packets or bytes, being less than some limits, then it will be considered as a micro flow.

A micro-flow is identified by its protocol ID. This means that a micro-flow represents several flows having (1) the same protocol ID, and, (2) their total number of bytes or packets are less than the given thresholds

A single micro flow statistics will not be reported separately, statistics from several micro flows will be aggregated and reported together. Aggregating micro flows statistics reduces the number of reports; however, one will lose fine grained information about each flow.

The micro-flows are not reported periodically but when one of its total number of packets, or bytes, or flows is greater than or equal some limit given by `report-packet-count`, `report-byte-count` and `report-flow-count`, respectively.

→ Note: Set value of a limit/threshold to zero to unlimit it

For example:

```
micro-flows
{
    enable = true
    packet-threshold =   2     # packets count threshold to
        consider a flow as a micro flow:
    byte-threshold   = 100     # data volume threshold to
        consider a flow as a micro flow:
```

```
    # a micro-flow will be reported to its output-channel:
    # - at the end of execution of MMT-Probe
    # - or when one of its stats (packets, bytes, flows) is
        greater than or equal some limit as specified below.
    report-packet-count   =  1000
    report-byte-count     = 10000
    report-flow-count     =   500

    output-channel = {}
}
```

data-output: The configuration block is intended for defining the criteria to be used regarding the reporting of specific meta-data. For the time being, it only includes criteria to indicate when to include user agent parsing. The include-user-agent indicates the threshold in terms of data volume for parsing the user agent in Web traffic. This configuration bloc will be extended in the future when new reporting needs arise.

For example:

```
data-output
{
    # Indicates the threshold in terms of data volume for
        parsing the user agent in Web traffic.
    # The value is in kilo Bytes (kB). If the value is zero,
        this indicates that the parsing of the user agent
        should be done.
    # To disable the user agent parsing, set the threshold to
         a negative value (-1).
    include-user-agent = 32
}
```

event-report: The configuration bloc allows to indicate what protocol attributes should be reported when an given event occurred. The event indicates the condition that should be satisfied in order to report the attributes; and, the attributes indicate the application (protocol) attributes that need to be reported when the event occurs. Events and attributes should be in "protocol.attribute" format.

There can be multiple event-report configuration blocs. Each bloc is uniquely identified by its name; for instance, event-report report1, event-report report2.

The output reports can be reported to a file, redis, kafka or a combination of these channels, as specified in the security.output-channel field.

For example: when Probe sees an IP packet having IP.SRC, it will report ARP.AR_HLN and IP.SRC. If a packet does not contain an attribute, e.g., ARP.AR_HLN, the attribute's value is replaced by an empty value (either 0 or "").

For example:

```
event-report ip-event
{
    enable          = true
    event           = "ip.src"
    attributes      = {"arp.ar_hln", "ip.src", "meta.
        proto_hierarchy"}
    output-channel = {socket}
}
```

system-report: The configuration bloc defines the CPU and memory usage reports to be generated. The period indicates the time-interval for reporting. The

output reports can be reported to a file, redis, kafka or a combination of these channels, as specified in the `security.output-channel` field.

For example:

```
cpu-mem-usage
{
    enable          = true
    period          = 5
    output-channel  = {}
}
```

`behavior`: The configuration bloc indicates that the Probe should produce reports for MMT-Behaviour. For example:

```
behaviour
{
    enable        = true
    output-file   = "data.csv"
    output-dir    = "/opt/mmt/probe/result/behaviour/online/"
    retain-files  = 300
}
```

`reconstruct-data ftp`: The output configuration bloc indicates that the Probe should reconstruct FTP files and generate reports.

To enable the reconstruction, one needs to enable the options `enable-tcp-reassembly = true` and `enable-ip-defragmentation = true` .

To enable the reports, the two following options must be enable: `session-report.enable = true` and `session-report.ftp = true`.

The FTP reconstruction reports will be written to `output-channel`, see `security.output-channel`.

The FTP reconstruction is only available for the single threaded operation.

For example:

```
reconstruct-data ftp
{
    enable          = true
    output-dir      = "/opt/mmt/probe/reconstruct-data/ftp/"
    output-channel  = {}
}
```

`reconstruct-data http`: The configuration bloc indicates that the Probe should reconstruct the HTTP data. It has same requirements as `construct-data ftp`.

For example:

```
condition_report reconstruct_http
{
    enable          = true
    output-dir      = "/opt/mmt/probe/reconstruct-data/http/"
    output-channel  = {}
}
```

`radius-report`: The bloc configures the reports for the RADIUS protocol.

The `message-id` indicates the kind of message one needs to report: set to 0 to report all messages; set to a number (from 1 to 255) to indicate the code of message to report, e.g., 1: Access-Request, 2: Access-Accept, 45: CoA-NAK, etc.

The output reports can be reported to a file, redis, kafka or a combination of these channels, as specified in the `security.output-channel` field.

```
radius-output
{
    enable          = true
    message-id      = 0
    output-channel  = {}
}
```

3.1.2 Execution Parameters

Probe can receive several parameters when starting via command line. For example, run `mmt-probe -h` to obtain its list of parameters:

```
mmt-probe [<Option>]
Option:
   -v               : Print version information, then exits.
   -c <config file> : Gives the path to the configuration
       file (default: ./mmt-probe.conf, /opt/mmt/probe/mmt-
       probe.conf).
   -t <trace file>  : Gives the trace file for offline
       analyze.
   -i <interface>   : Gives the interface name for live
       traffic analysis.
   -X attr=value    : Override configuration attributes.
                      For example "-X file-output.enable=
                          true -Xfile-output.output-dir=/tmp/
                          " will enable output to file and
                          change output directory to /tmp.
                      This parameter can appear several
                          times.
   -x               : Prints list of configuration attributes
        being able to be used with -X, then exits.
   -h               : Prints this help, then exits.
```

3.2 MMT-Operator

3.2.1 Configuration File

The configuration for the MMT-Operator is defined in the `/opt/mmt/operator/config.json` file. User can given another configuration file to Operator via its running parameter `-config`, for example: `mmt-operator -config=/home/tata/operator.json`.

The configuration file is structured as the following:

```
{
  "database_server": {
    "host": "localhost",
    "port": 27017
  },
  "redis_input": {
    "host": "localhost",
    "port": 6379
  },
  "kafka_input": {
    "host": "192.168.0.195",
    "port": 2181,
    "ssl.ca.location": "data/kafka-ca.cert"
  },
  "file_input": {
    "data_folder": ["/opt/mmt/probe/result/report/online/", "
        /mount/probe2/"],
    "delete_data": true,
    "nb_readers": 1
  },
  "input_mode": "file",
  "probe_analysis_mode": "offline",
  "local_network": [
```

```
    {
      "ip": "192.168.0.0",
      "mask": "255.255.0.0"
    },{
      "ip": "172.16.0.0",
      "mask": "255.240.0.0"
    },{
      "ip": "10.0.0.0",
      "mask": "255.0.0.0"
    },{
      "ip": "fe80::",
      "mask": "8"
    },{
      "ip": "127.0.0.1",
      "mask": "255.255.255.255"
    }
  ],
  "probe_stats_period": 5,
  "buffer": {
    "max_length_size": 30000,
    "max_interval": 5
  },
  "micro_flow": {
    "packet": 1,
    "byte": 64
  },
  "query_cache": {
    "enable" : true,
    "folder" : "/tmp/cache/",
    "bytes" : 10e6,
    "files" : 9999
  },
  "auto_reload_report": true,
  "retain_detail_report_period": 3600,
  "limit_database_size" :  999999999999,
  "port_number": 8080,
  "log_folder": "/opt/mmt/operator/log/",
  "log" : ["error", "warn", "info"],
  "is_in_debug_mode": true,
  "modules": ["link","network","dpi","security","enodeb"],
  "modules_config" : {
     "enodeb" : {
        "mysql_server" : {
           "host": "localhost",
           "port": 3306,
           "user": "root",
           "password": "montimage",
           "database": "eNodeB"
        }
     }
   }
}
```

The content of `config.json` can be divided into the following groups:

Input: This group specifies how MMT-Operator can receive meta-data generated by the MMT-Probe. The value of `input_mode` can be one of the following:

- `"file"`: MMT-Operator will read meta-data from a file in the folders `data_folder`. After reading a file, MMT-Operator can delete the file and its semaphore depending on the value defined by `delete_data`. The reading can be accelerated by using several reader in parallel.
- `"redis"`: MMT-Operator will receive meta-data from a REDIS sever defined by `redis_server`
- `"kafka"`: MMT-Operator will receive meta-data from a KAFKA

sever defined by `kafka_server`

Behaviour: This group configures the behaviour of the MMT-Operator:

- `port_number` is a port number used to connect to the MMT-Operator web application.
- `local_network` is an array indicating IP ranges of local networks. Each network is given by an IP and network mask.
- `buffer` limits the size of buffers of MMT-Operator either by (i) the number of elements, `max_length_size`, or by the timestamp, `max_interval`. With a small buffer, MMT-Operator must flush data more frequently to the database and to the Web browser clients.
- `micro_flow` indicates retaining detailed information on small flows. This parameter set defines how the MMT-Operator defines micro-flows: maximum number of packets in a flow and maximum number of bytes of data in a flow.
- `query_cache` configures the cache of data getting from mongoDB.
- `probe_stats_period` is a number, in seconds, indicating the rate of statistic reports from the MMT-Probe. This number must be the same as the value given by `stats_period` in the configuration file of MMT-Probe.
- `probe_analysis_mode` represents the executing mode of MMT-Probe. Its value must be the same as the value given by `input-mode` in the configuration file of MMT-Probe, e. g., either `"online"` or `"offline"`.
- `auto_reload_report` indicates whether MMT-Operator Web application should refresh periodically its the graphical charts that are displaying, in GUI, to user. This option needs to be `true` if one wants view some graphs in realtime.

Maintain Database: This group contains some thresholds used for the maintenance of the historical database.

- `retain_detail_report_period` is an interval, in seconds, indicating how long must the MMT-Operator retains information in the collection called `detail` in the database. This collection contains detailed information on each report generated by the MMT-Probe. For example, in the file `config.json` above, MMT-Operator will delete the reports that are older than 1 hour.
- `limit_database_size` is number o bytes indicating the size limit of database. When database overpasses this limit, the oldest documents of each collection will be deleted until the limitation is hold.

Execution log: This group configures the execution logs of the MMT-Operator.

- `log_folder` is the folder that will contain the execution logs of the MMT-Operator.
- `log` indicates kinds of log to be written to file. The full list of log can be `["error", "warn", "log", "info"]`
- `is_in_debug_mode` determines whether the MMT-Operator is running in debug mode, i. e., producing detailed log information. How-

ever, this will slow down the MMT-Operator. Its value can be `true` or `false`.

Modules: This group enables or disables some modules as well as their configurations.

- `modules` is a list of modules to be enabled.

  The full list of modules can be `["link", "network", "dpi", "application", "security", "behavior", "enodeb", "ndn", "video", "sla", "stat"]`

- `module_config` is specific configuration of each module.

3.2.2 Execution Parameters

An configuration option of MMT-Operator can be overridden by giving through running parameter `-X`.

For example, `-Xdatabase_server.host=10.0.0.2` will change `host` of `database_server` bloc to `10.0.0.2`.

The value of `-X` parameter is in format `attribute=value`, where:

- `attribute` is path to the attribute to be changed. Dot sign can be used to access to attributes of a bloc, for example,

  `-Xmodules_config.enodeb.mysql_server.host="10.0.0.2"`.

- `value` is a primitive value, such as, boolean, number, or string. It must not be an Object or Array.

3.3 MMT-Security

We can modify the default thresholds of the MMT-Security module in the file `/opt/mmt/security/mmt-security.conf`. The file must be created if it does not exist.

Input:

```
# Maximum size, in bytes, of a report received from the MMT-
    Probe:
input.max_message_size 3000
```

Security Engine:

```
# Maximum number of reports that will be stored in a ring
    buffer:
security.smp.ring_size 1000
# Maximum number of instances of a rule:
security.max_instances 100000
```

Alert:

```
# Number of consecutive alerts for one rule without the full
    description. The first alert of a rule always contains a
    description of its rule. However, to avoid huge outputs, a
     certain number of consecutive alerts of that rule can be
    sent without the full description. For instance if value
    is 20 then the first alert will contain the full
    description and the next 20 alerts generated by the same
    rule will not.
# Set value to 0 to include the description in all alerts.
output.ignore_description 20
```

4 MMT Security Properties

4.1 Description

The MMT-Security properties are intended for formally specifying the occurrence of events that denotes a security rule to be respected or an attack or vulnerability to be avoided. They rely on LTL (Linear Temporal Logic) and are written in XML format. This has the advantage of being a simple and straight forward structure for the verification and processing performed by the tool. A graphical editor for creating and editing MMT-Security properties is under development. In the context of this document, we use the terms of properties and rules interchangeably.

Description: An MMT-Security properties XML file can contain as many properties as required. The file needs to begin with a `<beginning>` tag and end with `</beginning>`. Each property begins with a `<property>` tag and ends with `</property>`. A property is a "general ordered tree" and can be graphically represented as shown in Figure 2:

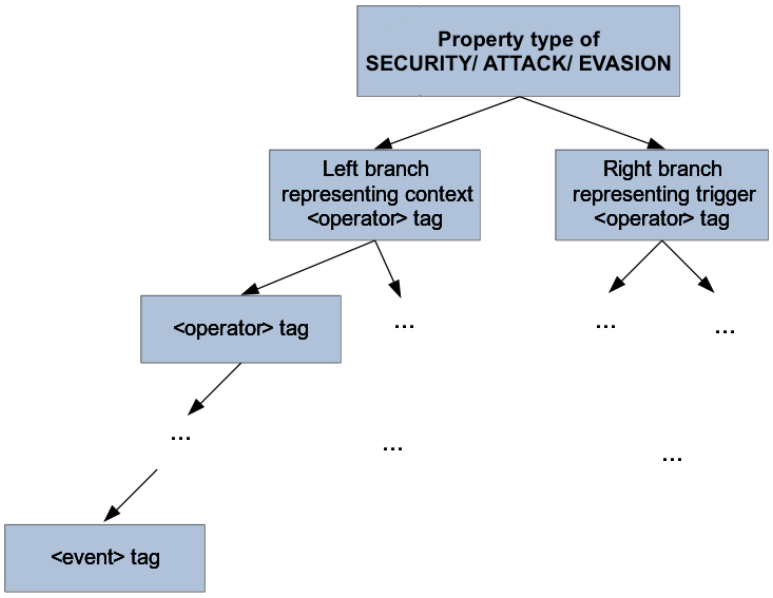

Figure 2: MMT-Security property structure

Property validation or violation: The nodes of the property tree are: the property node (required), operator nodes (optional) and event nodes (required). The property node is forcibly the root node and the event nodes are forcibly leaf nodes. In general, the left branch represents the context and the right branch represents the trigger. This means that the property is found valid when the trigger is found valid; and the trigger is checked only if the context is valid. In other words:

- If the context is verified and the trigger is not, then a property non-respect instance is detected.
  - In the case of a `SECURITY` rule, this means that the context of the rule has been found and, since the trigger was not, we conclude that the `SECUIRTY` rule has been violated.
  - In the case of an `ATTACK` or `EVASION` rule, this means that the context of an attack has occurred but the trace was attack free

because we did not detect the final trigger.

- If the context and the trigger are verified, then a property respect instance is detected.
  - In the case of a SECURITY rule, this means that the context of the rule has been found, as well as the trigger. We conclude that the SECURITY rule has been respected.
  - In the case of an ATTACK or EVASION rule, this means that the context of an attack has occurred, as well as the trigger. We conclude that the behavioural attack/evasion has been detected.

To illustrate how a property is specified, a simplified instance of the Figure 2) in XML gives :

General format of a property

```
<property value="THEN" delay_units="s" delay_min="0+"
    delay_max="2" property_id="1" type_property="ATTACK"
    description="(description of attack)">
    <operator value="THEN" delay_max="1" >
    <event value="COMPUTE" event_id="1"
        description="(description of event 1)"
        boolean_expression="(Boolean expression allowing to
            detect event 1)"/>
    <event value="COMPUTE" event_id="2"
        description="(description of event 2)"
        boolean_expression="(Boolean expression allowing to
            detect event 2))"/>
    </operator>
    <event value="COMPUTE" event_id="3"
        description="(description of event 3)"
        boolean_expression="(Boolean expression allowing to
            detect event 3)"/>
</property>
```

→ Note: A context can occur before, at the same time or after a trigger, depending on how the delays or counters attributes are defined (as explained in the next table 1). To date, we have a library of MMT rules including the following types:

- Uni-packet rules: The events will be verified on the same packet, i.e., the extracted meta-data of one packet satisfy both the context and the trigger conditions.
- Multi-packet, Uni-session rules: The events will be verified in multiple packets but belonging to the same session. This means that the source and destination IP addresses and ports correspond to the same session.
- Multi-session, Uni-probe rules: The events will be verified in multiple packets in multiple sessions based on the reports coming from one single MMT-Probe.
- Multi-session, Multi-probe rules: The events will be verified in multiple packets in multiple sessions based on the reports coming from several MMT-Probes.

Property attributes: The <property> tag contains several attributes, some required, some optional:

Table 1: Property tag attributes

Attribute	Accepted values	Req./opt., default value	Description
property_id	Integer	required	Property id number, should go from 1 to n where n is the total number of properties in the XML file. In one or different XML files, two different properties must have two different `property_id`
value	`"THEN"`	required	A property is a tree and each node can have one or more branches. The <property> tag here represents the root of a tree that can have 2 branches: the context and the trigger. The value attribute indicates that the left branch representing the context needs to be valid before, after or at the same time we check the right branch representing the trigger, depending on the delays defined for the node.
type_property	`"ATTACK"`, `"EVASION"`, `"SECURITY"`	required	Indicates that the property specifies a potential attack-/evasion (or abnormal behaviour) or that the property specifies a security rule that needs to be respected.
`delay_min`	Integer	optional, default value 0	Defines the validity period ([delay_min, delay_max]) of the left branch (e.g. context). If we have [a,b] with a=b=0 then the left branch needs to occur in the same time as the right branch. If we have a<b<=0 then the right branch needs to have occurred in the past time interval with respect to the left branch. If we have 0<=a<b then the right branch needs to occur in the future with respect to the left branch. If the time runs out before detecting the events concerned then we are in a TIMEOUT condition, i.e., the context occurs but the trigger never arrived in the specified time interval. This would mean that the property failed. Note that in the case that an event should not occur during a certain time interval, we refer to it as TIMEIN instead of TIMEOUT. The default unit of time used is the second but this can be changed using the attribute delay_units.
delay_max	Integer	optional, default value 0	For delay_min, a + sign before the value (e.g., "+0") means strictly greater than the value and for delay_max a - sign means less than.
delay_units	Integer	optional, default value is "s" (seconds)	Defines the time units used in delay_min and delay_max attributes. If default value is not wished then this has to be defined before these two attributes. Possible values are Y,M,D,H,m,s(default),ms or mms.
counter_max (under development)	Integer	optional, default value 0	Similar to [delay_min, delay_max] we can define [counter_min, counter_max] where the unit is the number of packets analysed. If the count runs out before detecting the events concerned, then we are in, respectively, a COUNTIN or COUNTOUT condition.
counter_min (under development)	Integer	optional, default value 0	
description	String	required	The text that clearly explains what the property is about.

Table 2: Property tag attributes (cont.)

Attribute	Accepted values	Req./Opt., default value	Description
if_satisfied	String	optional	Defines what action should be performed if the property is satisfied. The string gives the name of the function that should be executed (see Reactive Functions 4.2).

Operator attributes: The < operator> tag in a property contains several attributes, some required, some optional:

Table 3: Operator tag attributes

Attribute	Accepted values	Req./Opt., default value	Description
value	`"THEN"`, `"OR"`, `"AND"`, `"NOT"`	required	Operators are used to combine different events and build complex events. "THEN" operator is used to describe ordered events, "AND" operator is used to describe two events without any order and "OR" operator is used to describe the occurrence of a least one of the events. "NOT" negates the underlying sub-tree.
description	String	optional	Gives the text that clearly explains what the complex event is about.
delay_min	Integer	optional, default value 0	Same as for the <property> tag. Note that these attributes are not to be used for the "OR" and "NOT" operators.
delay_max	Integer	optional, default value 0	
counter_max	Integer	optional, default value 0	
counter_min	Integer	optional, default value 0	
repeat_times (under development)	Integer	optional, default value 1	Allows detecting the repetition of a complex event occurrence a number of times.

Event attributes: Properties indicate the sequence of events that need to be observed. Events indicate the conditions that need to be verified on a packet or a set of packets for the event to hold. The <event> tag in a property contains several attributes, some required, some optional:

Table 4: Event tag attributes

Attribute	Accepted values	Req./Opt., default value	Description
event_id	Integer	required	This field allows identifying each event and starts from 1 to n where n is the total number of events in the current property.
description	String	required	Gives the text that clearly explains what the event is about.
value	`"COMPUTE"`	required	Means that the events need to resolve a Boolean expression that is equal to `true` or `false` depending on the attributes values
boolean_expression	expression	required	A Boolean expression, similar to a Boolean expression in the C language (explained in the following paragraph).

Boolean expressions: The `boolean_expression` is a logical combination of "<protocol name>.<field name>" and "<number>" with the following operators: "&&", "||", ">", ">=", "<", "<=", "==", "!=", "+", "-", "/", "*". Blanks and tabs are ignored. Note that the "&&" operator needs to be written in XML as "&&", "<" needs to be written as "<", etc.

A simple `boolean_expression` can be of the form:

`(<expression> <operator> <expression>)`

in which, `<expression>` is one of the followings:

- `<protocol_name>.<attribute_name>`
- `<protocol_name>.<attribute_name>.<event_id>`
- `<numeric_value>`
- `<string_value>`
- `true`
- `false`
- embedded function

The Boolean expressions must respect some syntax rules:

- The building bloc needs to start with a "(" and end with a ")". Note that to avoid any ambiguities the quotes should explicitly define how the expression is to be resolved, e.g, A && B || C should be written as ((A && B) || C) or (A && (B || C)).
- An attribute needs to be identified with its `protocol_name` and its `attribute_name` separated by a dot ".".
- If the attribute refers to an attribute value from another event, then the name needs to be followed by a second "." and the `event_id` number of the event concerned. For instance we can have:
  - `(arp.ar_op == 0)` means that the field `arp.ar_op` should be equal to zero.

– `(arp.ar_op == arp.ar_op.1)` means that the field `arp.ar_op` should be equal to the field `arp.ar_op` of the event in the same rule with the `event_id` equal to 1.

Example of a Boolean expression:

For example; the following expression means that the event is valid if the packet received corresponds to the ARP protocol; the `ar_op` is 2; `ar_sip` is the same as `ar_tip` of an event 1; and, `ar_sha` is different with `ar_sha` of event 2. Where events 1 and 2 occurred before or will occur after depending on the order that the events should occur (defined by the time intervals specified).

```
"(((arp.ar_op == 2)&amp;&amp;(arp.ar_sip == arp.ar_tip.1))
&amp;&amp;(arp.ar_sha != arp.ar_sha.2))"
```

The following table gives a more detailed explanation about the information that is used in the Boolean expression:

Table 5: Boolean expressions

Name	Type	Description
protocol_name	String	Indicates the protocol of the packet that we need to inspect.
attribute_name	String	Indicates the field of the protocol that we will use for verifying a condition in the event.
numeric_value	Integer	Gives the value that will be used to compare with the packet field value. Note that the `protocol_name` and the `attribute_name` allow uniquely identifying the type of value a field contains (e.g. number, IP address, MAC address in binary format). Two specific constants, `true` and `false`, represent respectively a non-zero and zero number.
string_value	String	Gives a chain of characters, enclosed by ' and ', e.g., `'GET'`
reference	Integer	Indicates that we are to use the data obtained for the field from an event that occurred before (a previously received packet that validated the previous event). The name is a number that clearly identifies the event and needs to be exactly the same as the value given in the `event_id` attribute of the corresponding <event> tag. Note that one should refer to events that occurred beforehand in the property.

Extensions:

Embedded functions can be used to extend the elements used in the Boolean expressions (see section 4.2).

Property compilation:

After creating the XML properties, we need to compile them by using the executable `compile_rule`:

```
./compile_rule <rule_name>.so <property>.xml
```

4.2 Embedded Functions

Embedded functions are functions that allow implementing calculations that are too complicated to define using only classical operators on fields in the Boolean expressions of security properties. One can use existing embedded functions or implement a new function. In both cases, they can be used in the Boolean expressions by using the syntax:

#<name_of_function>(<list of parameters>)

For instance:

```
(#em_is_search_engine( http.user_agent ) == true)
```

where `http` is the protocol name and `user_agent` is the attribute name (i.e., packet meta-data).

Implement new Embedded Functions:

The embedded functions must respect the C syntax and are used in Boolean expressions using the syntax described above. Embedded functions are implemented inside `<embedded_functions>` tag of the XML rule files and will be compiled together with the rules. The input parameters are either of the type `const char *` or `int val`. For instance, the `em_is_search_engine` above could be implemented as follows:

```
<embedded_functions><![CDATA[
//code C
static inline bool em_is_search_engine(const char *agent){
    if( strstr( agent, "google" ) != NULL )
        return true;
    if( strstr( agent, "bing" ) != NULL )
        return true;
    return false;
}]]></embedded_functions>
```

→ Note: : The function name should be prefixed by `em_` to avoid any confusion with other functions in the system.

One can also implement code into 2 other functions to initialize and free common resource:

- `void on_load(){ ... }` is called when the rules inside the XML file being loaded into MMT-Security
- `void on_unload(){ ... }` is called when exiting MMT-Security

Pre-implemented Embedded Functions:

There exist several embedded functions that have been implemented inside MMT-Security.

1. `is_exist( proto.att )` checks whether an event has an attribute of a protocol, e. g., `is_exist( http.method )` will return `true` if the current event contains protocol HTTP and attribute `method` has a non-null value, otherwise it will return `false`.

   Normally MMT-Security has a filter that allows an event in a rule to be verified only if any proto.att used in its boolean expression contains value. If one of them has not, the rule will not be verified. This allows to reduce number of verification of boolean expression, thus increases the performance.

   For example, given an event having the following boolean expression:

   ```
   ((ip.src != ip.dst) && (#em_check_URI(http.uri) == 1))
   ```

   This expression is verified only if `ip.src` and `ip.dst` and `http.uri` are not null, hence only HTTP packets are verified (it does not verify every IP packets).

   However, if one use the following expression, that is totally having the same meaning with the previous one:

   ```
   ((ip.src != ip.dst) && ((is_exist(http.uri) == true )
           && (#em_check_URI(http.uri) == 1)))
   ```

MMT-Security need to verify the expression against any IP packet as the `is_exist` function tell MMT to exclude `http.uri` from its filter.

2. `is_empty( proto.att )`, e.g., `is_empty(http.uri)` checks whether the string value is empty, i.e., its length is zero.

3. User can use *any standard C functions* as embedded function, e. g., `(#strstr( http.user_agent, 'robot') != 0)` to check if `http.user_agent` contains a sub-string `"robot"`.

→ Note: Before using a C function, the library containing that function need to be included. The following libraries have been pre-included:

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "mmt_lib.h"
#include "pre_embedded_functions.h"
```

Thus when using a function that does not defined inside these libraries, one need to include its library. For example:

```
<embedded_functions><![CDATA[
#include <math.h>

static inline bool function em_check( double port ){
    double x = sqrt( port );
    ...
}
]]></embedded_functions>
```

Reactive Functions: Reactive functions allow user perform some action when a rule is satisfied. The functions will be called each time their rules are satisfied. When a security and attack rules are satisfied, they will give `not_respected` and `detected` verdicts respectively.

To implement and use a reactive function, one need to

1. implement a C function inside `<embedded_functions>` tag. The function has the following format:

```
typedef void (*mmt_rule_satisfied_callback)(
  const rule_t *rule,//rule being validated
  int verdict,       //DETECTED, NOT_RESPECTED
  uint64_t timestamp,//moment (by time) the rule is
      validated
  uint64_t counter,  //moment (by order of message) the
      rule is validated
  const mmt_array_t * const trace  //messages that
      validate the rule
);
```

2. put the function name in attribute `if_satisfied` of the rule you want to react. For example: `if_satisfied="em_print_out"`

```
<beginning>
<embedded_functions><![CDATA[
static void em_print_out( const rule_info_t *rule, int
    verdict, uint64_t timestamp, uint64_t counter, const
    mmt_array_t * const trace ){
   const char* trace_str =
       mmt_convert_execution_trace_to_json_string( trace,
        rule );
```

```
5     printf( "detect rule %d\n%s\n", rule->id, trace_str );
6  }]]></embedded_functions>
7
8  <property value="THEN" delay_units="s" delay_max="0"
       property_id="10" type_property="EVASION"
9      description="HTTP using a port different from 80 and
           8080." if_satisfied="em_print_out">
10     <event value="COMPUTE" event_id="1"
11         description="a port different from 80 and 8080"
12            boolean_expression="((http.method != '')&amp;&
                  amp;((tcp.dest_port != 80)&amp;&amp;(tcp.
                  dest_port != 8080)))"/>
13     <event value="COMPUTE" event_id="2"
14            description="HTTP packet"
15            boolean_expression="(ip.src != ip.dst)"/>
16 </property>
17 </beginning>
```

High Performance Rules: To write a rule having a high performance, one need to:

- use only the proto.att in boolean expression when need

  Please refer to the usage of function `is_exist` to get an example.

- Use explicitly the following tcp flags to filter out unwanted verification: tcp.fin, tcp.syn, tcp.rst, tcp.psh, tcp.ack, tcp.urg, tcp.ece, tcp.cwr.

  For example, the 2 following boolean expressions have the same meaning:

  `(tcp.flags == 4)` and `((tcp.flags == 4)  (tcp.rst == 1))`

  They both return true when only RST flag of a TCP packet is on, but the latter is better as MMT verifies its rule only when tcp.flags and tcp.rst are not zero. Usually less than about 1% packets having `tcp.rst != 0`, consequently the rule using the second expression will be verified against only 1% packets.

- reduce `delay_max` of a rule to a suitable value

  When having a higher value of `delay_max` MMT-Security creates more rule instances to correlate different events of different packets. When `delay_max` is zero, the rule is call simple rule, MMT-Security verifies the rule and gives verdict immediately without creating any rule instances. A simple rule is verified much faster than a complex one that has non-zero `delay_max`.

  → MMT-Security can verify 12400 simple rules or 600 complex rules at 10Gbps.

- optimize implementation of embedded functions

  The embedded functions are called each time their boolean expressions are verified. Consequently, rather than initialize something, for example, connection to database, inside these functions, one can do such a task, only once, inside function `on_load` then store the connection into a static local variable that will be used inside the embedded functions.

- always use embedded function with `static inline` keyword

  For more information about advantage of inline, please refer to document of gcc: *An Inline Function is As Fast As a Macro.*

4.3 Example

4.3.1 Property without Embedded Functions: ARP Spoofing Example

In order to provide a complete but easy to understand example, the detection of ARP spoofing is presented here in detail.

ARP spoofing: Address Resolution Protocol (ARP) is a telecommunications protocol used for resolution of network layer addresses into link layer addresses, a critical function in multiple-access networks. ARP was defined by RFC 826 in 1982[3]. ARP spoofing[4](RFC5227[5]) is a technique used to attack a local-area network (LAN). ARP spoofing may allow an attacker to intercept data frames on a LAN, modify the traffic, or stop the traffic altogether. Figure 3 represents an attack by a device using MAC address MAC3 (in red).

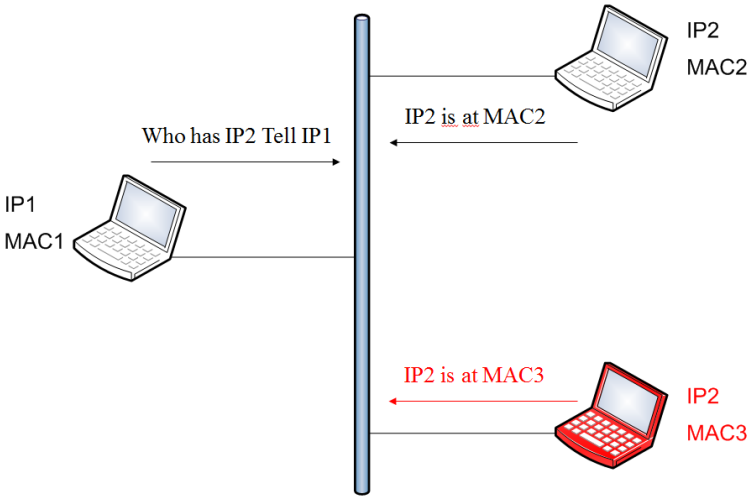

Figure 3: ARP spoofing

Attack model: Following an ARP request (*who has*), a node receives more than one reply with different MAC addresses. This could mean that an IP duplication has been detected that could potentially be an ARP spoofing attack.

This attack behaviour can be specified as follows:

$$(e_1; e_2)_{0,4} \;\; AFTER_{0,10} \;\; e_3$$

Where:

- **Event 1** (e_1): reception of ARP request message
- **Event 2** (e_2): reception of ARP reply message (src MAC2)
- **Event 3** (e_3): reception of ARP reply message with (scr MAC3) != (src MAC2)
- **AFTER$_{0,4}$** is an operator indicating that event e_3 occurs after the occurrence of the sequence of events e_1 ; e_2 with a time interval that represents the interval defined in RFC5227 between conflicting

[3] `https://en.wikipedia.org/wiki/Address_Resolution_Protocol`
[4] `https://en.wikipedia.org/wiki/ARP_spoofing`
[5] `http://www.networksorcery.com/enp/rfc/rfc5227.txt`

ARPs below which a host must not attempt to defend its address. This interval could be reduced or increased to detect spoofing attempts and avoid to many verdicts that are false positives.

This security attack is specified by MMT-Security rule as the following:

```
<beginning>
<property value="THEN" delay_units="s" delay_min="0+"
    delay_max="4" property_id="4" type_property="ATTACK"
          description="IPv4 address conflict detection (
              RFC5227). Possible arp poisoning.">
    <operator value="THEN" delay_units="s" delay_min="0+"
        delay_max="4">
      <event value="COMPUTE" event_id="1"
             description="An arp who was requested"
             boolean_expression="((arp.ar_op == 1) &amp;&amp;
                  (ethernet.src != ethernet.dst))"/>
      <event value="COMPUTE" event_id="2"
             description="An arp reply with MAC address"
             boolean_expression="(((arp.ar_op == 2)&amp;&amp
                 ;(arp.ar_sip == arp.ar_tip.1)) &amp;&amp; (
                 ethernet.packet_count &gt; 1))"/>
    </operator>
    <event value="COMPUTE" event_id="3"
           description="An arp reply but with different MAC
               address"
           boolean_expression="((((arp.ar_op == 2)&amp;&amp;(
               arp.ar_sip == arp.ar_tip.1))&amp;&amp;(arp.
               ar_sha != arp.ar_sha.2)) &amp;&amp; (ethernet.
               packet_count != 0))"/>
</property>
</beginning>
```

4.3.2 Property with an Embedded Function: NFS Example

This property aims to detect the occurrence of the event: "NFS Client uploads a file synchronized to another server" (Fig. 4). This action can violate the privacy of the data within an organization's network.

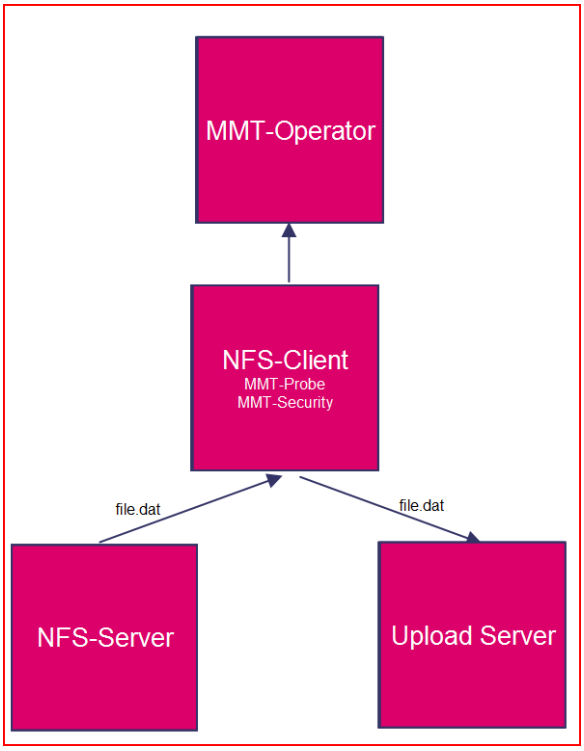

Figure 4: Upload a file synchronized from NFS Server to another server

Using a pre-implemented standard C function:

Suppose that we have one MMT-Probe that assesses NFS traffic as well as HTTP traffic. The attack behaviour can be specified by an MMT rule as follows:

$$(e_1; e_2)_{0,20}$$

```
<beginning>
<!-- Property 30: Simple multi-session property -->
<property value="THEN" delay_units="s" delay_min="0+"
    delay_max="20" property_id="30" type_property="ATTACK"
    description="NFS synchronization and then upload">
    <event value="COMPUTE" event_id="1"
           description="NFS synchronization"
           boolean_expression="(nfs.file_name != '')"/>
    <event value="COMPUTE" event_id="2"
           description="Upload the same file"
           boolean_expression="((http.packet_count != 0)&amp
               ;&amp;(#strstr(tcp.p_payload, nfs.file_name.1)
               != 0))"/>
</property>
</beginning>
```

Where:

- **Event 1** (e_1): The reception of an NFS synchronization packet and extraction of the file's name.
- **Event 2** (e_2): The reception of an HTTP packet with the file's name in the payload (verified by the function `#strstr(tcp.p_payload, nfs.file_name.1)`).

Using a new embedded function:

Suppose that we have two MMT-Probes: One verifies NFS traffic and sends reports to the REDIS bus. The other, analyzes HTTP traffic and

uses an Embedded Function to also report to the REDIS bus. The attack behaviour can be specified in an MMT rule as follows:

$$(e_1; e_2)_{0,0}$$

```
<beginning>
<embedded_functions><![CDATA[
#include "hiredis/hiredis.h"
#include <sys/time.h>

static inline int em_check_nfs_redis(const char *p_payload){
    redisContext *c, *command;
    redisReply *reply;

    const char *hostname = "127.0.0.1";
    int port = 6379;
    struct timeval timeout = { 1, 500000 }; // 1.5 seconds
    c = redisConnectWithTimeout(hostname, port, timeout);
    if (c == NULL || c->err) {
        if (c) {
            printf("Connection error: %s\n", c->errstr);
            redisFree(c);
        } else {
            printf("Connection error: Impossible to allocate
                redis context\n");
        }
        exit(0);
    }

    /* Let's check what we have inside the list */
    reply = redisCommand(c,"LRANGE multisession.report 0 -1")
        ;
    if (reply->type == REDIS_REPLY_ARRAY) {
      int j=0;
        for (j = 0; j < reply->elements; j++) {
            char probe_report[256];
            char *token;
            strcpy(probe_report, reply->element[j]->str);
            token = strtok(reply->element[j]->str, ",");
            int i = 0;
            while (token != NULL) {
                if (i==1) {
                    //check the validity of the report
                    struct timeval now;
                    gettimeofday(&now, NULL);
                    double element_ts = atof(token);
                    if (now.tv_sec - element_ts > 300) {
                        redisCommand(c,"LREM multisession.
                            report 1 %s", probe_report);
                        //Delete the out-dated report
                        i++;
                        break;
                    }
                }
                if (i==2){
                    if (token[0] == ' ')
                        break;
                    if (strstr(p_payload, token) != NULL){
                        redisFree(c);
                        return 1;
                    }
                }
                token = strtok(NULL, ",");
                i++;
            }
        }
    }
    redisFree(c);
    return 0;
```

```
62 }]]></embedded_functions>
63
64 <property value="THEN" delay_units="s" delay_min="0"
       delay_max="0" property_id="32" type_property="ATTACK"
       description="Upload a file coming from NFS server (
       detection based on two probes)">
65     <event value="COMPUTE" event_id="1"
66            description="Context: HTTP packet"
67            boolean_expression="((http.packet_count != 0) &amp
                  ;&amp; (tcp.payload_len != 0))"/>
68     <event value="COMPUTE" event_id="2"
69            description="Trigger: HTTP payload contains the
                  file comming from NFS server"
70            boolean_expression="(#em_check_nfs_redis(tcp.
                  p_payload) == 1)"/>
71 </property>
72 </beginning>
```

Where:

- **Event 1** (e_1): The reception of an HTTP packet with payload length greater than 0.
- **Event 2** (e_2): The embedded function, `em_check_nfs_redis`, is called to:
  - Get the names of all the files synchronized by the NFS Server. These reports are generated by another MMT-Probe and stored in the REDIS bus.
  - Verify if the name of the file, *file name*, is found in the packet payload.

4.4 Other Case Studies

Applying MMT-Security to a particular industrial scenario usually involves the following steps:

1. Develop any plugins needed to parse the data to be analysed. Note that parsers for most Internet protocols and applications are already available.
2. Develop a "main.c" that will use the libraries to capture and analyse the data.
3. Define a set of properties that specify the analysis to be performed.

The tool has been applied or is being applied to several industrial scenarios, such as:

- Generating broadband reporting for 4G mobile networks (for an equipment manufacturer);
- Capturing information useful for detecting botnets and other cyber attacks (`http://www.acdc-project.eu/` and `https://sissden.eu/`);
- Detecting malicious nodes in an ad-hoc network (`http://projects.celtic-initiative.org/SAN/`);
- Estimating user Quality of Experience (QoE) of video flows (for an equipment manufacturer);
- Analysing and fine tuning of delay tolerant networks;

- Detecting and mitigating attacks in radio protocol communications, Near Field Communication ecosystems and vehicular communications (`http://www.itea2-diamonds.org/index.html`);
- Monitoring secure interoperability (`http://inter-trust.eu/`);
- Performance analysis for web sites (`http://genibeans.com/cgi-bin/twiki/view/Pimi/WebHome/` and `http://swept.eu/`);
- Monitoring function for security and performance of Software Defined 5G Mobile Networks, Datacenters, Critical Infrastructures, Cloud infrastructures and services, Cyber-Physical Systems;

4.5 Compile MMT-Security Rules

MMT-Security rules are specified in plain text following an XML format. The rules need to be compiled into a dynamic C library used by MMT.

The compiled rules must put in either `./rules` or /opt/mmt/security/rules. The former has higher priority and only one of them will be taken into account by MMT-Security. Specifically, if MMT-Security found `./rules` folder in the current folder executing, it will use the rules inside this folder and it does not take into account the rules in `/opt/mmt/security/rules`.

Compile rules: MMT provides a tool to compile the rules: `/opt/mmt/security/bin/compile_rule`. To execute it do:

```
./compile_rule output_file property_file [-c] [gcc param]
```

where:

1. `output_file`: is the path of file containing the result that can be either a .c file or .so file.
2. `property_file`: is the path where the property file can be found.
3. optional parameters:
   - `-c`: will generate only the C code. This option allows manually modifying the generated code before compiling it. After generating the C code, the tool prints out the command that needs to be executed for compiling it.
   - `gcc param`: used to generate the C code, and compile it to obtain the .so file. These parameters will be directly transmitted to the gcc compiler, for example, `"-I/home/tata/include -lmath"`

For example, the following command compiles the rule detecting arp spoofing presented above:

```
./compile_rule /opt/mmt/security/rules/arp.so arp.xml
```

Obtain information from the compiled rules: To get some basic information about the compiled rules (such as, ID, description) MMT provides a tool: `/opt/mmt/security/bin/rule_info`.

- By default, the tool will print out the information on all rules located in `/opt/mmt/security/rules/`, for instance:

```
/opt/mmt/security/bin/rule_info
Found 35 rules.
1 - Rule id: 1
  - type             : attack
```

```
  - events_count    : 4
  - variables_count : 5
  - variables       : ip.dst (178.13), ip.src (178.12),
      tcp.dest_port (354.2), tcp.flags (354.6), tcp.
      src_port (354.1)
  - description     : Several attempts to connect via ssh
       (brute force attack). Source address is either
      infected machine or attacker (no spoofing is
      possible).
  - if_satisfied    : (null)
  - if_not_satisfied: (null)
  - create_instance : 0x7fad94d0be80
  - hash_message    : 0x7fad94d0bd50
  - version         : 1.1.2 (67fa423 - 2017-5-19
      18:14:31), dpi version 1.6.8.0 (b3e727b)
2 - Rule id: 10
  - type            : evasion
  - events_count    : 2
  ...
...
```

- The tool can also be used to inspect a specific compiled rule by giving the rule path as parameter, for instance:

```
./rule_info /opt/mmt/security/rules/4.arp.so
Found 1 rule.
1 - Rule id: 4
  - type            : attack
  - events_count    : 3
  - variables_count : 7
  - variables       : arp.ar_op (30.5), arp.ar_sha (30.6)
      , arp.ar_sip (30.7), arp.ar_tip (30.9), ethernet.
      dst (99.2), ethernet.packet_count (99.4099),
      ethernet.src (99.3)
  - description     : IPv4 address conflict detection (
      RFC5227). Possible arp poisoning.
  - if_satisfied    : (null)
  - if_not_satisfied: (null)
  - create_instance : 0x7f5063e7be70
  - hash_message    : 0x7f5063e7bcd0
  - version         : 1.1.2 (67fa423 - 2017-5-19
      18:14:34), dpi version 1.6.8.0 (b3e727b)
```

5 MMT-DPI

5.1 Extract API

MMT-DPI is the core packet processing module. It is a C library that analyses network traffic using Deep Packet/Flow Inspection (DPI/DFI) techniques in order to extract hundreds of network and application based events, including: protocols field values, network and application QoS parameters and KPIs. MMT-DPI has a plugin architecture for the addition of new protocols and a public API for integration with third party probes.

The Figure 5 shows how a User Program interacts with the library. It captures events that are processed by the library's Event Processing Engine. The library will parse the event using the integrated or added Plugins, classify it to determine what type of message it is, extract the needed data from it and send notifications back to the User Program. The classification uses algorithms that allow analysing them and even identify encrypted messages using, for instance, pattern matching and machine learning techniques.

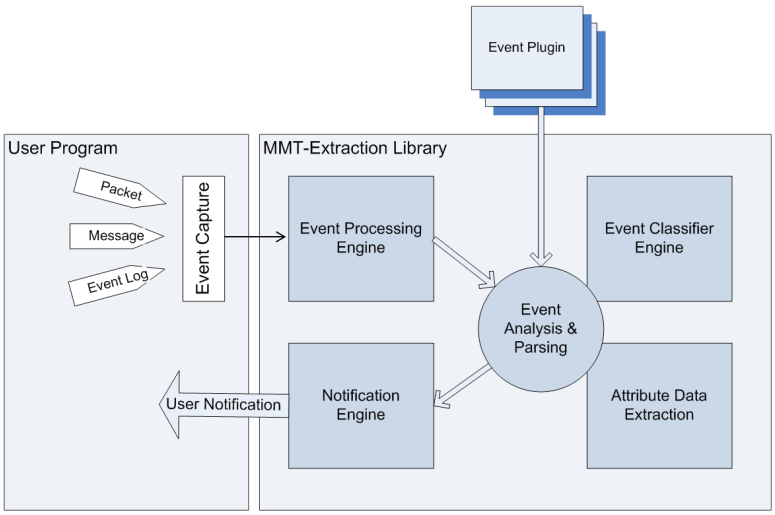

Figure 5: MMT-DPI Library

5.1.1 Using MMT-DPI in a development project

In order to use the MMT-DPI library in a development project, one must perform the followings:

- Include the header file `mmt_core.h`: This file is the only one that needs to be included.
- By default, the plugins folder locates at `/opt/mmt/plugins` after installing MMT-DPI. However you can create a folder called `plugins` in the directory where the executable is located, then you should copy the file `libmmt_tcpip.so` into `plugins` to be able to analyse TCP/IP based network traffic. If one has developped any new MMT-DPI plugin, it must be copied to this `plugins` folder. Explaining how to add new plugins is described later in this document;
- Build a `main.c` program that uses the library's API (see the following section);

- Compile the program, linking the MMT-DPI library by using the option: `-I/opt/mmt/dpi/include -L/opt/mmt/dpi/lib -lmmt_core`

5.1.2 Extraction API description

The MMT-DPI API is specified in the header files provided with the download package. In the following we will shortly describe the content of the different header files:

- `mmt_core.h`: this is where the core extraction API functions are defined;
- `data_defs.h`: this is where the data related API functions are defined. In addition, the different data structures that might be used in an integration project are defined here;
- `types_defs.h`: this is where the new data types are defined;
- `extraction_lib.h`: this file contains the generic extraction functions;
- `plugin_defs.h`: this file contains the definitions that can be used for the plugin development.

Initialization: In order to use the MMT-DPI library, one must initialize it by using the following function signature:

```
1 init_extraction () ;
```

The initialization returns a positive number on success and zero on failure. It is good practice to always check the return number of `init_extraction()`.

The following function will create an extraction handler:

```
1 mmt_handler_t *mmt = mmt_init_handler (
2         uint32_t stacktype,
3         uint32_t options,
4         char     *errbuf );
```

where the `stacktype` is 1 for 10Mb Ethernet, `options` is 0 and `errbuf` is a `char [1024]` to hold any error message in the case that the return value is `NULL`. The return value is the handle and will be needed by the other functions that need to be called.

For capturing the packets, you can select any library such as: DPDK, PCAP,... In some of our example, we use the PCAP library and, in particular, the following functions (not detailed in this document):

- `pcap_open_live`
- `pcap_open_offline`
- `pcap_next`
- `pcap_loop`
- `pcap_close`

Each time a packet is recuperated using `pcap_next` or `pcap_loop`, it is passed over to the Event Processing Engine using the `packet_process` function described below in Message processing.

The following function will close the extraction handler and free any previously allocated memory:

```
void mmt_close_handler( mmt_handler_t * mmt );
```

The following function will close the extraction and free any previously allocated memory:

```
int close_extraction ();
```

Message processing:

MMT-DPI can process network packets in the de-facto PCAP[6] format, raw messages, log messages, etc.

The following function hands a packet to the core engine of MMT-DPI:

```
int packet_process (
    mmt_handler_t *mmt,
    struct pkthdr *header,
    u_char        *packet );
```

This function should be called for every packet to process. The `header` parameter of the function is a pointer to the meta-data of the message that include the message arrival time, the message length, etc. The `packet` parameter of the function is a pointer to the message data. The `packet_process` function will return a positive value on successful processing. Zero is returned if an internal error is encountered, although this should never happen.

Registering extraction attributes:

The MMT-DPI library allows registering attributes for extraction. This limits the extraction overhead since only the data that is needed is recuperated. Attributes are fields in a network packet, specific data in a log entry, etc.

The any of the following two functions can be used to register an attribute for extraction:

```
void register_extraction_attribute (
    mmt_handler_t *mmt,
    uint32_t       protocol_id,
    uint32_t       attribute_id );
```

```
void register_extraction_attribute_by_name (
    mmt_handler_t *mmt,
    const char    *proto_name,
    const char    *attribute_name);
```

An attribute is identified by the protocol and attribute id or names. If the registration succeeds, a positive value will be returned; zero is returned on failure.

The following function allows verifying if an attribute, identified by its protocol and attribute identifiers, is already registered:

```
int is_registered_attribute (
        mmt_handler_t *mmt,
        uint32_t       protocol_id,
        uint32_t       attribute_id );
```

This function will return a positive value if the attribute is already registered.

The any of the two following functions allow unregistering an attribute:

[6] PCAP reference

```
int unregister_extraction_attribute (
    mmt_handler_t *mmt,
    uint32_t       protocol_id,
    uint32_t       attribute_id );
```

```
int unregister_extraction_attribute_by_name (
    mmt_handler_t *mmt,
    uint32_t       proto_name,
    uint32_t       attribute_name );
```

If the unregistration succeeds, a positive value will be returned.

To recuperate the data in an attribute, any of the following two functions will return a pointer to the data if it was detected in the current event (i.e., the current packet that is being analysed). If the attribute or data does not exist for the current event then `NULL` will be returned.

```
void * get_attribute_extracted_data (
    mmt_handler_t *mmt,
    uint32_t       protocol_id,
    uint32_t       attribute_id);
```

```
void * get_attribute_extracted_data_by_name (
    mmt_handler_t *mmt,
    const char    *protocol_name,
    const char    *attribute_name );
```

Registering callbacks:

The MMT-DPI library allows registering callback functions. A registered callback function will be called following the extraction of attributes whenever a packet is processed.

The following function allows registering a packet handler:

```
int register_packet_handler (
    mmt_handler_t                   *mmt,
    int                              handler_id,
    generic_packet_handler_callback function,
    u_char                          *args );
```

A packet handler is a callback that will be called each time a packet is processed. If needed, the user can provide a pointer to an argument that will be passed to the callback function when it is called. The callback function is associated with an identifier that should be unique, i.e., two callback functions cannot have the same identifier. This function will return a positive value upon success.

To verify that a packet handler callback function is registered with a given identifier, one can use the following function:

```
int is_registered_packet_handler (
        mmt_handler_t  *mmt,
        int             handler_id );
```

This function returns a positive value if a registered callback function is found for the provided identifier.

In order to unregister an already registered packet handler one can use:

```
int unregister_packet_handler (
        mmt_handler_t *mmt,
        int            handler_id );
```

In addition to packet handlers, it is possible to register a callback function to be called when an attribute is detected. This can be done with the following function:

```
int register_attribute_handler (
        mmt_handler_t              *mmt,
        uint32_t                    protocol_id,
        uint32_t                    attribute_id,
        attribute_handler_function handler_fct,
        void                       *handler_condition,
        void                       *user_args );
```

This function allows registering an attribute handler that is a callback that will be called when the attribute defined by the given protocol and attribute ids is detected (i.e, not NULL). If needed, the user can provide a pointer to an argument that will be passed to the callback function when it is called. In the current version the parameter defined by `handler_condition` is not implemented and should be set to NULL. This function will return a positive value upon success.

The following function registers an attribute handler given by its protocol and attribute names:

```
int register_attribute_handler_by_name (
        mmt_handler_t *mmt,
        const char    *protocol_name,
        const char    *attribute_name,
        attribute_handler_function handler_fct,
        void *handler_condition,
        void *user_args);
```

To verify that an attribute handler is registered, one can use the following function:

```
int has_registered_attribute_handler (
        mmt_handler_t *mmt,
        uint32_t       protocol_id,
        uint32_t       attribute_id );
```

This function returns a positive value if a handler is already registered with the attribute defined by its protocol and attribute identifiers.

In order to unregister an already registered attribute handler one can use:

```
int unregister_attribute_handler (
        mmt_handler_t * mmt,
        long protocol_id,
        long attribute_id );
```

```
int unregister_attribute_handler_by_name (
        mmt_handler_t *mmt,
        const char    *protocol_name,
        const char    *attribute_name );
```

Data functions: In addition to the core functions presented so far, MMT-DPI provides a number of utility functions to assist the user of the library.

The following function returns the protocol name for a given numeric identifier:

```
const char * get_protocol_name_by_id (
        mmt_handler_t *mmt,
        uint32_t       protocol_id );
```

NULL is returned if the given identifier does not correspond to any configured protocol.

The following function returns the identifier of the protocol for a given name:

```
long get_protocol_id_by_name (
        mmt_handler_t *mmt,
        const char    *protocol_name );
```

Zero is returned if the given name does not correspond to any configured protocol.

The following function indicates if the attribute exists for given protocol and attribute ids.

```
int is_protocol_attribute (
        mmt_handler_t *mmt,
        uint32_t       protocol_id,
        uint32_t       attribute_id );
```

The following function returns the name of the attribute corresponding to the given protocol and attribute identifiers:

```
const char * get_attribute_name_by_protocol_and_attribute_ids
     (
        mmt_handler_t *mmt,
        uint32_t       protocol_id,
        uint32_t       attribute_id );
```

NULL is returned if there is no attribute corresponding to the given identifiers.

The following function returns the identifier of the attribute corresponding to the given protocol and attribute names:

```
long get_attribute_id_by_protocol_and_attribute_names (
        mmt_handler_t *mmt,
        const char    *protocol_name,
        const char    *attribute_name );
```

Zero is returned if there is no attribute corresponding to the given names.

The following function returns the identifier of the attribute corresponding to the given protocol id and attribute name:

```
long get_attribute_id_by_protocol_id_and_attribute_name (
        mmt_handler_t *mmt,
        uint32_t       protocol_id,
        const char    *attribute_name );
```

Zero is returned if there is no attribute corresponding to the given parameters.

The following function returns the identifier of the data type of the attribute corresponding to the given protocol and attribute identifiers:

```
long get_attribute_data_type (
        mmt_handler_t *mmt,
        uint32_t       protocol_id,
        uint32_t       attribute_id );
```

Zero is returned if there is no attribute corresponding to the given parameters.

The following function returns the data size, in number of bytes, of the attribute corresponding to the given protocol and attribute identifiers:

```
int get_data_size_by_proto_and_field_ids (
        mmt_handler_t *mmt,
        uint32_t       protocol_id,
        uint32_t       attribute_id );
```

Zero is returned if there is no attribute corresponding to the given parameters.

The following function returns the data size of the given MMT data type:

```
int get_data_size_by_data_type (
        mmt_handler_t *mmt,
        int            data_type );
```

Zero is returned if the data type is unknown.

The following function returns the position in the message of the attribute corresponding to the given protocol and attribute identifiers:

```
int get_field_position_by_protocol_and_field_ids (
        mmt_handler_t *mmt,
        uint32_t       protocol_id,
        uint32_t       attribute_id );
```

For attributes where the position depends on the content of the message, `POSITION_NOT_KNOWN` (value -1) will be returned. The position is defined as the byte offset from the beginning of the packet.

5.2 Building the main

To build a main that uses the MMT-DPI and MMT-Security libraries one should carefully study the `ARP_probe` example (Sec. 4.3.1). The main steps that need to be followed for building a `main.c` are:

1. Implement a callback function that will be called each time MMT-DPI parse successfully a packet. In this function, one need to retrieve value of protocol attributes from MMT-DPI and pass them to MMT-Security by calling function `mmt_sec_process`.
2. Implement a callback function `mmt_sec_callback` that will be is called by MMT-Security each time a property is satisfied
3. Call the functions that initialize the processing:
   - `init_extraction`
   - `mmt_init_handler`
   - `mmt_sec_init`
   - `mmt_sec_register`
4. Initialise the PCAP API by calling one of the following functions:
   - `pcap_open_live`: for real time live analysis
   - `pcap_open_offline`: for offline analysis
5. Perform the loop that reads a packet from the input trace file or the network interface. This includes the following:
   - Call the function: `pcap_next`

- Set the header for the message before processing the packet:
  - set timestamp:
    header.ts = p_pkthdr.ts;
  - set length of data available for the packet:
    header.caplen = p_pkthdr.caplen;
  - set length of packet:
    header.len = p_pkthdr.len;
  - currently not used but should be set to NULL:
    header.user_args = NULL;
- Call MMT-DPI lib function that will parse the packet and analyse it: `packet_process`

  Note that each time a relevant packet (according to security properties) is received the `mmt_sec_process` function will be executed and, when a security property is satisfied or reaches a not satisfied condition, the callback function `mmt_sec_callback` will be executed.

6. Call the functions that end the processing:
   - `mmt_sec_unregister`
   - `mmt_sec_close`
   - `mmt_close_handler`
   - `pcap_close`

5.3 Add a new Protocol

5.3.1 MMT Plugin API

The MMT-DPI library has a plugin architecture. It is possible to extend the extraction engine with new protocols. For this, a plugin needs to be created specifying the extraction to add. An MMT plugin will initialize a protocol structure that contains the required information regarding the protocol attributes, as well as the functions allowing extracting the data corresponding to these attributes. In this section we will describe the required steps in order to create a new MMT plugin.

When creating a new MMT plugin, you must use the API functions described in the following sub-sections.

Initializing a protocol structure

Creating a new plugin requires the initialization of a protocol structure. A protocol is defined by a unique identifier. The first step in the process of creating a plugin is to get a protocol structure using the following function:

```
1 protocol_t *init_protocol_struct_for_registration (
2         int          protocol_id,
3         const char *protocol_name);
```

This function will return a pointer to a free protocol structure with the given identifier. If a protocol with the same identifier is already registered, this function will return NULL.

Registering protocol attributes

Once the protocol structure is initialized, you need to add the attributes belonging to the protocol. An attribute has an identifier, a name, a data type, a data length, a position within the packet (offset), a scope (packet

or session) and an extraction function (can be generic if the position and data length are known).

```
int register_attribute_with_protocol (
        protocol_t          *protocol_struct,
        attribute_metadata_t attr);
```

This function registers the attribute `attr` with the protocol `protocol_struct`.

Registering a classification function with parent protocol

For some protocol which is encapsulated in other protocols (parent protocols), you need to register a classification function with parent protocols to. For example, the classification function of "HTTP" protocol need to register with parent protocol is `TCP`. The classification function is protocol specific and needs to be implemented by the user.

```
void register_classification_function_with_parent_protocol (
        uint32_t                        parent_protocol_id,
        generic_classification_function classification_fct,
        int                             weight);
```

This function registers a classification function `classification_fct` with the parent protocol is the given `parent_protocol_id`.

Registering a classification function (optional)

Once the protocol structure is initialized, you need to add a classification function. A classification function identifies the type of protocol/message encapsulated in the current protocol/message. For example, the classification function of `IP` protocol will tell if the encapsulated protocol is `TCP`, `UDP`, `ICMP`, etc. The classification function is protocol specific and needs to be implemented by the user.

```
void register_classification_function (
        protocol_t                      *protocol_struct,
        generic_classification_function classification_fct);
```

This function registers a classification function `classification_fct` for the protocol identified by the `protocol_struct` parameter. The signature of the classification function is defined by the function type `generic_classification_function` defined in `mmt_core.h`

Registering an initialized protocol structure

The final step when creating a plugin is to register the created protocol structure in the MMT-DPI. For this purpose, you must use the following function:

```
int register_protocol (
        protocol_t *protocol_struct,
        uint32_t    protocol_id);
```

This function registers the protocol defined by the given protocol structure and protocol identifier in the extraction core. Remember that a protocol has a unique identifier, and registering a protocol structure with an already used identifier will cause the function to fail. On success, this function will return `PROTO_REGISTERED` (value 1). `PROTO_NOT_REGISTERED` (value 0) will be returned on failure.

Utility functions

The MMT Plugin API has a number of utility functions that can be very useful when creating a new plugin.

```
int is_valid_protocol_id ( uint32_t protocol_id );
```

This function verifies if a given identifier is valid. A protocol identifier must have a positive value less than the `PROTO_MAX_IDENTIFIER`. A positive value is returned if the given identifier is valid, a negative value otherwise.

```
int is_registered_protocol ( uint32_t protocol_id );
```

This function verifies if a protocol with the given identifier is already registered. `PROTO_REGISTERED` (value 1) is returned if a protocol is already registered, `PROTO_NOT_REGISTERED` (value 0) is returned otherwise.

```
int is_free_protocol_id_for_registraction (
        uint32_t protocol_id );
```

This function returns a positive value if there is no protocol registered with the given identifier. A negative value is returned otherwise.

Generic extraction functions

A number of generic extraction functions are implemented in the MMT-DPI and can be reused by the plugins. They include:

```
int general_byte_to_byte_extraction (
        const ipacket_t *packet,
        int              proto_index,
        attribute_t     *extracted_data);
```

This is a generic extraction function. It will copy, into the data part of `extracted_data` structure a defined number of bytes from the data part of the packet structure. Any extraction function must have the same signature and must return a positive value if the extraction is successful.

```
int general_short_extraction_with_ordering_change(
        const ipacket_t *packet,
        int              proto_index,
        attribute_t     *extracted_data);

int general_int_extraction_with_ordering_change(
        const ipacket_t *packet,
        int              proto_index,
        attribute_t     *extracted_data);

int general_short_extraction(
        const ipacket_t *packet,
        int              proto_index,
        attribute_t     *extracted_data);

int general_int_extraction(
        const ipacket_t *packet,
        int              proto_index,
        attribute_t     *extracted_data);
```

These 4 functions provide the extraction of `short` (2 bytes) and `int` (4 bytes) data with or without ordering change.

Utility structures

In addition to the utility functions, the MMT Plugin API has a number of structures that can help organize plugin related data.

```
typedef struct attribute_metadata_struct {
  int id; /**< identifier of the attribute. */
  char alias[Max_Alias_Len + 1]; /**< the alias(name) of the
      attribute */
  int data_type; /**< the data type of the attribute */
  int data_len; /**< the data length of the attribute */
  int position_in_packet; /**< the position in the packet of
      the attribute. */
  int scope; /**< the scope of the attribute (packet, session
      , ...). */
  generic_attribute_extraction_function extraction_function;
      /**< the extraction function for this attribute. */
} attribute_metadata_t;
```

This structure defines the attribute related information; it can be used to model the information of the protocol attributes.

This can be seen in the following code for the UDP protocol, that models UDP's attributes related information.

```
static attribute_metadata_t udp_attributes_metadata[
    UDP_ATTRIBUTES_NB] = {
  {UDP_SRC_PORT, UDP_SRC_PORT_ALIAS, MMT_U16_DATA, sizeof (
      short), 0, SCOPE_PACKET,
      general_short_extraction_with_ordering_change},
  {UDP_DEST_PORT, UDP_DEST_PORT_ALIAS, MMT_U16_DATA, sizeof (
      short), 2, SCOPE_PACKET,
      general_short_extraction_with_ordering_change},
  {UDP_LEN, UDP_LEN_ALIAS, MMT_U16_DATA, sizeof (short), 4,
      SCOPE_PACKET,
      general_short_extraction_with_ordering_change},
  {UDP_CHECKSUM, UDP_CHECKSUM_ALIAS, MMT_U16_DATA, sizeof (
      short), 6, SCOPE_PACKET,
      general_short_extraction_with_ordering_change},
};
```

5.3.2 Create a MMT Plugin

A MMT plugin, as described above, will initialize and register a protocol structure. This initialization/registration must be performed by using a function called `init_proto`. When the Extraction core tries to load a plugin, it will search for and execute the function with this name. If the plugin does not implement such a function, it is considered invalid. Therefore, the creation of a new plugin is equivalent to the implementation of this function `init_proto`. The following steps describe the creation of a plugin:

Pre-step Create a new function `init_proto`

```
int init_proto();
```

This function must implement the next steps. It is highly recommended that you use the utility function when creating a plugin. First step

Request a protocol structure with a defined protocol identifier for initialization using:

```
protocol_t *init_protocol_struct_for_registration(
    uint32_t    protocol_id,
    const char *protocol_name);
```

Second step Define the protocol attributes and create an array of the attributes. Register the defined attributes with the protocol structure initialized in step 1.

This is where the protocol specific code will be created.

Third step If the current protocol is encapsulated in other protocols, you need to register a classification function with parent protocols. For example, the classification function of `QUIC` protocol need to register with parent protocol is `UDP` protocol. To register a classification function with parent protocol use:

```
void register_classification_function_with_parent_protocol(
    uint32_t                        parent_protocol_id,
    generic_classification_function classification_fct,
    int                             weight);
```

Of the protocol does not require a classification function from parent protocols (or it does not have any parent protocol), this step can be omitted. For example ETHERNET protocol does not have a parent protocols and thereforce it does not require registering a classifification with parent protocol.

Forth step

Once the protocol structure is initialized, you can add a classification function. A classification function identifies the type of protocol/message encapsulated in the current protocol/message. For example, the classification function of IP protocol will tell if the encapsulated protocol is TCP, UDP, ICMP, etc. The classification function is protocol specific and needs to be implemented by the user. To register a classification function use:

```
void register_classification_function(
        protocol_t *protocol_struct,
        generic_classification_function classification_fct);
```

If the protocol does not require a classification function, this step can be omitted, or NULL can be registered. For example, ARP protocol does not encapsulate any other protocol, and therefore it does not require a classification function.

Fifth step

The final step, when creating a plugin, is to register the created protocol structure in the MMT-Extraction core. For this purpose, you must use:

```
int register_protocol(
        protocol_t *protocol_struct,
        uint32_t    protocol_id);
```

This function will register the protocol structure if no other protocol is already registered with the given protocol identifier.

If the protocol registration is successful, the plugin function init_proto must return a positive value indicating that the plugin has been successfully loaded; otherwise a negative value must be returned to let the core perform the necessary cleanup.

5.3.3 Create a MMT Plugin by using Plugin Generator

You can create a MMT Plugin by using Plugin Generator. It is a Java Application which can create generate a basic source of a plugin bases on the plugin file

Create plugin file

The plugin file contains the basic information to generate a plugin.

The ETHERNET protocol plugin can be created base on the following plugin file:

```
Protocol {
  define ETH_P_IP = 0x0800
  define ETH_P_ARP = 0x0806
  define ETH_P_IPV6 = 0x86DD
  define ETH_P_RARP = 0x8035
  define PROTO_ARP = 11
  define PROTO_IP = 15
  define PROTO_IP6 = 16

  Properties {
    label = "ETHERNET"
    id = "12"
    context = "false"
    session = "false"
    encapsulation = "true"
```

```
    encoding = "network"
  }

  Attributes {
    struct ethhdr {
      MMT_DATA_MAC_ADDR h_dest;
      MMT_DATA_MAC_ADDR h_source;
      uint16_t h_proto;
    }
    attribute {
      alias="src" data_type="MMT_DATA_MAC_ADDR" data_len="
          sizeof (mac_addr_t)"
      offset="0" scope="SCOPE_PACKET"
    }
    attribute {
      alias="dst" data_type="MMT_DATA_MAC_ADDR" data_len="
          sizeof (mac_addr_t)"
      offset="2" scope="SCOPE_PACKET"
    }
  }

  classifynext {
    switch(val(h_proto)) {
      case ETH_P_IP:
      nextencaps = PROTO_IP
      nextoffset = sizeof(struct ethhdr)
      break
      case ETH_P_ARP:
      nextencaps = PROTO_ARP
      nextoffset = sizeof(struct ethhdr)
      break
      case ETH_P_IPV6:
      nextencaps = PROTO_IP6
      nextoffset = sizeof(struct ethhdr)
      break
      case ETH_P_RARP:
      nextencaps = PROTO_ARP
      nextoffset = sizeof(struct ethhdr)
      break
      default:
      break
    }
  }
}
```

Another example about QUIC protocol:

```
Protocol {
  Properties {
    label = "QUIC"
    id = "630"
    context = "false"
    session = "false"
    encapsulation = "false"
    encoding = "network"
  }
  Attributes {
    attribute {
      alias="connection_id" data_type="MMT_U32_DATA" data_len
          ="sizeof(int)"
      offset="0" scope="SCOPE_PACKET"
    }
    attribute {
      alias="sequence" data_type="MMT_U32_DATA" data_len="
          sizeof(int)"
      offset="0" scope="SCOPE_PACKET"
    }
  }
}
```

In the plugin file, we can define the protocol properties: label, id, context, session (`true` : if the protocol maintains session, `false` : if the protocol does not maintain session), encapsulation (true: if the protocol encapsulates other protocol), encoding. We also can define the attributes of protocols: header structure, attribute extractions. Finally we can define the classification function of the protocol if it encapsulates other protocol.

Generate basic plugin source code

After creating the plugin file, we can generate the source code for new plugin by using the `MMTPluginGenerator` (please contact Montimage if you need this tool).

```
java -jar MMTPluginGenerator.jar plugin_file plugin_dir
```

There will be 2 source files generated, a `.c` file and a `.h` file.

For example, to generate source code for ETHERNET plugin:

```
mkdir quic_src
java -jar MMTPluginGenerator.jar quic quic_src/
ls quic_src/
quic_mmt_plugin.c quic_mmt_plugin.h
```

Following is the `quic_mmt_plugin.h` file generated by Plugin Generator

```
/* Generated with MMT Plugin Generator */

#ifndef QUIC_H
#define QUIC_H
#ifdef  __cplusplus
extern "C" {
#endif

#include "plugin_defs.h"
#include "mmt_core.h"


#define PROTO_QUIC 630

#define PROTO_QUIC_ALIAS "QUIC"


enum quic_attributes {

  QUIC_CONNECTION_ID = 1,

  QUIC_SEQUENCE,

  QUIC_ATTRIBUTES_NB = QUIC_SEQUENCE,

};


#define QUIC_CONNECTION_ID_ALIAS "connection_id"

#define QUIC_SEQUENCE_ALIAS "sequence"


int init_quic_proto_struct();



#ifndef CORE
int init_proto();
#endif //CORE



#ifdef  __cplusplus
}
```

```
#endif
#endif  /* QUIC_H */
```

Following is the `quic_mmt_plugin.c` file generated by Plugin Generator

```
/* Generated with MMT Plugin Generator */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "quic_mmt_plugin.h"
#include "extraction_lib.h"




/*
 * QUIC data extraction routines
 */



int quic_connection_id_extraction(const ipacket_t * ipacket,
    int proto_index,
                                  attribute_t *
                                      extracted_data) {
  /* Get the protocol offset */
  int proto_offset = get_packet_offset_at_index(ipacket,
      proto_index);


  int attribute_offset = quic_attributes_metadata[0].
      position_in_packet;


  /* Get the attribute data length */
  int attribute_length = sizeof(MMT_U32_DATA);


  *((unsigned int *) extracted_data->data) = ntohl(*((
      unsigned int *) & ipacket->data[proto_offset +
      attribute_offset]));




  return 1;
}



int quic_sequence_extraction(const ipacket_t * ipacket, int
    proto_index,
                             attribute_t * extracted_data) {
  /* Get the protocol offset */
  int proto_offset = get_packet_offset_at_index(ipacket,
      proto_index);


  int attribute_offset = quic_attributes_metadata[1].
      position_in_packet;


  /* Get the attribute data length */
  int attribute_length = sizeof(MMT_U32_DATA);


  *((unsigned int *) extracted_data->data) = ntohl(*((
      unsigned int *) & ipacket->data[proto_offset +
      attribute_offset]));

```

```
55
56
57    return 1;
58  }
59
60
61  classified_proto_t quic_stack_classification(ipacket_t *
        ipacket) {
62    classified_proto_t retval;
63    retval.offset = 0;
64    retval.proto_id = PROTO_QUIC;
65    retval.status = Classified;
66    return retval;
67  }
68
69  static attribute_metadata_t quic_attributes_metadata[
        QUIC_ATTRIBUTES_NB] = {
70
71    {QUIC_CONNECTION_ID, QUIC_CONNECTION_ID_ALIAS, MMT_U32_DATA
          , sizeof(uint32_t), 0, SCOPE_PACKET,
          quic_connection_id_extraction},
72
73    {QUIC_SEQUENCE, QUIC_SEQUENCE_ALIAS, MMT_U32_DATA, sizeof(
          uint32_t), 0, SCOPE_PACKET, quic_sequence_extraction},
74
75  };
76
77
78  int init_quic_proto_struct() {
79    protocol_t * protocol_struct =
          init_protocol_struct_for_registration(PROTO_QUIC,
          PROTO_QUIC_ALIAS);
80
81    if (protocol_struct != NULL) {
82
83      int i = 0;
84      for (; i < QUIC_ATTRIBUTES_NB; i ++) {
85        register_attribute_with_protocol(protocol_struct, &
              quic_attributes_metadata[i]);
86      }
87
88
89      register_protocol_stack(PROTO_QUIC, PROTO_QUIC_ALIAS,
            quic_stack_classification);
90      return register_protocol(protocol_struct, PROTO_QUIC);
91    } else {
92      return -1;
93    }
94  }
95
96  #ifndef CORE
97  int init_proto() {
98    return init_quic_proto_struct();
99  }
100 #endif //CORE
```

From the basic source code, we need to complete the plugin by adding/updating some more function such as: classification functions, extraction functions,....

Complete plugin source code:

Register classification function with parent protocol

After having the basic source code, we need to adding the classification functions if needed.

The following is the classification function of QUIC protocol which will be called in QUIC's parent protocol is UDP protocol, it needs to be registered with UDP protocol.

```
int mmt_check_quic(ipacket_t * ipacket, unsigned index)
{
  printf("[inf] mmt_check_quic: %lu - %d\n", ipacket->
      packet_id, index );
  int l4_offset = get_packet_offset_at_index(ipacket, index);
  // int l4_packet_len = ipacket->p_hdr->caplen - l4_offset;
  struct udphdr * udp = NULL;
  udp = (struct udphdr *) & ipacket->data[l4_offset];
  char * payload = (char*) &ipacket->data[l4_offset + sizeof(
      struct udphdr)];
  int quic_offset = l4_offset + sizeof(struct udphdr);
  classified_proto_t quic_proto = quic_stack_classification(
      ipacket);
  quic_proto.offset = quic_offset;
  int ver_offs;
  if (udp != NULL) {
    uint16_t sport = ntohs(udp->source), dport = ntohs(udp->
        dest);
    // debug("QUIC: Calculating QUIC over UDP");
    if ((((sport == 80) || (dport == 80) || (sport == 443) ||
         (dport == 443))))
    {
      // Settings without version. First check if PUBLIC
          FLAGS & SEQ bytes are 0x0. SEQ must be 1 at least.
      if ((payload[0] == 0x00 && payload[1] != 0x00) || ((
          payload[0] & QUIC_NO_V_RES_RSV) == 0))
      {
        int seq = sequence(payload);
        fprintf(stderr, "[PROTO_QUIC] %lu sequence: %d!\n",
            ipacket->packet_id, seq);
        if (seq < 1)
        {
          fprintf(stderr, "[PROTO_QUIC] %lu mmt_check_quic:
              Not QUIC!\n", ipacket->packet_id);
        }

        fprintf(stderr, "[PROTO_QUIC] %lu mmt_check_quic:
            FOUND QUIC!\n", ipacket->packet_id);
        return set_classified_proto(ipacket, index + 1,
            quic_proto);
      }

      // Check if version, than the CID length.
      else if (payload[0] & QUIC_VER_MASK)
      {
        // Skip CID length.
        ver_offs = connect_id(payload[0]);
        fprintf(stderr, "[PROTO_QUIC] %lu connection id: %d!\
            n", ipacket->packet_id, ver_offs);
        if (ver_offs >= 0)
        {
          unsigned char vers[] = {payload[ver_offs], payload[
              ver_offs + 1],
                                  payload[ver_offs + 2],
                                      payload[ver_offs + 3]
                                 };

          // Version Match.
          if ((vers[0] == 'Q' && vers[1] == '0') &&
              ((vers[2] == '2' && (vers[3] == '5' || vers[3]
                  == '4' || vers[3] == '3' || vers[3] == '2'
                  ||
                                       vers[3] == '1' || vers[3]
                                           == '0')) ||
               (vers[2] == '1' && (vers[3] == '9' || vers[3]
                   == '8' || vers[3] == '7' || vers[3] == '6'
                   ||
                                       vers[3] == '5' || vers[3]
                                           == '4' || vers[3] == '3
```

```
                                        ' || vers[3] == '2' ||
50                                    vers[3] == '1' || vers[3]
                                        == '0')) ||
51                 (vers[2] == '0' && vers[3] == '9')))
52
53           {
54             fprintf(stderr, "[PROTO_QUIC] %lu mmt_check_quic:
                    FOUND QUIC!\n", ipacket->packet_id);
55             return set_classified_proto(ipacket, index + 1,
                   quic_proto);
56           }
57         }
58       }
59     }
60     else
61     {
62       fprintf(stderr, "[PROTO_QUIC] %lu mmt_check_quic: Not
           QUIC!\n", ipacket->packet_id);
63     }
64   }
65   fprintf(stderr, "[PROTO_QUIC] %lu mmt_check_quic: Not QUIC
       !\n", ipacket->packet_id);
66   return 0;
67 }
```

The registering classification function with parent protocol is done in `init_quic_proto_struct`:

```
1  int init_quic_proto_struct() {
2      protocol_t * protocol_struct =
           init_protocol_struct_for_registration(PROTO_QUIC,
           PROTO_QUIC_ALIAS);
3
4      if (protocol_struct != NULL) {
5
6          int i = 0;
7          for(; i < QUIC_ATTRIBUTES_NB; i ++) {
8              register_attribute_with_protocol(protocol_struct,
                   &quic_attributes_metadata[i]);
9          }
10
11         if(!
               register_classification_function_with_parent_protocol
               (PROTO_UDP, mmt_check_quic, 50)){
12             };
13         return register_protocol(protocol_struct, PROTO_QUIC)
               ;
14     } else {
15         return -1;
16     }
17 }
```

After the registration, the QUIC protocol plugin has been registered as a child protocol of UDP, and for any UDP packet, MMT-DPI will check whether it contains QUIC protocol.

Register classification function to find the next protocol

If the current protocol encapsulates other protocol, then we need to register a classification function to find the next protocol. For example, following is the classification function to find the next protocol after ETHERNET protocol:

```
1  int ethernet_classify_next_proto(ipacket_t * ipacket,
       unsigned index) {
2      int offset = get_packet_offset_at_index(ipacket, index);
3
```

```
4     const struct ethhdr *ethernet = (struct ethhdr *) &
          ipacket->data[offset];
5     classified_proto_t retval;
6     retval.offset = -1;
7     retval.proto_id = -1;
8     retval.status = NonClassified;
9     switch (ntohs(ethernet->h_proto)) // Layer 3 protocol
          identifier
10    {
11            /* IPv4 */
12        case ETH_P_IP:
13            retval.proto_id = PROTO_IP;
14            retval.offset = sizeof (struct ethhdr);
15            retval.status = Classified;
16            break;
17            /* IPv6 */
18        case ETH_P_IPV6:
19            retval.proto_id = PROTO_IPV6;
20            retval.offset = sizeof (struct ethhdr);
21            retval.status = Classified;
22            break;
23            /* ARP */
24            /* RARP: will be processed as ARP */
25        case ETH_P_RARP:
26        case ETH_P_ARP:
27            retval.proto_id = PROTO_ARP;
28            retval.offset = sizeof (struct ethhdr);
29            retval.status = Classified;
30            break;
31            /* 802.1Q */
32        case ETH_P_8021Q:
33            retval.proto_id = PROTO_8021Q;
34            retval.offset = sizeof (struct ethhdr);
35            retval.status = Classified;
36            break;
37        case ETH_P_NDN:
38            retval.proto_id = PROTO_NDN;
39            retval.offset = sizeof (struct ethhdr);
40            retval.status = Classified;
41            break;
42        case 0x9100:
43        case 0x9200:
44        case 0x9300:
45            retval.proto_id = PROTO_8021Q;
46            retval.offset = sizeof (struct ethhdr) + 4;
47            retval.status = Classified;
48            break;
49             /* PPPoE Discovery */
50        case ETH_P_PPPoED:
51            retval.proto_id = PROTO_PPPOE;
52            retval.offset = sizeof (struct ethhdr);
53            retval.status = Classified;
54            break;
55            /* PPPoE Session */
56        case ETH_P_PPPoES:
57            retval.proto_id = PROTO_PPPOE;
58            retval.offset = sizeof (struct ethhdr);
59            retval.status = Classified;
60            break;
61            /* Batman */
62        case ETH_P_BATMAN:
63            retval.proto_id = PROTO_BATMAN;
64            retval.offset = sizeof (struct ethhdr);
65            retval.status = Classified;
66            break;
67            break;
68            break;
69        default:
70            break;
71    }
```

```
72      return set_classified_proto(ipacket, index + 1, retval);
73      //return retval;
74  }
```

The registration need to be done in `init_proto_ethernet_struct`:

```
1   int init_proto_ethernet_struct() {
2       protocol_t * protocol_struct =
            init_protocol_struct_for_registration(PROTO_ETHERNET,
            PROTO_ETHERNET_ALIAS);
3
4       if (protocol_struct != NULL) {
5
6           int i = 0;
7           for(; i < ETHERNET_ATTRIBUTES_NB; i ++) {
8               register_attribute_with_protocol(protocol_struct,
                    &ethernet_attributes_metadata[i]);
9           }
10
11          register_classification_function(protocol_struct,
                ethernet_classify_next_proto);
12
13          // Ethernet is a major encapsulating protocol,
                register it as a stack
14          register_protocol_stack(DLT_EN10MB,
                PROTO_ETHERNET_ALIAS,
                ethernet_stack_classification); //TODO: check the
                return value of this
15          //register_protocol_stack_full(DLT_EN10MB,
                PROTO_ETHERNET_ALIAS,
                ethernet_stack_classification,
                ehternet_stack_internal_cleanup, (void *)
                setup_tcpip_internal_packet(), (void *)
                setup_tcpip_internal_context()); //TODO: check the
                 return value of this
16
17          return register_protocol(protocol_struct,
                PROTO_ETHERNET);
18      } else {
19          return 0;
20      }
21  }
```

Complete the extracting functions

For some attributes which cannot be extracted by generic function, we need to add/update the extrating functions. The following functions are the completed extracting function for QUIC protocol:

```
1   int mmt_connection_id_extraction(const ipacket_t * ipacket,
        unsigned proto_index,
2           attribute_t * extracted_data) {
3           int l4_offset = get_packet_offset_at_index(ipacket,
                proto_index);
4           char * payload =(char*) &ipacket->data[l4_offset +
                sizeof(struct udphdr)];
5           int conn_id = connect_id(payload[0]);
6           *((uint32_t *) extracted_data->data) = conn_id;
7           return 1;
8       }
9
10  int mmt_sequence_extraction(const ipacket_t * ipacket,
        unsigned proto_index,
11          attribute_t * extracted_data) {
12          int l4_offset = get_packet_offset_at_index(ipacket,
                proto_index);
13          char * payload =(char*) &ipacket->data[l4_offset +
                sizeof(struct udphdr)];
14          int seq = sequence(payload);
```

```
        *((uint32_t *) extracted_data->data) = seq;
        return 1;
    }
```

Other functions, variables, and macros...

We also need to add some more functions, variables, and macros if needed.

The following source code is a completed source code for QUIC protocol:

```
/* Generated with MMT Plugin Generator */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "quic_mmt_plugin.h"
#include "extraction_lib.h"
#include "tcpip/mmt_tcpip.h"
#include <netinet/udp.h>


#define QUIC_NO_V_RES_RSV 0xC3  // 1100 0011

#define QUIC_CID_MASK 0x0C      // 0000 1100
#define QUIC_VER_MASK 0x01      // 0000 0001
#define QUIC_SEQ_MASK 0x30      // 0011 0000

#define CID_LEN_8 0x0C          // 0000 1100
#define CID_LEN_4 0x08          // 0000 1000
#define CID_LEN_1 0x04          // 0000 0100
#define CID_LEN_0 0x00          // 0000 0000

#define SEQ_LEN_6 0x30          // 0011 0000
#define SEQ_LEN_4 0x20          // 0010 0000
#define SEQ_LEN_2 0x10          // 0001 0000
#define SEQ_LEN_1 0x00          // 0000 0000

#define SEQ_CONV(ARR) (ARR[0] | ARR[1] | ARR[2] | ARR[3] | ARR[4] | ARR[5] << 8)


classified_proto_t quic_stack_classification(ipacket_t * ipacket) {
  classified_proto_t retval;
  retval.offset = 0;
  retval.proto_id = PROTO_QUIC;
  retval.status = Classified;
  return retval;
}


static int connect_id(char pflags)
{
  u_int cid_len;

  // Check CID length.
  switch (pflags & QUIC_CID_MASK)
  {
  case CID_LEN_8: cid_len = 8; break;
  case CID_LEN_4: cid_len = 4; break;
  case CID_LEN_1: cid_len = 1; break;
  case CID_LEN_0: cid_len = 0; break;
  default:
    return -1;

  }
  // Return offset.
  return cid_len + 1;
}

```

```
59 static int sequence(char *payload)
60 {
61   unsigned char conv[6] = {0};
62   u_int seq_value = -1;
63   int seq_lens;
64   int cid_offs;
65   int i;
66
67   // Search SEQ bytes length.
68   switch (payload[0] & QUIC_SEQ_MASK)
69   {
70   case SEQ_LEN_6: seq_lens = 6; break;
71   case SEQ_LEN_4: seq_lens = 4; break;
72   case SEQ_LEN_2: seq_lens = 2; break;
73   case SEQ_LEN_1: seq_lens = 1; break;
74   default:
75     return -1;
76   }
77   // Retrieve SEQ offset.
78   cid_offs = connect_id(payload[0]);
79
80   if (cid_offs >= 0 && seq_lens > 0)
81   {
82     for (i = 0; i < seq_lens; i++)
83       conv[i] = payload[cid_offs + i];
84
85     seq_value = SEQ_CONV(conv);
86   }
87
88   // Return SEQ dec value;
89   return seq_value;
90 }
91
92 int mmt_check_quic(ipacket_t * ipacket, unsigned index)
93 {
94   printf("[inf] mmt_check_quic: %lu - %d\n", ipacket->
       packet_id, index );
95   int l4_offset = get_packet_offset_at_index(ipacket, index);
96   // int l4_packet_len = ipacket->p_hdr->caplen - l4_offset;
97   struct udphdr * udp = NULL;
98   udp = (struct udphdr *) & ipacket->data[l4_offset];
99   char * payload = (char*) &ipacket->data[l4_offset + sizeof(
       struct udphdr)];
100  int quic_offset = l4_offset + sizeof(struct udphdr);
101  classified_proto_t quic_proto = quic_stack_classification(
       ipacket);
102  quic_proto.offset = quic_offset;
103  int ver_offs;
104  if (udp != NULL) {
105    uint16_t sport = ntohs(udp->source), dport = ntohs(udp->
         dest);
106    // debug("QUIC: Calculating QUIC over UDP");
107    if ((((sport == 80) || (dport == 80) || (sport == 443) ||
         (dport == 443))))
108    {
109      // Settings without version. First check if PUBLIC
           FLAGS & SEQ bytes are 0x0. SEQ must be 1 at least.
110      if ((payload[0] == 0x00 && payload[1] != 0x00) || ((
           payload[0] & QUIC_NO_V_RES_RSV) == 0))
111      {
112        int seq = sequence(payload);
113        fprintf(stderr, "[PROTO_QUIC] %lu sequence: %d!\n",
             ipacket->packet_id, seq);
114        if (seq < 1)
115        {
116          fprintf(stderr, "[PROTO_QUIC] %lu mmt_check_quic:
               Not QUIC!\n", ipacket->packet_id);
117        }
118
119        fprintf(stderr, "[PROTO_QUIC] %lu mmt_check_quic:
```

```
            FOUND QUIC!\n", ipacket->packet_id);
120         return set_classified_proto(ipacket, index + 1,
                quic_proto);
121       }
122
123       // Check if version, than the CID length.
124       else if (payload[0] & QUIC_VER_MASK)
125       {
126         // Skip CID length.
127         ver_offs = connect_id(payload[0]);
128         fprintf(stderr, "[PROTO_QUIC] %lu connection id: %d!\
              n", ipacket->packet_id, ver_offs);
129         if (ver_offs >= 0)
130         {
131           unsigned char vers[] = {payload[ver_offs], payload[
                ver_offs + 1],
132                                   payload[ver_offs + 2],
                                        payload[ver_offs + 3]
133                                  };
134
135           // Version Match.
136           if ((vers[0] == 'Q' && vers[1] == '0') &&
137               ((vers[2] == '2' && (vers[3] == '5' || vers[3]
                    == '4' || vers[3] == '3' || vers[3] == '2'
                    ||
138                                    vers[3] == '1' || vers[3]
                                        == '0')) ||
139                (vers[2] == '1' && (vers[3] == '9' || vers[3]
                    == '8' || vers[3] == '7' || vers[3] == '6'
                    ||
140                                    vers[3] == '5' || vers[3]
                                        == '4' || vers[3] == '3
                                        ' || vers[3] == '2' ||
141                                    vers[3] == '1' || vers[3]
                                        == '0')) ||
142                (vers[2] == '0' && vers[3] == '9')))
143
144           {
145             fprintf(stderr, "[PROTO_QUIC] %lu mmt_check_quic:
                     FOUND QUIC!\n", ipacket->packet_id);
146             return set_classified_proto(ipacket, index + 1,
                    quic_proto);
147           }
148         }
149       }
150     }
151     else
152     {
153       fprintf(stderr, "[PROTO_QUIC] %lu mmt_check_quic: Not
              QUIC!\n", ipacket->packet_id);
154     }
155   }
156   fprintf(stderr, "[PROTO_QUIC] %lu mmt_check_quic: Not QUIC
          !\n", ipacket->packet_id);
157   return 0;
158 }
159
160
161
162 int mmt_connection_id_extraction(const ipacket_t * ipacket,
        unsigned proto_index,
163                                  attribute_t * extracted_data
                                        ) {
164   int l4_offset = get_packet_offset_at_index(ipacket,
          proto_index);
165   char * payload = (char*) &ipacket->data[l4_offset + sizeof(
          struct udphdr)];
166   int conn_id = connect_id(payload[0]);
167   *((uint32_t *) extracted_data->data) = conn_id;
168   return 1;
```

```
169 }
170
171 int mmt_sequence_extraction(const ipacket_t * ipacket,
        unsigned proto_index,
172                             attribute_t * extracted_data) {
173   int l4_offset = get_packet_offset_at_index(ipacket,
          proto_index);
174   char * payload = (char*) &ipacket->data[l4_offset + sizeof(
          struct udphdr)];
175   int seq = sequence(payload);
176   *((uint32_t *) extracted_data->data) = seq;
177   return 1;
178 }
179
180 static attribute_metadata_t quic_attributes_metadata[
        QUIC_ATTRIBUTES_NB] = {
181
182   {QUIC_CONNECTION_ID, QUIC_CONNECTION_ID_ALIAS, MMT_U32_DATA
          , sizeof(uint32_t), 0, SCOPE_PACKET,
          mmt_connection_id_extraction},
183
184   {QUIC_SEQUENCE, QUIC_SEQUENCE_ALIAS, MMT_U32_DATA, sizeof(
          uint32_t), 0, SCOPE_PACKET, mmt_sequence_extraction},
185
186 };
187
188 int init_quic_proto_struct() {
189   protocol_t * protocol_struct =
          init_protocol_struct_for_registration(PROTO_QUIC,
          PROTO_QUIC_ALIAS);
190
191   if (protocol_struct != NULL) {
192
193     int i = 0;
194     for (; i < QUIC_ATTRIBUTES_NB; i ++) {
195       register_attribute_with_protocol(protocol_struct, &
              quic_attributes_metadata[i]);
196     }
197
198     if (!
          register_classification_function_with_parent_protocol(
          PROTO_UDP, mmt_check_quic, 50)) {
199     };
200
201     // register_protocol_stack(PROTO_QUIC, PROTO_QUIC_ALIAS,
          quic_stack_classification);
202     return register_protocol(protocol_struct, PROTO_QUIC);
203   } else {
204     return -1;
205   }
206 }
207
208 #ifndef CORE
209 int init_proto() {
210   return init_quic_proto_struct();
211 }
212 #endif //CORE
```

After completing the source code, make sure you have installed MMT-DPI before compiling the plugin.

5.3.4 Compile new plugin

To compile new plugin, we can use the command line:

```
cd quic_src/
gcc -Wall -fPIC -shared -o libmmt_quic.so -I /opt/mmt/dpi/
    include/ quic_mmt_plugin.c
```

After executing the command, we should have the plugin file : `libmmt_quic.so`

5.3.5 Install and test new plugin

Now you have the plugin file (`.so` file), we are ready to install and test the new plugin.

Install new plugin — To install the plugin, simply copy the .so file into your plugins folder. (by default, the plugin folder locates in `/opt/mmt/plugins/`)

Test new plugin — You can compile and run the file test `proto_attributes_iterator.c` (locates in `/opt/mmt/examples/`), if the plugin successfully installed, we should have something like this:

```
$ ./proto_attributes_iterator
Protocol id 630 --- Name QUIC
    Attribute id 1 --- Name connection_id
    Attribute id 2 --- Name sequence
    Attribute id 4096 --- Name p_hdr
    Attribute id 4097 --- Name p_data
    Attribute id 4098 --- Name p_payload
    Attribute id 4099 --- Name packet_count
    Attribute id 4100 --- Name data_count
    Attribute id 4101 --- Name payload_count
    Attribute id 4102 --- Name ip_frag_packets_count
    Attribute id 4103 --- Name ip_frag_data_volume
    Attribute id 4104 --- Name ip_df_packets_count
    Attribute id 4105 --- Name ip_df_data_volume
    Attribute id 4106 --- Name session_count
    Attribute id 4107 --- Name a_session_count
    Attribute id 4108 --- Name t_session_count
    Attribute id 4109 --- Name first_packet_time
    Attribute id 4110 --- Name last_packet_time
    Attribute id 4111 --- Name stats
..
```

If you have the pcap file which contains the new protocol, you can compile and run the file `extract_all.c` (locates in `/opt/mmt/examples/`), the example will extract all attributes of all protocols.

5.3.6 FAQ

Question: Can we overwrite a plugin?

Answer: Yes and No

It totally depends on how we write the `init_proto()` function.

If you don't want to overwrite plugin, you can check the return of the function `init_quic_proto_struct()`. Otherwise you can ignore the return.

Not allow to overwrite:

```
/* quic_mmt_plugin.c - NOT OVERWRITE*/
#ifndef CORE
        int init_proto() {
            if (!init_quic2_proto_struct()) {
                exit(0);
            }
            return 1;
        }
#endif
```

Allow to overwrite:

```
/* quic_mmt_plugin.c - NOT OVERWRITE*/
#ifndef CORE
        int init_proto() {
            init_quic2_proto_struct();
        }
#endif
```

Question: Do we need to care about the order of plugins (.so files) in plugins/ folder?

Answer: Yes

The plugins will be loaded in alpha-beta order

The plugins should be loaded in dependency ordering

The overwrite plugin should be loaded after the old plugin

Independent plugin can be loaded without concern about the order

Best practice: Alway name your plugin after `libmmt_tcpip.so`

6 Conclusions

In this document we have explained how MMT works. The MMT-DPI library is a commercial library to classify and decode network packets et flows. It has a limited release that is distributed as freeware to be used with the open source version of the MMT-Security library. The goal of Montimage is to show how the MMT-DPI library can be used in combination with security properties to detect functional and security related behaviours and alert incidents. The technique presented in this tutorial can be applied for the analysis of any type of events producing structured information. Theses events can be telecommunication packets (as shown in this document), log entries, application messages, traces etc. The analysis itself can done at runtime or offline using a precaptured trace.

For further information, please send an email to `contact@montimage.com`.