Polymorph: A Real-Time Network Packet Manipulation Framework

Santiago Hernández Ramos
shramos@protonmail.com

April 2018
Special thanks to Lucas Fernández for all the help given in the development of the project.
Contents

1 Introduction 5
2 State of the art 5
3 Introduction to Polymorph 5
4 Polymorph Installation 6
4.1 Download and installation on Linux (Recommended) 6
4.2 Download and installation on Windows 6
4.3 Docker installation 7
5 Polymorph Interfaces 8
6 Interception of the communication 9
6.1 ARP spoofing 9
7 Sniffing of network packets 9
8 Template abstraction 10
8.1 Structure of a template 11
8.2 Template generation 12
8.3 Template dissection 13
8.4 Export Templates 14
8.5 Import Templates 14
9 Intercepting packages 14
9.1 Interception in Linux 14
9.2 Interception in Windows 14
9.3 Templates in the interception process 15
10 Preconditions, Executions and Postconditions 16
10.1 Conditional functions 16
10.2 Precondition 17
10.3 Executions 18
10.4 Postconditions 19
11 Packets modification. Syntax and access methods 19
11.1 Reading the fields of a packet 19
11.2 Insertion of new values in the packets 20
11.3 Own methods of the package 21
11.4 Global variables 21
12 Dynamic dissection of packet fields 22
13 Creating custom layers and fields 23
14 ANNEX 1: Practical case: Modifying MQTT

14.1 Approach of the case ................................................. 26
14.2 Intercepting communication between two nodes in the network 26
14.3 Capturing packets and generating templates .................. 26
14.4 Modifying the Template ........................................... 28
14.5 Modifying the package in real time ...................... 33

15 ANNEX 2: Case of study: Modifying WINREG protocol 35

15.1 Approach of the case ................................................. 35
15.2 Generating the template ........................................... 35
15.3 Modifying the template ........................................... 36
15.4 Adding conditional functions ................................... 40

16 ANNEX 3: All commands and their function 45

16.1 Main interface ..................................................... 46
16.2 Template List interface .......................................... 46
16.3 Template interface ............................................... 47
16.4 Layer interface .................................................... 49
16.5 Field interface .................................................... 50
1 Introduction

The modification of network packets in real time, often called modification "on the air" consists of intercepting the network packets that circulate between two or more machines in the same network, in such a way, that the intercepting machine has the capacity to modify them and forward them in a consistent state and keeping communication between both ends stable. This technique has a large number of applications, ranging from the protection of certain services through the modification of packets that can reveal certain sensitive information, to the modification of packets for security verification of a system, a computer application or a network. In spite of the numerous advantages, its execution presents a high difficulty, which varies in relation to the complexity of the protocols that the packet to be modified implement.

Taking all this into account, this paper proposes a framework for real time network packet modification that presents advantages like, support for a large number of protocols or flexibility and ability to modify network packets at byte level, at the same time that reduces the amount of effort the user must perform to implement this technique.

2 State of the art

Currently there are some tools that allow the user to modify network packets in real time using different techniques.

Tools like MITMF or Bettercap allow the user to modify some packets that implement protocols such as TCP, HTTP or HTTPS through a set of predefined filters. Other tools, such as Hexinject, allow the user to make this modification on another reduced set of protocols by using regular expressions and substitution patterns.

3 Introduction to Polymorph

Polymorph is a framework written in the Python3 programming language that allows the modification of network packets in real time, providing maximum control to the user over the contents of the packet. This framework is intended to provide an effective solution for real-time modification of network packets that implement practically any existing protocol, including private protocols that do not have a public specification. In addition to this, one of its main objectives is to provide the user with the maximum possible control over the contents of the packet and with the ability to perform complex processing on this information.

In the following sections, each of the parts of the framework will be covered, providing an introductory vision of the techniques that the framework imple-
ments and a practical vision of the actions that a user must perform to use this techniques.

4 Polymorph Installation

This section introduces the different installation methods available for the tool. Polymorph is a multiplatform framework, which works on both Windows and Linux. The simplest way to generate a test environment is through the docker deployment (explained at the end of the section). For those who want to have the tool installed in their host operating system, the steps that must be followed are explained below.

4.1 Download and installation on Linux (Recommended)

Polymorph is specially designed to be installed and run on a Linux operating system, such as Kali Linux. Its installation is considerably simple and can be done through the Python content manager, pip3. Before installing the framework, the following requirements must be installed through the package manager corresponding to the operating system that the user is using. The dependencies for a Kali-Linux operating system are specified below.

```
apt-get install build-essential python-dev libnetfilter_queue-dev tshark tcpdump python3-pip wireshark nohup
```

After the installation of the dependencies, the framework itself can be installed with the Python pip package manager in the following way.

```
pip3 install --process-dependency-links polymorph
```

4.2 Download and installation on Windows

Polymorph can also be installed on Windows operating systems. The requirements necessary for the framework to work correctly are the following.

- Installation of Python3 (add it to PATH). URL: [https://www.python.org/downloads/](https://www.python.org/downloads/)
- Installation of Wireshark (add it to PATH). URL: [https://www.wireshark.org/download.html](https://www.wireshark.org/download.html)
- WinPcap installation (If you have not installed it with Wireshark). URL: [https://www.winpcap.org/install/default.htm](https://www.winpcap.org/install/default.htm)

Once the dependencies are installed, the only thing that the user must do is open a console and execute the following command.

```
pip install --process-dependency-links polymorph
```
After completing the installation, Polymorph will be accessible from the terminal from any system path. It is important to note that in Windows, Polymorph must be executed in a console with administrative privileges.

4.3 Docker installation

Docker is a light virtualization system that allows you to create different systems isolated from each other on the host machine. This technology has been popularized in recent years by the facilities it provides in the creation of multiple environments in a versatile, dynamic and lightweight way.

Polymorph has a docker environment in with which you can quickly assemble three machines to test the tool in any operating system in a matter of minutes. The configuration file is written in YAML for the Docker compose tool and consists of the following elements:

- **Polymorph**: Main machine based on Kali Linux, with ip 10.24.0.2. This environment will have the polymorph application that will be accessible from any system path with the command `polymorph`, it has the tools of the top 10 of Kali Linux together with all the necessary packages to make Polymorph work.

- **Alice**: Victim machine with MQTT installed to establish communication with Bob. It has a fixed IP of 10.24.0.3.

- **Bob**: Victim machine with MQTT installed to establish communication with Alice. It has a fixed IP of 10.24.0.4. In addition to all this, Alice and Bob have two aliases to be able to subscribe to an MQTT topic or post a message to the desired IP with `receive` and `send`.

The implementation of this environment is simple, only consists of three steps:

- Download and install Docker on the host machine, to do so go to the Docker homepage and follow the installation instructions for the desired operating system.

- Once the user has downloaded and started docker, the user can access the project in the path `/polymorph` and execute:

  ```
  docker-compose up -d
  ```

  Docker will then take care of creating the containers following the specifications set in the Dockerfile and in the YAML of the compose, as soon as the configuration is finished the three machines will be up and ready to be used. Each time the docker service is restarted, it will be necessary to execute:

  ```
  docker-compose up -d
  ```
to setup the containers again.

- To access any of the machines the user must execute:
  
  ```bash
docker exec -ti [polymorph | alice | bob] bash
  ```

5 Polymorph Interfaces

All the actions detailed in the following sections of the paper are carried out through the Polymorph interface. The framework is formed by a set of interfaces that change dynamically depending on the context in which the user is located. In this way, the following subinterfaces can be distinguished:

- **Main interface**: It corresponds to the first screen that is displayed when the application is executed, at this point, the user is not yet in a certain context. Allows the performance of actions such as spoofing or sniffing. The prompt that is shown is the following:
  
  ```text
  PH >
  ```

- **tlist interface**: It corresponds to the interface that is shown after the completion of the sniffing process, as will be seen in the next section. The user is in the context of a list of templates that is generated from the captured packets. The prompt that is shown is the following:
  
  ```text
  PH: cap >
  ```

- **template interface**: It corresponds to the interface that is shown after the selection of a certain template (more details about it in the following sections). The user is in the context of a template, and may take actions to modify their values. The prompt that is shown is the following:
  
  ```text
  PH: cap/t5 >
  ```

- **Layer interface**: It corresponds to the interface that is displayed after the selection of a layer within a template (more details about it in the following sections). The user is in the context of a layer, and can take actions to modify their values. The prompt that is shown is the following:
  
  ```text
  PH: cap/t5/TCP >
  ```

- **Field interface**: It corresponds to the interface that is displayed after the selection of a field within a layer (more details about it in the following sections). The user is in the context of a field, and can perform actions to modify their values. The prompt that is shown is the following:
  
  ```text
  PH: cap/t5/TCP/sport >
  ```
6 Interception of the communication

One of the most important conditions that must be met to be able to modify network packets in real time is to be in the middle of the communication between two machines. The packets that flow from one end of the other must flow through the machine where Polymorph is installed, in such a way, that the tool must be able to intercept these packets, process them and forward them.

There are numerous techniques to intercept the communication between two machines that are in the same network. The framework implements some of them, in a certain way that they are very simple to execute by the user. Below are some of these techniques.

6.1 ARP spoofing

This technique consists of sending ARP (Address Resolution Protocol) packets generated by an attacker in a local network. The objective is to associate the MAC address of the attacker’s machine with the IP address of another machine on the network, such as the default gateway, causing all traffic to that IP address to pass in its place by the attacker’s machine.

The realization of this technique with Polymorph is relatively simple. From any of the interfaces of the tool we can use the command `spoof` to carry it out in the following way.

```
PH > spoof -t 192.168.1.50 -g 192.168.1.1 -i eth0
```

The options of the `spoof` command are the following:

Usage: `spoof -t <targets> -g <gateway>`

Options:
- `-h` prints the help.
- `-t` targets to perform the ARP spoofing. Separated by ",".
- `-g` gateway to perform the ARP spoofing.
- `-i` network interface.

The mandatory parameters for the realization of the ARP spoofing are the `gateway (-g)` and at least one target ip address `target (-t)`. If not interface is specified, Polymorph will automatically get the one that is currently in use.

7 Sniffing of network packets

The sniffing process, in this case, corresponds to the capture of the network packets that flow between two machines, without making any modification on them.
Once the interception of the communication between two nodes of the network has been made, the next step is to start doing sniffing of the network packets that flow through the attacker machine. The sniffing process must be maintained until it is captured one of the packets that corresponds to the type of packet that the user wants to modify. This process is done from the main interface of the framework using the command `capture`. This command has the following options:

**Usage:** capture [-option]

**Options:**
- `-h` prints the help.
- `-f` allows packet filtering using the BPF notation.
- `-c` number of packets to capture.
- `-t` stop sniffing after a given time.
- `-file` read a .pcap file from disk.
- `-v` verbosity level medium.
- `-vv` verbosity level high.

After performing the sniffing process, Polymorph will convert the captured packets to a particular structure of the framework called `template`. The template corresponds to the main abstraction of the whole system, in the following section its details are explained in depth.

## 8 Template abstraction

Traditionally, there have been tools that allowed us to craft network packets by specifying the characteristics of their components, such as the protocols they implement, the layers they have or the size of their fields. This way of defining a package, although effective, is extremely expensive and consumes a large amount of time. To solve this, Polymorph implements the template concept, which corresponds to one of the most important of all the framework and that provides the following essential characteristics:

- Allows users to access packages that are intercepted in real time in a simple manner
- Allows the creation of new features within a packet in a quickly and easily way
- Allow a certain configuration of a Polymorph session to be stored on disk and exchange it between environments and users

In the next subsections, different features and applications of the template concept within the framework are covered.
### 8.1 Structure of a template

All the templates have the same structure, which corresponds to a `.json` file when they are exported from the framework.

```json
{
    "Name": "Template:ETHER/IP/UDP/DNS",
    "Description": "",
    "Version": "0.1",
    "Timestamp": "2018-04-06 05:13:14.232177",
    "Functions": {
        "preconditions": {},
        "executions": {},
        "postconditions": {}
    },
    "layers": [
        {
            "name": ,
            "custom": ,
            "lSlice": ,
            "structs": {},
            "fields": [
                {
                    "name": "",
                    "value": "",
                    "type": "",
                    "size": "",
                    "slice": "",
                    "frepr": "",
                    "custom": ""
                },
                {
                    "name": "",
                    "value": "",
                    "type": "",
                    "size": "",
                    "slice": "",
                    "frepr": "",
                    "custom": ""
                },
                {
                    "name": "",
                    "value": "",
                    "type": "",
                    "size": "",
                    "slice": "",
                    "frepr": "",
                    "custom": ""
                }
            ],
            "raw": ""
        }
    ],
}
```

(Some of the fields that appear in the template structure are hard to understand at this point of the paper, but they will be specified in detail later.)

- **Name**: Name of the template
- **Description**: Description of the template
- **Version**: Template version
- **Timestamp**: Creation date of the template
- **Functions**: A set of functions that the user can define to perform advanced processing on packets that are intercepted in real time
8.2 Template generation

After performing the sniffing process, the captured packets are automatically converted into templates by the framework. The transformed packets would look similar to the following:

```
---[ ETHER ]---
hex dst = 005056e6721b (00:50:56:e6:72:1b)
hex src = 000c299909fa (00:0c:29:99:09:fa)
int type = 2048 (2048)

---[ IP ]---
int version = 69 (4)
int ihl = 69 (5)
int tos = 0 (0)
int len = 58 (58)
int id = 18907 (18907)
...
The objective is to capture a packet that implements the protocols that the user is interested in modifying in real time. Polymorph will transform this packets into templates in which the layers, fields, value of the fields and the rest of the values described above will be stored. From this moment all the modifications made on that packets with Polymorph, will be done on the generated templates. The templates will later allow the user to modify network packets in real time.

8.3 Template dissection

One of the biggest problems that real-time package modification tools have today is the ability to dissect the protocols that implement the captured packets. This translates into a great limitation when modifying them since the structure of the packet bytes can not be determined.

To solve this problem and try to interpret as much as possible of the existing protocols in the packets that are captured, Polymorph performs the following process:

- Capture the packets bytes using the sniffing process
- Use the Tshark dissectors, which is probably the tool that more protocols is able to interpret in the world, to dissect these bytes
- Build a template with the dissected fields, the position that the field occupies within the set of bytes of the packet, the layer to which it belongs and other added values.

Using this technique, Polymorph is able to interpret a high number of network protocols, as many as tools such as Wireshark can interpret, with the difference that this interpretation will be used later to modify similar packages in real time.

Dissection with Polymorph is done using the command `dissect` in the interface corresponding to the list of templates, which is accessed immediately after performing the process of sniffing in the main interface. If the user does not perform the dissection process and directly accesses a template, the framework will automatically dissect the selected template and all the previous ones, in such a way that, if the user selects the template 15 without having previously executed the command `dissect`, Polymorph will automatically dissect the first 15 templates and then, access the template 15.
8.4 Export Templates

The templates can be exported in a very simple way from the framework. The result is a file with extension `.json`. This process can be carried out using the command `save` in the template interface.

8.5 Import Templates

The templates can be easily imported into the framework so that the user can work on a template previously saved on disk. Assuming that a template has been previously stored on disk, the user can execute Polymorph in the following way to start locking in the context of an existing template:

```
python3 polymorph -t "path_to_template"
```

9 Intercepting packages

Once the communication between two nodes of the network has been intercepted, the network packets exchanged between them flow through the intercepting machine. At this point, it is necessary to implement a technique to be able to move the packets from kernel space to user space in which they will be processed by Polymorph.

9.1 Interception in Linux

In the case of Linux operating systems, the Netfilter suite is used to perform this task. The tool uses a module called Nfqueue that will be responsible for moving the packages from Kernel space to user space through a queue from where Polymorph will consume them. To redirect the packets, another well-known module of this suite, Iptables, is used.

9.2 Interception in Windows

In the case of Windows operating systems, a module called Windivert is used. This module allows things like:

- Capture network packets
- Filter or remove network packets
- Inject network packets
- Modify network packages
9.3 Templates in the interception process

It is important to understand the role of templates in the process of interception of packages. When the user executes the order to intercept packages in Polymorph, it does so in the context of a template. This means that for each of the packets that are intercepted and arrive at the tool, a projection of the template is performed on its bytes, in this way, we can access the fields of the new package by means of the specific syntax of the template that it is being used to intercept.

To understand this better, the following scenario is presented:

- The user wants to access the source port field of a packet with the layers, Ethernet, IP, TCP
- A sniffing process is performed with the tool to capture a packet with these characteristics
- Once captured, the packet is automatically converted into a template and the user uses Polymorph to access the context of this template using the template command

Once placed in the template interface, the user can use the show command to display each of the layers and fields of the template, if the TCP layer is accessed (where the source port is found) using the command layer and the command show is entered again, something similar to the following should appear:

```
--- [ TCP ] ---
int sport = 39440 (39440)
int dport = 443 (443)
int seq = 2693320198 (2693320198)
int ack = 2991926589 (2991926589)
int dataofs = 20500 (5)
int reserved = 20500 (0)
str flags = P (RA)
int window = 40880 (40880)
int chksum = 18239 (18239)
int urgptr = 0 (0)
str options = ([]) 
```

Here you can see several things, on the one hand, you have the name of the fields of the layer, along with an important characteristic of them, their type. All fields can be of 4 different types, int, str, hex, bytes depending on the value they have. These types can be modified by the user. Accessing the source port field using the command field sport, and entering the command show again, will display different characteristics of the field:

```
--- [ sport ] ---
value = 39440 (39440)
bytes = b'\xa\x07'
```
hex = 9a10
size = 2
slice = [34, 36]
custom = False
type = int

In this representation of the field you can see its value in different formats, its size, its type, and what is important, the position that the field occupies within the packet bytes. The command dump displays the position of the field in the bytes of the packet:

```
00000000: 00 50 56 E6 72 1B 00 0C 29 99 09 FA 08 00 45 00
00000010: 00 28 87 FD 40 00 40 06 D3 DF C0 A8 73 81 D8 3A
00000020: D2 8E 9A 10 01 BB A0 88 CE 06 B2 55 2D 3D 50 14
00000030: 9F B0 47 3F 00 00
```

If at this point you want to start intercepting packets, you must return to the template context (by going back with the back command to the layer interface and then to the template interface), in this context you can use the command intercept to begin intercepting the packets that pass through the intercepting machine in the context of that template. If at this point you would like to access the source port field of all the packets that arrived at the machine, the user could do it with the syntax package ['TCP']['sport'], Polymorph will extract the position that this field occupies within the template used as context and also the type that the field has, and after that, the framework will dissect the bytes of the packets that arrive by extracting those sequence of bytes that correspond to the field sport of the template (in this case the 34, 35 and 36) and making an interpretation of them according to the type of the field in the template, in this case int.

## 10 Preconditions, Executions and Postconditions

Once the communication between two nodes of the network has been intercepted and the packets passing through the intercepting machine are being intercepted by the techniques described in the previous section, it is time to modify the packets in real time. This is one of the most relevant sections of the paper, and presents the abstractions that allow users to perform complex processing on any network packet that implements any protocol in a relatively simple way.

### 10.1 Conditional functions

The conditional functions are the framework abstraction that will allow the user to add code in the Python programming language that will be executed in real
time on the packets that are intercepted.

There are three types of conditional functions, preconditions, executions and postconditions. The separation between the three types is logical, which means that at the technical level there are no differences between them and what they intend is to provide a degree of order and separation between the different processes that may be required on a package.

The conditional functions will be executed sequentially and following the order in which they were added, this means that the first to be executed will be the preconditions, starting with the first precondition added by the user, then the executions and finally the postconditions. The user can add as many conditional functions as he needs. All the functions have the following structure:

```python
def new_prec(packet):
    # your code here
    # If the condition is meet
    return packet
```

As can be seen, the structure is very simple. On the one hand, a function must be defined in python, which can have any name and which must receive a parameter as an argument. This parameter corresponds to the packet that the framework is intercepting in real time at that moment. Inside the function are the instructions written by the user, this instructions can perform some processing on the packet, or some generic processing such as displaying text on the screen. After processing, the user has two possibilities, to return the packet variable, which implies that the following conditional functions continue to be executed, or to return `None`, which implies that the packet is forwarded with the processing done up to that moment without execute more conditional functions.

The conditional functions are stored in the template, and are exported serialized along with it. The objective of each of the types of conditional functions is defined in depth below.

### 10.2 Precondition

When the user starts intercepting packets in real time with Polymorph, the framework will probably intercept many more packets than the user wants to modify. The objective of the preconditions is to provide a previous step of filtering the packets until they have the desired packet. As explained in previous sections, when packets are intercepted in real time, it is done in the context of a template, which allows access in a simple way to the fields of the intercepted packets without knowing in advance the protocols or structure that the packet implements. The following is a use case in which the user wants to modify in real time packages that implement the ICMP protocol.
Since we want to modify ICMP protocol packets, previously, packets of this protocol must have been captured and transformed into a template, so that we are working on a template similar to the packets that we want to modify. In this case, it should be something similar to the following:

```
---[ ETHER ]---
hex dst = 005056e6721b (00:50:56:e6:72:1b)
...

---[ IP ]---
...
int proto = 1 (1)
int chksum = 34610 (34610)
hex src = c0a87385 (192.168.115.133)
hex dst = c0a80101 (192.168.1.1)
hex options = c0a80101 ([]) 

---[ ICMP ]---
int type = 8 (8)
int code = 0 (0)
int chksum = 14180 (14180)
int id = 3997 (3997)
int seq = 2 (2)
...
```

As you can see in the template, within the IP layer there is a field called `proto` that tells us the type of protocol in use. In this case it is type `int` and has the value 1. Our precondition would be something similar to the following:

```python
def new_prec(packet):
    try:
        if packet["IP"]['proto'] == 1:
            return packet
    except:
        return None
```

With a precondition as simple as the one shown, all the packets belonging to the ICMP protocol will be filtered, allowing only those that fulfill our precondition to continue executing the rest of the conditions that have been defined, and those that do not comply with it will be immediately forwarded.

### 10.3 Executions

As explained in previous sections, the difference between the conditional functions is logical, therefore, the executions work in the same way as the preconditions. The difference between both is at the organizational or functionality
level, the executions are designed to add functions that perform processing on
the intercepted packets, an example of execution could be the following:

```python
def new_exec(packet):
    packet["IP"]["len"] = 56
    return packet
```

In this execution we would be modifying in real time the length field of the
IP layer of the packets that arrive and comply with our defined preconditions.

### 10.4 Postconditions

Postconditions, like executions, work in the same way as preconditions, but
their purpose is different. When modifying packets at the byte level, it is very
common that after the modification some control fields of the packet, such
as checksums or lengths become inconsistent. The purpose of this group of
functions is to process the packet to be completely consistent before sending it
to the destination machine. An example of a postcondition to recalculate the
control fields of the TCP/IP layers is the following one:

```python
def recalculate_tcp_ip(packet):
    from scapy.all import IP
    pkt = IP(packet.raw[14:])
    if pkt.haslayer('IP') and pkt.haslayer('TCP'):
        del pkt['IP'].chksum
        del pkt['TCP'].chksum
        del pkt['IP'].len
        pkt.show2()
        packet.raw = bytes(pkt)
    return packet
```

As you can see, in the set of conditional functions you can use other frame-
works or tools to perform certain functions, this gives Polymorph more power
and flexibility. In this case, Scapy is used to recalculate the control fields.

### 11 Packets modification. Syntax and access meth-
ods

Once that have been presented the different methods to add custom functions
(preconditions, executions and postconditions) to a template that will be exe-
cuted in real time on the packets that are intercepted, it is time to explain the
syntax that can be used to build these conditional functions and modify the
packets in real time.

#### 11.1 Reading the fields of a packet

As it has been explained several times throughout the paper, it is important to
remember that the interception is done in the context of a template, this means
that when you access a certain field of a packet that has been intercepted, you are checking the position that the field occupies in the template and dissecting that same position in the bytes of the intercepted packet. After this, the bytes that have been extracted from the intercepted packet, are converted to the type that the field has in the template (more information about this in section 10, subsection 10.3).

Taking all this into account, the following is the syntax that should be used to access the fields of a particular packet that has been intercepted based on the template that is being used as context in the intercepting process.

Assuming you want to access the `chksum` field of the IP layer of all the packets that the framework is intercepting in real time, and assuming that the template that the user has generated and is using as intercept context has the IP layer:

```
---[ ETHER ]---
hex dst = 005056e6721b (00:50:56:e6:72:1b)
...
---[ IP ]---
...
int proto = 1 (1)
int chksum = 34610 (34610)
hex src = c0a87385 (192.168.115.133)
hex dst = c0a80101 (192.168.1.1)
hex options = c0a80101 ([])
```

The syntax to access this field would be the following:

```
packet ["IP"["chksum"]
```

First the user must specify the layer in square brackets and finally the field that the user wants to access in square brackets. In the next example we will use this syntax within a precondition that displays all the `chksums` of the intercepted packets:

```
def new_prec(packet):
    try:
        print(packet ["IP"["chksum"])
    except:
        return None
```

### 11.2 Insertion of new values in the packets

The insertion of new values in the intercepted packets is similar to the query of a value. If you want to add a new value for the `chksum` field of the IP layer, as shown in the previous section, it can be carried out using the following syntax:
One of the most important things that should be taken into account when adding a new value, is the type that field has in the template that is being used as intercepting context. If we go back to the template showed in the previous section, the field `chksum`, has type `int`, which means that the user must assign a value of type `int` when inserting. The type of the field in the template can be modified by the user prior to the interception process.

11.3 Own methods of the package

In addition to accessing all the fields that are specified in the template that is being used as the interception context and the assignment of new values to them, Polymorph provides a series of simple methods that can be applied to the packets that are captured and whose objective is to facilitate the calculation of certain lengths or the assignment of new values to the packet.

- **raw**: It is a property of the packet and returns its bytes. It is useful if the user uses other frameworks that modify the bytes of the packet. It can be used as follows:

```python
def new_prec(packet):
    # Imprimimos por pantalla los bytes del paquete actual
    print(packet.raw)
    # asignamos nuevo valor al paquete
    packet.raw = b"\x00"
    return packet
```

- **len**: Provides the total length of the packet or the different layers of the packet. It can be used as follows:

```python
def new_prec(packet):
    # Imprime la longitud del paquete
    print(len(packet))
    # Imprime la longitud de la capa IP
    print(len(packet["IP"]))
    return packet
```

11.4 Global variables

There are certain occasions in which it is required to maintain global variables that are persistent between conditional functions and between intercepted packets. Polymorph allows the option to define a global variable whose value will be maintained for all packets that are intercepted. The way to define these variables is as follows:

```python
def create_global_vars(packet):
    try:
```
packet.insert
except AttributeError:
    setattr(packet, 'insert', False)

This will create a variable called insert that will be maintained even though the current packet is forwarded and can be accessed through the packet.insert statement. This can be useful if the user wants to start executing a certain execution function after a certain event occurs. For example, if a certain packet is received, the variable insert is set to True and from that moment all the packets that are intercepted are modified.

12 Dynamic dissection of packet fields

At this point of paper we have already explained many of the key concepts of the framework that is presented. In this section we present another abstraction that will allow the user to dissect fields of different lengths of a packet intercepted in real time, the concept of Struct.

If the previous sections of paper have been followed, and the concept of template has been understood along with the relation it has with the interception and modification of the packets in real time, probably someone has noticed the following casuistry:

- Assuming that the user has generated a template similar to the following, with the aim of modifying packets that implement the MQTT protocol:

  ```
  ---[ ETHER ]---
  ...
  ---[ IPV6 ]---
  ...
  ---[ TCP ]---
  ...
  ---[ RAW ]---
  ...
  ---[ RAW.MQTT ]---
  str hdrflags = 0 (0x00000030)
  int msgtype = 48 (3)
  int dupflag = 48 (0)
  int qos = 48 (0)
  int retain = 48 (0)
  int len = 10 (10)
  int topic_len = 4 (4)
  str topic = test (test)
  str msg = hola (hola)
  ```

- If the user would like to display the msg field of the RAW.MQTT layer of
all the packets that are intercepted, the necessary precondition function would be something similar to the following:

def new_prec(packet):
    try:
        print(packet["RAW.MQTT"]["msg"])
    except:
        return None

• The way in which Polymorph will perform this action, is looking for the position that the field msg occupies in the template, in this case, from byte 94 to 98, and it will try to dissect these four bytes of the intercepted packets and transform them to the type the field has in the template, in this case str.

• As you can see, there is a problem if the msg field is a variable length field, so if the intercepting packet has a value in the msg field equal to Hello how are you, the framework, will only dissect the first four bytes (the ones that occupy this field in the template), and will display the value Hell omitting o how are you.

To avoid this problem, polymorph implements the concept of Struct, that allows the user to indicate the size of a field based on the size of other fields in the template. The Structs will be stored in the template when it is exported.

These structures are defined from the framework itself in the context of a given layer, and have two main components:

• **Start byte**: The start byte of the field that you want to recalculate dynamically

• **Expression**: The expression used to calculate the new length of the field, can be simple or complex

For the use case that was shown in the previous example, the msg field of the RAW.MQTT layer can be recalculated dynamically, defining the following Struct in Polymorph:

PH: capt14/RAW.MQTT > recalculate -f msg -sb "70 + this.topic_len" -e "this.len - 2 - this.topic_len"

It is important to emphasize that to refer to fields of the layer itself in both the start byte(-sb) and the expression(-e) the prefix this. must precede the field.

13 Creating custom layers and fields

To finish with the concepts implemented in the framework, a characteristic is presented that allows the user to modify the templates to add their own layers
As noted throughout the paper when packets are intercepted and modified, it is done in the context of a template, and will be the template fields and the position they occupy in the packet bytes the attributes that will allow the dissection of the new packets that are intercepted. In some cases, it is possible that Polymorph is not able to properly dissect some bytes of the packet, either because the protocol is not correctly defined, or because it is a private protocol without public specification. For these cases, Polymorph allows the user to manually create layers and fields, so that in the interception phases you can access them in a simple way as shown in previous sections.

The process of adding layers and fields is carried out from the template and layer interface and is a simple process. The command that is used to add a new layer is the following:

```
PH: cap/t14 > layer -a NEW LAYER
```

Start byte of the custom layer:
Polymorph will show all the bytes of the packet and request two values:

- **Start byte**: The start byte of the layer
- **End byte**: The byte in which the layer ends

Once the user has entered these values, the new layer is added to the template, and in the same way, you can begin to define fields by means of the statement:

```
PH: cap/t14/NEW LAYER > field -a newfield
```

Start byte of the custom field:
Indicating in this case,
- **Start byte**: The beginning byte of the field
- **End byte**: The byte in which the field ends
- **type**: The type of the field you want to create (int, hex, bytes, str)

From this point, the user can begin to modify the value of the layer and the field by accessing it through the different interfaces of the framework.
14 ANNEX 1: Practical case: Modifying MQTT

14.1 Approach of the case

It is intended to perform the modification in real time of network packets belonging to the MQTT protocol. Specifically, it is intended to modify the message that is carried in the packets of type *MQTT publish*.

14.2 Intercepting communication between two nodes in the network

The first thing that the user must do is to intercept the communication between two nodes of the network that will be communicated through the MQTT protocol. To perform this step, the user can use the *spoof* command from the Polymorph main interface.

![Image of MQTT ARP Spoofing](image1.png)

**Figure 1: MQTT ARP Spoofing**

To check if the poisoning is taking effect, you can access one of the poisoned machines and make a *traceroute* to the other legitimate machine. The communication must flow through the attacking machine.

![Image of Testing the ARP Spoofing](image2.png)

**Figure 2: Testing the ARP Spoofing**

14.3 Capturing packets and generating templates

Once the communication between the legitimate machines has been intercepted, the next step is the capture of packets belonging to the MQTT protocol, among which a packet *MQTT publish* must be found. Once these packets are captured,
Polymorph will automatically convert them into templates. You can start capturing packets as shown in figure 3.

![Figure 3: Sniffing with Polymorph](image)

Figure 3: Sniffing with Polymorph

After stopping the capture process, it can be seen that the framework has automatically entered the interface where a list of templates is displayed, with the command `show` we can see the generated templates. Figure 4 shows the execution of the command.

![Figure 4: Show template list](image)

Figure 4: Show template list
It is important to understand that at this point, the generated templates correspond to each of the packets captured in the sniffing process, and their representation in relation to the protocols implemented by each of the captured packets is the one that Scapy provides. If you look closely, Scapy has not been able to dissect the MQTT layer, therefore, the user must use the command *dissect*, to use more advanced dissectors on the bytes of these packets, that will finish interpreting all their layers. Figure 5 shows the execution of the command.

![Dissection of the Templates](image)

Figure 5: Dissection of the Templates

Once more advanced dissectors have been used to interpret all the layers of the packets, you can see how templates appear with the MQTT layer, to access one of the generated templates and check if it belongs to a *MQTT publish* packet, the user must use the command *template* with the number of the template he want to access. Figure 6 shows the execution of the command.

If you have captured a large number of templates, and you need to perform an exhaustive search among all generated templates, you can use the *wireshark* command, which will open the capture with Wireshark in *.pcap* format, so that the user can quickly select the templates that he wants to modify.

### 14.4 Modifying the Template

Once a template with the MQTT layer has been selected, the framework automatically redirects the user to the Template interface. To show the contents of
the selected template, use the command `show`. Figure 7 shows the execution of
the command.

It is important to stop for a moment in this phase to understand the values
that this command shows on the screen. On the left, we can see all the layers
of the template, below each layer, we can see all the fields belonging to each
layer preceded by the type that the field has in the template. The type, as will
be seen later, is a very important feature when writing the conditional functions
of the template. On the right, there is, in white, the value that the field has
in the template and in blue, an orientative value that the dissectors have taken
from the field. It is important to realize that the value with which the user must
operate is the one that is in white color.

If you pay attention to the fields of the RAW.MQTT layer that appear in
figure 7, you can locate a field called `msgtype`, this is the field that determines
the type of MQTT package, that in this case, corresponds to a 48 (MQTT Pub-
lish). As can be seen, the field has a small difference between the value that
the template has and the value produced by the dissectors, this is because Poly-
morph has interpreted the complete byte and converted it to integer, while the
value 3 originated by the dissectors, is extracted only from the first 4 bits of the
byte, and is the one that actually represents the type of message. If you want
to check the value of the field in more detail, you can access the layer with the
command `layer raw.mqtt` and the field with the command `field msgtype`, with
the command `show` you can see the characteristics of the field.

Once a field has been determined within the template that uniquely identifies
Figure 7: Show command

the type of packets that we want to modify, the user can write a precondition so that, at the moment of interception, all the packets are filtered and only the desired packets are obtained. A precondition that would filter the packets by
the msgtype field could be the following:

```python
def new_preq(packet):
    try:
        if packet['RAW.MQTT']['msgtype'] == 48:
            return packet
    except:
        return None
```

In the precondition we access the msgtype field of the packets that are intercepted in real time by the framework, and it is checked if the value that is in that position in the bytes of the intercepted packet corresponds to a 48. If this is the case the framework continue executing the conditional functions (return packet), otherwise the execution of the functions is broken and the packet is forwarded (return None). It is important to take into account the type that the field has in the template when comparing it with a new value. In this example, the field is of type int and therefore in the conditional functions it must be compared or assigned with values of type int. The type of the field in the template can be modified from the field interface. Figure 8 shows the command needed to add the precondition.

![Figure 8: Add precondition](image)

Once the precondition that will filter the MQTT Publish packets is added, we are going to add a simple execution function, which shows on the screen the msg field of the intercepted packets. The execution function is the following:

```python
def new_exec(packet):
    print(packet['RAW.MQTT']['msg'])
    return packet
```

![Figure 9: Add execution](image)

After adding the execution function, the user can begin to intercept packets in real time, these packets will be filtered with the precondition and will go to
the execution, where the field msg will be shown on the screen as it is described in the template which is used as an interception context. To start intercepting packets, the user can use the command intercept as shown in image 10.

```plaintext
PH:cap/t8 > intercept
[*] Waiting for packets...

hola
hhhh
adio
buen
```

Figure 10: Intercept

If messages start to be published among the legitimate machines, we will observe how the framework starts capturing the packets, filtering the MQTT Publish packets and displaying the msg field.

Although everything seems correct, it is easy to notice that the values that appear on the screen, in certain occasions, are cut off and only the first 4 letters of the message are shown. This is because in the template that is being used as the intercept context, the field msg has a length of 4 bytes, and therefore in the incoming packets the framework only dissects those 4 bytes. If the user would like to dissect the field msg dynamically in function of the control fields of the layer RAW.MQTT, it should add a Struct, and this is done in the following way (figure 11).

```plaintext
PH:cap/t8 > layer raw.mqtt
PH:cap/t8/RAW.MQTT > recalculate -t msg -sb "70 + this.topic_len" -e "this.len - 2 - this.topic_len"
[*] Struct added to field msg

PH:cap/t8/RAW.MQTT > b "hola"

PH:cap/t8/RAW.MQTT > b
PH:cap/t8 > intercept
[*] Waiting for packets...

buenos
buenos dias
hola que tal estas
```

Figure 11: Recalculate field

With this statement the user tells Polymorph that the msg field will have a variable length. The first thing that is indicated is the start byte (-sb), which will be the byte where the field will begin within the set of bytes of the packet,
finally it must be indicated an expression $(-e)$ which will indicate how the length of the field should be calculated. To refer to the content of fields within the template, the user must use the prefix `this`.

If after adding the `Struct`, we return to intercept packets, we can see how this time the dissection of the `msg` field is done dynamically and the messages sent from one legitimate machine to the other appears on the screen.

### 14.5 Modifying the package in real time

Once the user has added a precondition to filter the `MQTT Publish` packages and a function to recalculate the `msg` field dynamically, modifying this field is a simple task. The only thing that the user must do is add a set of executions and postconditions to make sure that after inserting the value, the package remains consistent.

In this case, if we modify the value of the `msg` field of the RAW.MQTT layer, these would be the control fields that would be inconsistent:

- `packet['IP']['len']`
- `packet['IP']['chksum']`
- `packet['TCP']['chksum']`
- `packet['RAW.MQTT']['len']`

The following postconditions (which may be reused for other use cases) re-calculate those fields.

```python
def mqtt_len_rec(packet):
    packet['RAW.MQTT']['len'] += packet['RAW.MQTT']['topic_len'] + 2 + len('attacker value')
    return packet

def recalculate_tcp_ip(packet):
    from scapy.all import IP
    pkt = IP(packet.raw[14:])
    if pkt.haslayer('IP') and pkt.haslayer('TCP'):
        del pkt['IP'].chksum
        del pkt['TCP'].chksum
        del pkt['IP'].len
        pkt.show2()
        packet.raw = bytes(pkt)
    return packet
```

So, adding the precondition for filtering `MQTT Publish` packets, an execution to insert a new value in the `msg` field of the RAW.MQTT layer:
def new_exec(packet):
    packet ['RAW.MQTT'] ['len'] = 'attacker value'

And the postconditions cited above, we could begin modifying MQTT protocol packets in real time (figure 11).

Figure 12: Inserting value
15 ANNEX 2: Case of study: Modifying WIN-REG protocol

15.1 Approach of the case

In this case study, it is intended to perform real-time modification of network packets belonging to the Windows remote registry protocol. Specifically, we intend to modify the `setvalue` packet, in order to introduce a value modified by the attacker in the registry of a victim’s machine. The packets that implement the Remote Registry Protocol have the protocol stack shown in Figure 13, which, a priori, makes its modification difficult.

To configure the environment, you can follow the following video: https://www.youtube.com/watch?v=fzkeEJG7l4Q

![Figure 13: Winreg packet](Figure 13: Winreg packet)

15.2 Generating the template

As in the previous case, in this case, the user needs to intercept the communication between the two machines (Windows) that communicate with the Remote Registry Protocol using one of the techniques offered by the tool (or other techniques that implement other frameworks). The objective of this step is to capture a package of type `setvalue` belonging to the Remote Registry Protocol and generate a template. With this template, different processing will be performed and packets will be intercepted and modified in real time. Figure 14 shows the steps to follow to capture packets and generate the list of templates.
As can be seen in Figure 14, the list of templates can be very long. To speed up the process of filtering the template that interests us, we will use the command *wireshark*, which will open the Wireshark tool in another window, where we can apply filters to find the corresponding packet number with the message *setvalue*. Figure 15 shows the process.

### 15.3 Modifying the template

Once we have located the packet number in wireshark, we access the template using the command *template* (it is important to keep in mind that the number
in the framework is equal to the number of wireshark - 1). Once inside the template interface, we use the `show` command to display its contents on the screen. Figure 16 shows the contents.

At this point we can observe how the packets of this protocol are complex and implement several protocols that Polymorph has been able to dissect. However, if we look closely, we can see how in the RAW.WINREG layer there is no value field in which the message sent by a legitimate user to the other legitimate user appears. If we use the `dump` command as shown in figure 17, we can see in what position within the packet bytes this message is found.

If we go back to the capture in Wirehsark, we can see how the package, in the `Winreg` layer, does not have a specific field where the value that is sent in the message is specified, the tshark dissectors do not generate it, and that’s why it does not appear in Polymorph. Since this value will be modified in this use case, we will create a new field in the RAW.WINREG layer with the value of the user. Figure 18 shows the process.

In the figure we can see how a new field is added. First, we must be in the context (interface) of the layer where we want to add the new field. Once there we execute the command shown in figure 18 and we will request the start and end bytes of the field, and the type of the field, we can check these values in the `dump` that is shown, or in the capture that we had open in Wireshark. After performing this process, if we use the command `show` in the context of the layer, we can observe the new field created with the value of the user. Figure 19 shows the result of the command.

After adding the new field, what we are going to do is build a `Struct`, in such a way that the field is recalculated dynamically when we are capturing depending on other fields of the layer. When we add a `Struct` it is necessary to take into account that the field that is used to recalculate the length must be before the actual field that is being recalculated. If we look at Wireshark, we can see that there is a length field just before the message begins. Polymorph
Figure 16: SetValue Template

```
| Figure 17: Dump of the Template bytes |

Figure 17: Dump of the Template bytes

```

38
has not dissected this field correctly, so let’s create it. In figure 20 its creation is shown.

If we access the interface of the new field created and show its value with the command `show`, we can see how the value that appears does not match the one shown by Wireshark or the dissectors, this is because polymorph interprets
Figure 20: Add Size field

by default all int values as big endian, and in this case the Windows Remote Registry protocol introduces them in the package as little endian, to modify the field type, the only thing that we have to do is use the sentence shown in figure 21.

Once we have found and interpreted the control field that determines the length of the value field that we have previously added, we are going to add the Struct to recalculate this field dynamically. Figure 22 shows the script.

In addition to creating the Struct, we can use the -t option to verify that it is correctly formed. The result should be the user’s field value.

To finish, and before starting to write the conditional functions, we are going to convert another field to little endian, the opnum field of the RAW.DCERPC layer. This field will be used to write the precondition that filters the setValue protocol packages. Figure 23 shows the script.

Before starting with the next section, we will save the template in case any problem occurs that causes an unexpected shutdown of the application. Figure 24 shows the necessary commands.

15.4 Adding conditional functions

Once we have the template fields that interest us correctly interpreted and we have also saved it in disk in case an unexpected event happens, we will start with the conditional functions that will process the packets in the air.

We are going to start by adding a simple precondition that filters all the
Figure 21: Change the order of the field

```python
def winreg_setvalue(packet):
    try:
        if packet['RAW.DCERPC']["opnum"] == 22:
            return packet
    except:
        return None
```

We are going to also add an execution that takes out the value of the `value` field from the RAW.WINREG layer on the screen, in this way we would have

Figure 22: Recalculate Value field

intercepted packets and stays with the `setValue` packets, for that we are going to use the `opnum` field of the RAW.DCERPC layer as filter. The precondition is shown below.
built a small sniffer of the WINREG protocol. The execution is shown below.

def print_value(packet):
    print(packet['RAW.WINREG']['value'])
    return packet

Finally, we use the command **intercept** to start intercepting packages and execute the conditional functions that we have added on them. Figure 25 shows the execution of all the previous commands and the result of sending a few setValue packets from one legitimate machine to the other.

As you can see, at this point we would be intercepting all setValue packets of the legitimate user and printing them on the screen. To end the case study, we will add the execution and postcondition necessary to modify the value introduced by the legitimate user.
Let’s start by seeing the format in bytes that the value field has in the packet. To do this, we return to the Polymorph interface, access the field and transform it into type bytes. In figure 26 you can see the process.

```
def insert_value(packet):
    f_len = packet['RAW.WINREG']['value_size']
    i_value = b'a\x00t\x00\x00a\x00c\x00k\x00e\x00r\x00'
    i_value += b'\x00' * (f_len - len(i_value))
    packet['RAW.WINREG']['value'] = i_value
    return packet
```

As can be seen, the execution will not be useful for all use cases. In order to
not make the case too long and complicated, values that are smaller than the
original will be inserted and then padding will be made with null bytes up to
the size of the original field. In this way we do not leave all the control fields
related to the size of the packet inconsistent.

To finish, we are going to reuse a postcondition that we used in the previous
section to recalculate the checksum control fields of the IP and TCP layers.

def recalculate_tcp_ip(packet):
    from scapy.all import IP
    pkt = IP(packet.raw[14:])
    if pkt.haslayer('IP') and pkt.haslayer('TCP'):
        del pkt['IP'].chksum
        del pkt['TCP'].chksum
        pkt.show2()
        packet.raw = bytes(pkt)
    return packet

Once these conditional functions have been inserted, the only thing left for us
is to put the framework to intercept packages. The result should be something
similar to Figure 27, and the value that must be set in the registry of the remote
machine must be the value entered by the attacker (Figure 28).
16 ANNEX 3: All commands and their function

This section describes all the commands of each of the interfaces of the framework, as well as all the options they have.
16.1 Main interface

- **capture**: Capture packets from a specific interface and transform them into a template list.
  
  Options:
  - `-h` prints the help.
  - `-f` allows packet filtering using the BPF notation.
  - `-c` number of packets to capture.
  - `-t` stop sniffing after a given time.
  - `-file` read a .pcap file from disk.
  - `-v` verbosity level medium.
  - `-vv` verbosity level high.

- **spoof**: Performs an ARP spoofing between machines in the network.
  
  Options:
  - `-h` prints the help.
  - `-t` targets to perform the ARP spoofing. Separated by ‘,’
  - `-g` gateway to perform the ARP spoofing
  - `-i` network interface.

- **clear**: Clears the screen.

16.2 Template List interface

- **show**: Prints information about the list of templates.
  
  Options:
  - `-h` prints the help.
  - `-t` show a particular template.

- **dissect**: Dissects the captured packets with the Tshark dissectors and generates a template from the packet.
  
  Options:
  - `-h` prints the help.
  - `-t` dissects until a particular template.

- **template**: Access the content of a particular template.
  
  Options:
  - `-h` prints the help.

- **wireshark**: Opens the captured file with Wireshark.
  
  Options:
  - `-h` prints the help.
  - `-p` indicate a new path to the wireshark binary.

- **back**: Returns to the previous interface.
16.3 Template interface

- **show**: Prints information about the template.
  Options:
  -h prints the help.
  -l shows a particular layer.

- **name**: Manages the name of the Template.
  Options:
  -h prints the help.
  -n set a new name to the template.

- **layer**: Access and manage the layers of the Template.
  Options:
  -h prints the help.
  -a adds a new layer to the Template.
  -d deletes a custom layer from the Template.

- **dump**: Dumps the packet bytes in different formats.
  Options:
  -h prints the help.
  -hex dump the packet bytes encoded in hexadecimal.
  -b dump the packet bytes without encoding.
  -hexstr dump the packet bytes as an hexadecimal stream.

- **layers**: Prints the layers of the Template.
  Options:
  -h prints the help.
  -c prints the custom layers.

- **preconditions**: Will run when a packet arrive. This functions receive a parameter that is the packet that will arrive when intercepting in real time. Furthermore, this functions must return the same packet if the user want to continue with the execution of the next conditions. There are three different types of conditions, Preconditions, which are associated with some input requirements that the packets must meet, Executions, which will do processing actions to the previously filtered packets and Postconditions, which have to be with actions that must be performed before forwarding the packet (ex. checksum reclaculations).
  Options:
  -h prints the help.
  -a adds a new condition to the set.
  -d deletes a condition from the set.
-e open a text editor that is in the path with the existing conditions, by default pico.
-s prints the conditions with the source code.
-i import a function from a file.
-sa prints all the conditions on disk.
-sas prints all the conditions source on disk.

• **postconditions**: Will run when a packet arrive. This functions receive a parameter that is the packet that will arrive when intercepting in real time. Furthermore, this functions must return the same packet if the user want to continue with the execution of the next conditions. There are three different types of conditions, Preconditions, which are associated with some input requirements that the packets must meet, Executions, which will do processing actions to the previously filtered packets and Postconditions, which have to be with actions that must be performed before forwarding the packet (ex. checksum reclaculations).

Options:
- h prints the help.
- a adds a new condition to the set.
- d deletes a condition from the set.
- e open a text editor that is in the path with the existing conditions, by default pico.
- s prints the conditions with the source code.
- i import a function from a file.
- sa prints all the conditions on disk.
- sas prints all the conditions source on disk.

• **executions**: Will run when a packet arrive. This functions receive a parameter that is the packet that will arrive when intercepting in real time. Furthermore, this functions must return the same packet if the user want to continue with the execution of the next conditions. There are three different types of conditions, Preconditions, which are associated with some input requirements that the packets must meet, Executions, which will do processing actions to the previously filtered packets and Postconditions, which have to be with actions that must be performed before forwarding the packet (ex. checksum reclaculations).

Options:
- h prints the help.
- a adds a new condition to the set.
- d deletes a condition from the set.
- e open a text editor that is in the path with the existing conditions, by default pico.
- s prints the conditions with the source code.
- i import a function from a file.
- sa prints all the conditions on disk.
prints all the conditions source on disk.

- **intercept**: Starts intercepting packets in real time.
  Options:
  -h prints the help.
  -ipt iptables rule for ipv4
  -ip6t iptables rule for ipv6

- **timestamp**: Shows the timestamp of the Template.

- **save**: Saves the Template to disk.
  Options:
  -h prints the help.
  -p path where the Template will be written.

- **version**: Manages the version of the Template.
  Options:
  -h prints the help.
  -n sets a new version.

- **description**: Manages the description of the Template.
  Options:
  -h prints the help.
  -n sets a new description.

- **spoof**: Performs an ARP spoofing between machines in the network.
  Options:
  -h prints the help.
  -t targets to perform the ARP spoofing. Separated by ‘,’
  -g gateway to perform the ARP spoofing
  -i network interface.

- **back**: Returns to the previous interface.

16.4 **Layer interface**

- **show**: Prints information about the layer.
  Options:
  -h prints the help.
  -f shows a particular field.

- **field**: Access and manage the fields of the layer.
Options:
−h prints the help.
−a adds a new field to the layer.
−d deletes a custom field from the layer.

- **fields**: Prints the fields of the layer.

Options:
−h prints the help.
−c prints the custom fields.

- **dump**: Dumps the layer bytes in different formats.

Options:
−h prints the help.
−hex dump the packet bytes encoded in hexadecimal.
−b dump the packet bytes without encoding.
−hexstr dump the packet bytes as an hexadecimal stream.

- **recalculate**: Creates a structure that relates a field to other fields of the layer, so that its value can be calculated dynamically at run time.

Options:
−h prints the help.
−f field to be recalculated
−sb start byte of the field that you want to recalculate.
−e expression that recalculates the field
−t tests a previously created structure
−s shows the struct for a particular field
−d deletes a struct from a field.

- **back**: Returns to the previous interface.

### 16.5 Field interface

- **show**: Shows the characteristics of the field.

- **value**: Manages the field value.

Options:
−h prints the help.
−a add a new value to the field
−t type of the value ("hex", "bytes", "str", "int").
  By default "str".
−hex prints the value encoded in hex.
−b prints the value encoded in bytes.

- **type**: Manages the field type.
Options:
- h prints the help.
- a add a new type to the field ('hex', 'str', 'bytes', 'int')
- o order for int type ('big', 'little')

- **name**: Manage the name of the field.
  Options:
  - h prints the help.
  - n set a new name to the field.

- **slice**: Prints the slice of the field.

- **custom**: Manage the custom property of the field.
  Options:
  - h prints the help.
  - set set the field as custom.
  - unset unset the field as custom

- **size**: Prints the size of the field.

- **dump**: Prints the hex dump of the field.

- **back**: Returns to the previous interface.