Netzob Documentation

Release 1.0~git

Frédéric Guihéry, Georges Bossert

May 08, 2017

Contents

Netzob is an open source tool for reverse engineering, traffic generation and fuzzing of communication protocols. It can be used to infer the message format and the state machine of a protocol through passive and active processes. The model can afterward be used to simulate realistic and controllable trafic.

The main *features* of Netzob are:

Protocol Vocabulary Modeling and Inference Netzob includes a complete model to represents the message format of a protocol (aka its vocabulary). Using specific algorithms, it allows to learn it from provided traces.

Protocol Grammar Modeling and Inference The state machine of a protocol (aka its grammar) defines the valid sequences of exchanged messages. Netzob allows to learn it semi-automaticaly using specific algorithms.

Protocol Simulation To support the inferring process, a dynamic analysis is perfomed based on simulated actors. These can initiate and take part in a complex communication following the infered protocol.

CHAPTER 1

Contact information

Website http://www.netzob.org

Email contact@netzob.org

Mailing list Users, developers and announces lists are available, use the SYMPA web interface to register.

IRC You can hang-out with us on Freenode's IRC channel #netzob @ freenode.org.

Wiki Discuss strategy on Netzob's wiki

Twitter Follow Netzob's official accounts (@Netzob)

CHAPTER 2

Netzob Overview

Overview of Netzob

Netzob has been initiated by security auditors of AMOSSYS and the CIDre research team of Supélec to address the reverse engineering of communication protocols.

Originaly, the development of Netzob has been initiated to support security auditors and evaluators in their activities of modeling and simulating undocumented protocols. The tool has then been extended to allow smart fuzzing of unknown protocol.

The following picture depicts the main modules of Netzob:

Fig. 2.1: Architecture of Netzob

- **Import module:** Data import is available in two ways: either by leveraging the channel-specific captors (currently network and IPC – Inter-Process Communication), or by using specific importers (such as PCAP files, structured files and OSpy files).
- **Protocol inference modules:** The vocabulary and grammar inference methods constitute the core of Netzob. It allows both passive and active reverse engineering of communication flows through automated and manuals mechanisms.
- **Simulation module:** Given vocabulary and grammar models previously inferred, Netzob can understand and generate communication traffic between multiple actors. It can act as either a client, a server or both.
- **Export module:** This module permits to export an inferred model of a protocol in formats that are understandable by third party software or by a human. Current work focuses on export format compatible with main traffic dissectors (Wireshark and Scapy) and fuzzers (Peach and Sulley).

And here is a screenshot of the main graphical interface:

The following sections will describe in more details the available mechanisms.

Import and capture data

The first step in the inferring process of a protocol in Netzob is to capture and to import messages as samples. There are different methods to retrieve messages depending of the communication channel used (files, network, IPC, USB, etc.) and the format (PCAP, hex, raw binary flows, etc.).

The figure below describes the multiple communication channels and therefore possible sniffing point's Netzob aims at addressing.

Fig. 2.2: Multiple communication flows arround an application

The current version (version 0.4) of Netzob deals with the following data sources :

- **Live network communications**
- **Captured network communications** (PCAPs)
- **Inter-Process Communications** (IPCs)
- **Text and binary files**
- **API flows** through oSpy file format support

Otherwise, if you plan to reverse a protocol implemented over an supported communication channel, Netzob's can manipulates any communications flow through an XML representation. Therefore, this situation only requires a specific development to capture the targeted flow and to save it using a compatible XML.

Fig. 2.3: Importing data from an unknown communication channel using the XML definition

Inferring message format and state machine with Netzob

The vocabulary of a communication protocol defines all the words which are integrated in it. For example, the vocabulary of a malware's communication protocol looks like a set of possible commands : {"attack www.google.fr", "dnspoison this.dns.server.com", "execute 'uname -a'", ...}. Another example of a vocabulary is the set of valids words in the HTTP protocol : { "GET /images/logo.png HTTP/1.1 ...", "HTTP/1.1 200 OK ...", ...}.

Netzob's vocabulary inferring process has been designed in order to retrieve the set of all possible words used in a targeted protocol and to identify their structures. Indeed words are made of different fields which are defined by their value and types. Hence a word can be described using the structure of its fields.

We describe the learning process implemented in Netzob to semi-automatically infer the vocabulary and the grammar of a protocol. This process, illustrated in the following picture, is performed in three main steps:

1. **Clustering messages and partitioning these messages in fields.**
2. **Characterizing message fields and abstracting similar messages in symbols.**
3. **Inferring the transition graph of the protocol.**

Fig. 2.4: The main functionalities

Step 1: clustering Messages and Partitioning in Fields

To discover the format of a symbol, Netzob supports different partitioning approaches. In this article we describe the most accurate one, that leverages sequence alignment processes. This technique permits to align invariants in a set of messages. The Needleman-Wunsh algorithm performs this task optimally. Needleman-Wunsh is particularly effective on protocols where dynamic fields have variable lengths (as shown on the following picture).

Fig. 2.5: Sequence alignment with Needleman-Wunsh algorithm

When partitioning and clustering processes are done, we obtain a relevant first approximation of the overall message formats. The next step consists in determining the characteristics of the fields.

If the size of those fields is fixed, as in TCP and IP headers, it is preferable to apply a basic partitioning, also provided by Netzob. Such partitioning works by aligning each message by the left, and then separating successive fixed columns from successive dynamic columns.

To regroup aligned messages by similarity, the Needleman-Wunsh algorithm is used in conjunction with a clustering algorithm. The applied algorithm is UPGMA.

Step 2 : characterization of Fields

The field type identification partially derives from the partitioning inference step. For fields containing only invariants, the type merely corresponds to the invariant value. For other fields, the type is automatically materialized, in first approximation, with a regular expression, as shown on next figure. This form allows to easily validate the data conformity with a specific type. Moreover, Netzob offers the possibility to visualize the definition domain of a field. This helps to manually refine the type associated with a field.

Fig. 2.6: Characterization of field type

Some intra-symbol dependencies are automatically identified. The size field, present in many protocol formats, is an example of intra-symbol dependency. A search algorithm has been designed to look for potential size fields and their associated payloads. By extension, this technique permits to discover encapsulated protocol payloads.

Environmental dependencies are also identified by looking for specific values retrieved during message capture. Such specific values consist of characteristics of the underlying hardware, operating system and network configuration. During the dependency analysis, these characteristics are searched in various encoding.

Step 3: inferring the Transition Graph of the Protocol

The third step of the learning process discovers and extracts the transition graph from a targeted protocol (also called the grammar). More formally, the grammar of a communication protocol defines the set of valid sentences which can be produced by a communication. A sentence is a sorted set of words which may be received or emmited by a protocol handler. An exemple of a simple sentence is :

```
["attack www.google.fr", "attack has failed", "attack www.kernel.org", "root access␣
↪granted."]
```

which can be described using the following simple automata with S0 the initial state :

Fig. 2.7: Schema of a simple grammar

The learning process step is achieved by a set of active experiments that stimulate a real client or server implementation using successive sequences of input symbols and analyze its responses.

In Netzob, the automata used to represent or model a communication protocol is an extended version of a Mealy automata which includes semi-stochastic transitions, contextualized and parametrized inputs and outputs. The first academic presention of this model is included in a dedicated scientific paper provided in the documentation section.

The model is inferred through a dedicated **active** process which consists in stimulating an implementation and to analyze its responses. In this process, we use the previously infered vocabulary to discover and to learn the grammar of the communication protocol. Each stimulation is computed following an extension of the **Angluin L** algorithm*.

Protocol simulation

One of our main goal is to generate realistic network traffic from undocummented protocols. Therefore, we have implemented a dedicated module that, given vocabulary and grammar models previously infered, can simulate a communication protocol between multiple bots and masters. Besides their use of the same model, each actors is independent from the others and is organized around three main stages.

The first stage is a dedicated library that reads and writes from the network channel. It also parses the flow in messages according to previous protocols layers. The second stage uses the vocabulary to abstract received messages into symbols and vice-versa to specialize emitted symbols into messages. A memory buffer is also available to manage dependency relations. The last stage implements the grammar model and computes which symbols must be emitted or received according to the current state and time.

Smart fuzzing with Netzob

A typical example of dynamic vulnerability analysis is the robustness tests. It can be used to reveal software programming errors which can leads to software security vulnerabilities. These tests provide an efficient and almost automated solution to easily identify and study exposed surfaces of systems. Nevertheless, to be fully efficient, the fuzzing approaches must cover the complete definition domain and combination of all the variables which exist in a protocol (IP adresses, serial numbers, size fields, payloads, message identifer, etc.). But fuzzing typical communication interface requires too many test cases due to the complex variation domains introduced by the semantic layer of a protocol. In addition to this, an efficient fuzzing should also cover the state machine of a protocol which also brings another huge set of variations. The necessary time is nearly always too high and therefore limits the efficiency of this approach.

With all these contraints, achieving robustness tests on a target is feasible only if the expert has access to a specially designed tool for the targeted protocol. Hence the emergence of a large number of tools to verify the behavior of an application on one or more communication protocols. However in the context of proprietary communications protocols for which no specifications are published, fuzzers do not provide optimal results.

Netzob helps the security evaluator by simplifying the creation of a dedicated fuzzer for a proprietary or undocumented protocol. It allows the expert to execute a semi-automated inferring process to create a model of the targeted protocol. This model can afterward be refined by the evaluator. Finally, the created model is included in the fuzzing module of Netzob which considers the vocabulary and the grammar of the protocol to generate optimized and specific test cases. Both mutation and generation are available for fuzzing.

Export protocol model

The following export formats are currently provided by Netzob:

- XML format
- human readable (Wireshark like)
- Peach fuzzer export: this allows to combine efficiency of Peach Fuzzer on previously undocummented protocols.

Besides, you can write your own exporter to manipulate the inferred protocol model in your favorite tool.

Netzob has been initiated by security auditors of AMOSSYS and the CIDre research team of Supélec to address the reverse engineering of communication protocols. A detailed overview of the project is *available here*.

CHAPTER 3

Tutorials

Discover features of Netzob The goal of this tutorial is to present the usage of each main component of Netzob (inference of message format, construction of the state machine, generation of traffic and fuzzing) through an undocumented protocol.

Modeling your Protocol with Netzob This tutorial details the main features of Netzob's protocol modeling aspects. It shows how your protocol fields can be described with Netzob's language.

CHAPTER 4

API Documentation

netzob package

Subpackages

netzob.Common package

Subpackages

netzob.Common.C_Extensions package

Submodules

netzob.Common.C_Extensions.WrapperArgsFactory module

netzob.Common.C_Extensions.WrapperMessage module

class `WrapperMessage` (*message*, *symbolID*)

Bases: `object`

Definition of a wrapped message ready to be sent to any C extension

Module contents

netzob.Common.Utils package

Subpackages

netzob.Common.Utils.DataAlignment package

Submodules

netzob.Common.Utils.DataAlignment.DataAlignment module

netzob.Common.Utils.DataAlignment.ParallelDataAlignment module

Module contents

netzob.Common.Utils.Serialization package

Submodules

netzob.Common.Utils.Serialization.JSONSerializator module

class **JSONSerializator**
Bases: object

static **serialize**(*obj*)
Serialize the specified object under a specific JSON format. It inspects the specified object to search for attributes to serialize.

```
>>> from netzob.all import *
>>> msg = RawMessage("hello")
>>> print(JSONSerializator.serialize(msg))
```

It's not possible to serialize a None object

```
>>> JSONSerializator.serialize(None)
Traceback (most recent call last):
...
TypeError: Cannot serialize a None object
```

Parameters **obj** (object) – the object to serialize

Returns the object serialized in JSON

Return type str

Module contents

netzob.Common.Utils.UndoRedo package

Submodules

netzob.Common.Utils.UndoRedo.AbstractMemento module

class **AbstractMemento**(*originator*)
Bases: object

This class represents a Memento meaning the serialization of an object state.

originator
The instance from which the memento has been computed

netzob.Common.Utils.UndoRedo.AbstractMementoCreator module

class AbstractMementoCreator
Bases: object

Parent class of objects to save for Undo/Redo.

This abstract class must be inherited by all the objects which need to be saved for Undo/Redo processes. These objects have to provide two methods, storeInMemento and restoreFromMemento both used to save and restore current state of the object.

restoreFromMemento(*memento*)
This method restores current object internals with provided memento.

The provided memento should be created by the storeInMemento method and represents the current object. It returns the current state of the object before the restore operation

Parameters **memento** (netzob.Common.Utils.UndoRedo.AbstractMemento.AbstractMemento) – memento containing internals to set in current object to restore it.

Returns the memento of current object before executing the restore process

Return type *netzob.Common.Utils.UndoRedo.AbstractMemento.AbstractMemento*

storeInMemento()
This method creates a memento to represent the current state of object.

This memento should be stored in the UndoRedo action stack and might be used as a parameter of the restoreFromMemento method.

Returns the created memento representing current object

Return type *netzob.Common.Utils.UndoRedo.AbstractMemento.AbstractMemento*

Module contents

Submodules

netzob.Common.Utils.Decorators module

NetzobLogger(*klass*)
This class decorator adds (if necessary) an instance of the logger (self.__logger) to the attached class and removes from the getState the logger.

typeCheck(**types*)
Decorator which reduces the amount of code to type-check attributes.

Its allows to replace the following code:

```
@id.setter
def id(self, id):
    if not isinstance(id, uuid.UUID):
       raise TypeError("Invalid types for argument id, must be an UUID")
    self.__id = id
```

with:

```
@id.setter
@typeCheck(uuid.UUID)
def id(self, id):
    self.__id = id
```

Note: set type = "SELF" to check the type of the self parameter

Warning: if argument is None, the type checking is not executed on it.

netzob.Common.Utils.MatrixList module

class MatrixList
Bases: list

This type of list has been created to represent it as matrix which means its a list of list.

The __str__ method has been redefined to propose a nice representation of its content.

headers
A list of sorted strings. Each string will be displayed as a column header

netzob.Common.Utils.SortableObject module

class SortableObject
Bases: object

priority()

netzob.Common.Utils.SortedTypedList module

netzob.Common.Utils.TypedList module

class TypedList(*membersTypes*, **args*)
Bases: collections.abc.MutableSequence

A strong typed list based on collections.MutableSequence.

The idea is to verify members type when editing the list. By using this class instead of the typical list, we enforce members type.

```
>>> typedList = TypedList(str)
>>> typedList.append("toto")
>>> typedList.extend(["titi", "tata"])
>>> len(typedList)
3
>>> typedList[1]
'titi'
>>> typedList.append(3)
Traceback (most recent call last):
```

```
TypeError: Invalid type for argument, expecting: <type 'str'>
>>> typedList.extend(["tutu", 5])
Traceback (most recent call last):
TypeError: Invalid type for argument, expecting: <type 'str'>
```

check(*v*)

insert(*i*, *v*)

netzob.Common.Utils.all module

Module contents

Submodules

netzob.Common.CommandLine module

class CommandLine

Bases: object

Reads, validates and parses the command line arguments provided by users

configure()

Configure the parser based on Netzob's usage and the definition of its options and arguments

getConfiguredParser()

Return (if available) the parser configured to manage provided arguments and options by user. @return: the parser

getOptions()

isInteractiveConsoleRequested()

Compute and returns if the user has requested the initiation of an interactive session

parse()

Read and parse the provided arguments and options

netzob.Common.DepCheck module

class DepCheck

Bases: object

Dependency checker. Provides multiple static method to check is required and optionnal dependency are available.

static checkCExtensions()

static checkRequiredDependency()

netzob.Common.LoggingConfiguration module

LoggingConfiguration(**args*, ***kwargs*)

singleton(*cls*, **args*, ***kwargs*)
This decorator allows to implement some kind of Singleton design pattern. In our case, we only allow one instanciation.

netzob.Common.NetzobException module

exception NetzobException(*value*)
Bases: Exception

Class of handling Netzob specific exceptions

exception NetzobImportException(*source*, *message*, *statusCode=None*, *subCode=None*)
Bases: *netzob.Common.NetzobException.NetzobException*

Raised if an error was encountered while importing data

netzob.Common.all module

Module contents

netzob.Import package

Subpackages

netzob.Import.FileImporter package

Submodules

netzob.Import.FileImporter.FileImporter module

netzob.Import.FileImporter.all module

Module contents

netzob.Import.PCAPImporter package

Submodules

netzob.Import.PCAPImporter.ImpactDecoder module

class BaseDecoder
Bases: *netzob.Import.PCAPImporter.ImpactDecoder.Decoder*

decode(*buff*)

class DataDecoder
Bases: *netzob.Import.PCAPImporter.ImpactDecoder.Decoder*

decode(*aBuffer*)

class Decoder
Bases: object

decode(*aBuffer*)

get_protocol(*aprotocol*)

set_decoded_protocol(*protocol*)

class EthDecoder
Bases: *netzob.Import.PCAPImporter.ImpactDecoder.Decoder*

decode(*aBuffer*)

class ICMPDecoder
Bases: *netzob.Import.PCAPImporter.ImpactDecoder.Decoder*

decode(*aBuffer*)

class IPDecoder
Bases: *netzob.Import.PCAPImporter.ImpactDecoder.Decoder*

decode(*aBuffer*)

class IPDecoderForICMP
Bases: *netzob.Import.PCAPImporter.ImpactDecoder.Decoder*

This class was added to parse the IP header of ICMP unreachables packets If you use the "standard" IPDecoder, it might crash (see bug #4870) ImpactPacket.py because the TCP header inside the IP header is incomplete

decode(*aBuffer*)

class LinuxSLLDecoder
Bases: *netzob.Import.PCAPImporter.ImpactDecoder.Decoder*

decode(*aBuffer*)

class TCPDecoder
Bases: *netzob.Import.PCAPImporter.ImpactDecoder.Decoder*

decode(*aBuffer*)

class UDPDecoder
Bases: *netzob.Import.PCAPImporter.ImpactDecoder.Decoder*

decode(*aBuffer*)

netzob.Import.PCAPImporter.ImpactPacket module

class Data(*aBuffer=None*)
Bases: *netzob.Import.PCAPImporter.ImpactPacket.Header*

This packet type can hold raw data. It's normally employed to hold a packet's innermost layer's contents in those cases for which the protocol details are unknown, and there's a copy of a valid packet available.

For instance, if all that's known about a certain protocol is that a UDP packet with its contents set to "HELLO" initiate a new session, creating such packet is as simple as in the following code fragment: packet = UDP() packet.contains('HELLO')

get_size()

set_data(*data*)

class Ethernet(*aBuffer=None*)
Bases: *netzob.Import.PCAPImporter.ImpactPacket.Header*

static as_eth_addr(*anArray*)

get_ether_dhost()
Return 48 bit destination ethernet address as a 6 byte array

get_ether_shost()
Return 48 bit source ethernet address as a 6 byte array

get_ether_type()
Return ethernet data type field

get_header_size()
Return size of Ethernet header

get_packet()

get_tag(*index*)
Returns an EthernetTag initialized from index-th VLAN tag. The tags are numbered from 0 to self.tag_cnt-1 as they appear in the frame. It is possible to use negative indexes as well.

load_header(*aBuffer*)

pop_tag(*index=0*)
Removes the index-th VLAN tag and returns it as an EthernetTag object. Index defaults to 0 (the top of the stack).

push_tag(*tag*, *index=0*)
Inserts contents of an EthernetTag object before the index-th VLAN tag. Index defaults to 0 (the top of the stack).

set_ether_dhost(*aValue*)
Set destination ethernet address from 6 byte array 'aValue'

set_ether_shost(*aValue*)
Set source ethernet address from 6 byte array 'aValue'

set_ether_type(*aValue*)
Set ethernet data type field to 'aValue'

set_tag(*index*, *tag*)
Sets the index-th VLAN tag to contents of an EthernetTag object. The tags are numbered from 0 to self.tag_cnt-1 as they appear in the frame. It is possible to use negative indexes as well.

class **EthernetTag**(*value=2164260864*)
Bases: *netzob.Import.PCAPImporter.ImpactPacket.PacketBuffer*

Represents a VLAN header specified in IEEE 802.1Q and 802.1ad. Provides methods for convenient manipulation with header fields.

get_dei()
Returns Drop Eligible Indicator

get_pcp()
Returns Priority Code Point

get_tpid()
Returns Tag Protocol Identifier

get_vid()
Returns VLAN Identifier

set_dei(*value*)
Sets Drop Eligible Indicator

set_pcp(*value*)
Sets Priority Code Point

set_tpid(*value*)
Sets Tag Protocol Identifier

set_vid(*value*)
Sets VLAN Identifier

class **Header**(*length=None*)
Bases: *netzob.Import.PCAPImporter.ImpactPacket.PacketBuffer*, *netzob.Import.PCAPImporter.ImpactPacket.ProtocolLayer*

This is the base class from which all protocol definitions extend.

calculate_checksum()
Calculate and set the checksum for this header

ethertype = None

get_data_as_string()
Returns all data from children of this header as string

get_header_size()
Return the size of this header, that is, not counting neither the size of the children nor of the parents.

get_packet()
Returns the raw representation of this packet and its children as a string. The output from this method is a packet ready to be transmitted over the wire.

get_pseudo_header()
Pseudo headers can be used to limit over what content will the checksums be calculated.

get_size()
Return the size of this header and all of it's children

list_as_hex(*aList*)

load_header(*aBuffer*)
Properly set the state of this instance to reflect that of the raw packet passed as argument.

packet_printable = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'(

protocol = None

class **ICMP**(*aBuffer=None*)
Bases: *netzob.Import.PCAPImporter.ImpactPacket.Header*

ICMP_UNREACH = 3

get_checksum()

get_code()

get_header_size()

get_icmp_type()

get_identifier()

get_packet()

get_sequence_number()

protocol = 1

set_checksum(*value*)

set_code(*value*)

set_icmp_type(*value*)

set_identifier(*value*)

set_sequence_number(*value*)

class **IP**(*aBuffer=None*)

Bases: *netzob.Import.PCAPImporter.ImpactPacket.Header*

add_option(*option*)

ethertype = 2048

fragment_by_list(*aList*)

fragment_by_size(*aSize*)

get_header_size()

get_ip_df()

get_ip_dst()

get_ip_hl()

get_ip_id()

get_ip_len()

get_ip_mf()

get_ip_off()

get_ip_offmask()

get_ip_p()

get_ip_rf()

get_ip_src()

get_ip_sum()

get_ip_tos()

get_ip_ttl()

get_ip_v()

get_packet()

get_pseudo_header()

load_header(*aBuffer*)

reset_ip_sum()

set_ip_df(*aValue*)

set_ip_dst(*value*)

set_ip_hl(*value*)

set_ip_id(*value*)

set_ip_len(*value*)

set_ip_mf(*aValue*)

set_ip_off(*aValue*)

set_ip_offmask(*aValue*)

set_ip_p(*value*)

set_ip_rf(*aValue*)

set_ip_src(*value*)

set_ip_sum(*value*)

set_ip_tos(*value*)

set_ip_ttl(*value*)

set_ip_v(*value*)

class IPOption(*opcode=0*, *size=None*)
Bases: *netzob.Import.PCAPImporter.ImpactPacket.PacketBuffer*

IPOPT_EOL = 0

IPOPT_LSRR = 131

IPOPT_NOP = 1

IPOPT_RR = 7

IPOPT_SSRR = 137

IPOPT_TS = 68

append_ip(*ip*)

get_code()

get_flags(*flags*)

get_len()

get_ptr()

print_addresses()

set_code(*value*)

set_flags(*flags*)

set_len(*len*)

set_ptr(*ptr*)

exception ImpactPacketException(*value*)
Bases: Exception

class LinuxSLL(*aBuffer=None*)
Bases: *netzob.Import.PCAPImporter.ImpactPacket.Header*

get_addr()
Returns the sender's address field

get_addr_len()
Returns the length of the sender's address field

get_arphdr()
Returns the ARPHDR value for the link layer device type

get_ether_type()
Return ethernet data type field

get_header_size()
Return size of packet header

get_packet()

get_type()
Returns the packet type field

get_type_desc()

set_addr(*addr*)
Sets the sender's address field to addr. Addr must be at most 8-byte long.

set_addr_len(*len*)
Sets the length of the sender's address field to len

set_arphdr(*value*)
Sets the ARPHDR value for the link layer device type

set_ether_type(*aValue*)
Set ethernet data type field to 'aValue'

set_type(*type*)
Sets the packet type field to type

type_descriptions = ['sent to us by somebody else', 'broadcast by somebody else', 'multicast by somebody else', 'sen

class PacketBuffer(*length=None*)
Bases: object

Implement the basic operations utilized to operate on a packet's raw buffer. All the packet classes derive from this one.

The byte, word, long and ip_address getters and setters accept negative indexes, having these the a similar effect as in a regular Python sequence slice.

compute_checksum(*anArray*)
Return the one's complement of the one's complement sum of all the 16-bit words in 'anArray'

get_buffer_as_string()
Returns the packet buffer as a string object

get_byte(*index*)
Return byte at 'index'

get_bytes()
Returns the packet buffer as an array

get_ip_address(*index*)
Return 4-byte value at 'index' as an IP string

get_long(*index*, *order='!'*)
Return 4-byte value at 'index'. See struct module's documentation to understand the meaning of 'order'.

get_long_long(*index*, *order='!'*)
Return 8-byte value at 'index'. See struct module's documentation to understand the meaning of 'order'.

get_word(*index*, *order='!'*)
Return 2-byte word at 'index'. See struct module's documentation to understand the meaning of 'order'.

normalize_checksum(*aValue*)

set_byte(*index*, *value*)
Set byte at 'index' to 'value'

set_bytes(*bytes*)
Set the packet buffer from an array

set_bytes_from_string(*data*)
Sets the value of the packet buffer from the string 'data'

set_checksum_from_data(*index*, *data*)
Set 16-bit checksum at 'index' by calculating checksum of 'data'

set_ip_address(*index*, *ip_string*)
Set 4-byte value at 'index' from 'ip_string'

set_long(*index*, *value*, *order='!'*)
Set 4-byte 'value' at 'index'. See struct module's documentation to understand the meaning of 'order'.

set_long_long(*index*, *value*, *order='!'*)
Set 8-byte 'value' at 'index'. See struct module's documentation to understand the meaning of 'order'.

set_word(*index*, *value*, *order='!'*)
Set 2-byte word at 'index' to 'value'. See struct module's documentation to understand the meaning of 'order'.

class ProtocolLayer
Bases: `object`

Protocol Layer Manager for insertion and removal of protocol layers.

child()
Return the child of this protocol layer

contains(*aHeader*)
Set 'aHeader' as the child of this protocol layer

parent()
Return the parent of this protocol layer

set_parent(*my_parent*)
Set the header 'my_parent' as the parent of this protocol layer

unlink_child()
Break the hierarchy parent/child child/parent

class ProtocolPacket(*header_size*, *tail_size*)
Bases: *netzob.Import.PCAPImporter.ImpactPacket.ProtocolLayer*

body

body_string

get_body_as_string()

get_body_size()
Return frame body size

get_header_as_string()

get_header_size()
Return frame header size

get_packet()

get_size()
Return frame total size

get_tail_as_string()

get_tail_size()
Return frame tail size

header

load_body(*aBuffer*)
Load the packet body from string. WARNING: Using this function will break the hierarchy of preceding protocol layer

load_header(*aBuffer*)

load_packet(*aBuffer*)
Load the whole packet from a stringWARNING: Using this function will break the hierarchy of preceding protocol layer

load_tail(*aBuffer*)

tail

tail_string

class TCP(*aBuffer=None*)
Bases: *netzob.Import.PCAPImporter.ImpactPacket.Header*

TCP_FLAGS_MASK = 255

add_option(*option*)

calculate_checksum()

get_ACK()

get_CWR()

get_ECE()

get_FIN()

get_PSH()

get_RST()

get_SYN()

get_URG()

get_flag(*bit*)

get_header_size()

get_options()

get_packet()
Returns entire packet including child data as a string. This is the function used to extract the final packet

get_padded_options()
Return an array containing all options padded to a 4 byte boundry

get_th_ack()

get_th_dport()

get_th_flags()

get_th_off()

get_th_reserved()

get_th_seq()

get_th_sport()

get_th_sum()

get_th_urp()

get_th_win()

load_header(*aBuffer*)

protocol = 6

reset_ACK()

reset_CWR()

reset_ECE()

reset_FIN()

reset_PSH()

reset_RST()

reset_SYN()

reset_URG()

reset_flags(*aValue*)

set_ACK()

set_CWR()

set_ECE()

set_FIN()

set_PSH()

set_RST()

set_SYN()

set_URG()

set_flags(*aValue*)

set_th_ack(*aValue*)

set_th_dport(*aValue*)

set_th_flags(*aValue*)

set_th_off(*aValue*)

set_th_seq(*aValue*)

set_th_sport(*aValue*)

set_th_sum(*aValue*)

set_th_urp(*aValue*)

set_th_win(*aValue*)

swapSourceAndDestination()

class **TCPOption**(*kind*, *data=None*)

Bases: *netzob.Import.PCAPImporter.ImpactPacket.PacketBuffer*

TCPOPT_EOL = 0

TCPOPT_MAXSEG = 2

TCPOPT_NOP = 1

TCPOPT_SACK = 5

TCPOPT_SACK_PERMITTED = 4

TCPOPT_SIGNATURE = 19

TCPOPT_TIMESTAMP = 8

TCPOPT_WINDOW = 3

get_kind()

get_len()

get_mss()

get_shift_cnt()

get_size()

get_ts()

get_ts_echo()

set_kind(*kind*)

set_left_edge(*aValue*)

set_len(*len*)

set_mss(*len*)

set_right_edge(*aValue*)

set_shift_cnt(*cnt*)

set_ts(*ts*)

set_ts_echo(*ts*)

class UDP(*aBuffer=None*)

Bases: *netzob.Import.PCAPImporter.ImpactPacket.Header*

calculate_checksum()

get_header_size()

get_packet()

get_uh_dport()

get_uh_sport()

get_uh_sum()

get_uh_ulen()

protocol = 17

set_uh_dport(*value*)

set_uh_sport(*value*)

set_uh_sum(*value*)

set_uh_ulen(*value*)

netzob.Import.PCAPImporter.PCAPImporter module

netzob.Import.PCAPImporter.all module

Module contents

Submodules

netzob.Import.all module

Module contents

netzob.Inference package

Subpackages

netzob.Inference.Grammar package

Subpackages

netzob.Inference.Grammar.AutomataFactories package

Submodules

netzob.Inference.Grammar.AutomataFactories.ChainedStatesAutomataFactory module

netzob.Inference.Grammar.AutomataFactories.OneStateAutomataFactory module

netzob.Inference.Grammar.AutomataFactories.PTAAutomataFactory module

netzob.Inference.Grammar.AutomataFactories.all module

Module contents

netzob.Inference.Grammar.EquivalenceOracles package

Submodules

netzob.Inference.Grammar.EquivalenceOracles.AbstractEquivalenceOracle module

class **AbstractEquivalenceOracle**(*type*)

Bases: `object`

findCounterExample(*mmstd*)

netzob.Inference.Grammar.EquivalenceOracles.WMethodNetworkEquivalenceOracle module

Module contents

netzob.Inference.Grammar.Oracles package

Submodules

netzob.Inference.Grammar.Oracles.AbstractOracle module

class **AbstractOracle**(*type*)

Bases: `object`

start(*mmstd*)

stop()

netzob.Inference.Grammar.Oracles.NetworkOracle module

Module contents

netzob.Inference.Grammar.Queries package

Submodules

netzob.Inference.Grammar.Queries.MembershipQuery module

class **MembershipQuery**(*symbols*)

Bases: `object`

Represents a membership queryset of query which will be submited to an oracle

addSymbol(*symbol*)

getMQSuffixedWithMQ(*mq*)

getNotEmptyPrefixes()

getSymbols()

getSymbolsWhichAreNotEmpty()

isStrictlyEqual(*other*)

multiply(*mqs*)

toMMSTD(*dictionary*, *isMaster*)

Module contents

netzob.Inference.Grammar.lstar package

Submodules

netzob.Inference.Grammar.Istar.ObservationTable module

class **ObservationTable**(*alphabet*)

Bases: object

Implementation of an Observation Table (OT) as described by Angluin in "Learning Regular Sets from Queries and Counterexamples

alphabet

initialize(*initialSuffixes*, *mqOracle*)

Module contents

Submodules

netzob.Inference.Grammar.Angluin module

class **MealyLSTAR**(*inputVocabulary*, *membershipOracle*)

Bases: object

This class is an implementation of the Angluin L* Algorithm as detailled in "Learning regular sets from queries and counterexamples" [Ang87].

This active grammatical inference algorithm infers state machine. It communicates with a target by sending membership queries which requires to have access to an implementation of the protocol.

To illustrate its usage, we will infer the grammar of a fake simple protocol.

```
>>> from netzob.all import *
>>> import time
```

We first create a fake server which requires a vocabulary of input (I) and output (O) symbols:

```
>>> i0 = Symbol(name="a", fields=[Field("a\n")])
>>> i1 = Symbol(name="b", fields=[Field("b\n")])
>>> i2 = Symbol(name="c", fields=[Field("c\n")])
>>> i3 = Symbol(name="d", fields=[Field("d\n")])
>>> # List of Client > Server messages
>>> I = [i0, i1, i2, i3]
```

```
>>> o0 = Symbol(name="0", fields=[Field("0")])
>>> o1 = Symbol(name="1", fields=[Field("1")])
>>> o2 = Symbol(name="2", fields=[Field("2")])
>>> o3 = Symbol(name="3", fields=[Field("3")])
>>> # List of Server > Client messages
>>> O = [o0, o1, o2, o3]
```

```
>>> symbolList = I + O
```

Now we can create the grammar which includes 5 states

```
>>> s0 = State(name="S0")
>>> s1 = State(name="S1")
>>> s2 = State(name="S2")
>>> s3 = State(name="S3")
>>> s4 = State(name="S4")
```

and their transitions

```
>>> t0 = Transition(s0, s1, i0, [o0])
>>> t1 = Transition(s1, s1, i1, [o1])
>>> t2 = Transition(s1, s2, i2, [o2])
>>> t3 = Transition(s2, s1, i1, [o1])
>>> t4 = Transition(s2, s3, i0, [o0])
>>> t5 = Transition(s3, s1, i1, [o1])
>>> t6 = Transition(s3, s4, i2, [o2])
>>> t7 = Transition(s1, s4, i0, [o1])
```

we add an initial state and an ending state with open and close channel transitions

```
>>> initialState = State(name="Initial")
>>> endingState = State(name="End")
>>> openTransition = OpenChannelTransition(initialState, s0)
>>> closeTransition = CloseChannelTransition(s4, endingState)
>>> automata = Automata(initialState, symbolList)
```

```
>>> # Create an actor: Alice (a server)
>>> channel = UDPServer(localIP="127.0.0.1", localPort=8887)
>>> abstractionLayer = AbstractionLayer(channel, symbolList)
>>> alice = Actor(automata = automata, initiator = False,␣
↪abstractionLayer=abstractionLayer)
>>> alice.start()
```

We finaly create an angluin-based grammar learner

```
>>> # Creates an inference channel
>>> angluinChannel = UDPClient(remoteIP="127.0.0.1", remotePort=8887)
# >>> angluin = MealyLSTAR(inputSymbols = I, outputSymbols = O,␣
↪channel=angluinChannel)
# >>> angluin.start()
```

We wait for the results

```
>>> time.sleep(10)
```

>>> while (angluin.alive): time.sleep(5) >>> print("Inference finish") Inference finish

```
>>> alice.stop()
```

```
>>> print(angluin.initialStateOfInferedGrammar)
State
```

[Ang87] @article{Ang87, author = {Angluin, Dana}, title = {Learning regular sets from queries and counterexamples}, journal = {Inf. Comput.}, year = {1987}, volume = {75}, pages = {87–106}, month = {November} }

hypothesisModel

inputVocabulary

membershipOracle

refineHypothesis(*counterExample*)

startLearning()

netzob.Inference.Grammar.GenericMAT module

netzob.Inference.Grammar.GrammarInferer module

class **GrammarInferer**(*vocabulary*, *inputDictionary*, *oracle*, *equivalenceOracle*, *resetScript*, *cb_submitedQuery*, *cb_hypotheticalAutomaton*)

Bases: `threading.Thread`

applyMessagesOnAutomata(*automaton*, *messages*)

getHypotheticalAutomaton()

getInferedAutomaton()

getSubmitedQueries()

hasFinish()

infer()

run()

stop()

netzob.Inference.Grammar.LearningAlgorithm module

netzob.Inference.Grammar.MQCache module

class **MQCache**

Bases: `object`

cacheResult(*mq*, *result*)

dumpCache()

getCachedResult(*mq*)

preloadCache(*datas*, *vocabulary*)

preloadCacheEntry(*data*, *vocabulary*)

netzob.Inference.Grammar.all module

Module contents

netzob.Inference.Vocabulary package

Subpackages

netzob.Inference.Vocabulary.FormatOperations package

Subpackages

netzob.Inference.Vocabulary.FormatOperations.FieldSplitAligned package

Submodules

netzob.Inference.Vocabulary.FormatOperations.FieldSplitAligned.FieldSplitAligned module

Module contents

netzob.Inference.Vocabulary.FormatOperations.FieldSplitStatic package

Submodules

netzob.Inference.Vocabulary.FormatOperations.FieldSplitStatic.FieldSplitStatic module

netzob.Inference.Vocabulary.FormatOperations.FieldSplitStatic.ParallelFieldSplitStatic module

Module contents

Submodules

netzob.Inference.Vocabulary.FormatOperations.ClusterByAlignment module

netzob.Inference.Vocabulary.FormatOperations.ClusterByApplicativeData module

netzob.Inference.Vocabulary.FormatOperations.ClusterByKeyField module

netzob.Inference.Vocabulary.FormatOperations.ClusterBySize module

netzob.Inference.Vocabulary.FormatOperations.FieldOperations module

netzob.Inference.Vocabulary.FormatOperations.FieldReseter module

netzob.Inference.Vocabulary.FormatOperations.FieldSplitDelimiter module

netzob.Inference.Vocabulary.FormatOperations.FindKeyFields module

Module contents

netzob.Inference.Vocabulary.Search package

Submodules

netzob.Inference.Vocabulary.Search.SearchEngine module

netzob.Inference.Vocabulary.Search.SearchResult module

netzob.Inference.Vocabulary.Search.SearchTask module

netzob.Inference.Vocabulary.Search.all module

Module contents

Submodules

netzob.Inference.Vocabulary.CorrelationFinder module

netzob.Inference.Vocabulary.EntropyMeasurement module

class `EntropyMeasurement`

Bases: `object`

This utility class exposes various methods related to Entropy. This measure can be usefull to identify encrypted and compressed chunk of data accross various messages. By entropy we refer to the Shanon's one.

```
>>> import binascii
>>> from netzob.all import *
>>> fake_random_values = [b"00000906", b"00110906", b"00560902", b"00ff0901"]
>>> messages = [RawMessage(binascii.unhexlify(val)) for val in fake_random_values]
>>> [byte_entropy for byte_entropy in EntropyMeasurement.measure_
↪entropy(messages)]
[0.0, 2.0, 0.0, 1.5]
```

In the following example, 1000 messages are generated under a simple specification. In the specification, 5 bytes are randomly generated. This specificity can easily be spoited by the entropy measurement as illustred below.

```
>>> f1 = Field(b"hello ")
>>> f2 = Field(Raw(nbBytes=5))
>>> f3 = Field(b", welcome !")
>>> s = Symbol(fields=[f1, f2, f3])
>>> messages = [RawMessage(s.specialize()) for x in range(1000)]
>>> bytes_entropy = [byte_entropy for byte_entropy in EntropyMeasurement.measure_
↪entropy(messages)]
>>> min(bytes_entropy[6:11]) > 7
True
```

You can also measure the entropy of the data that are accepeted by a specific field.

```
>>> f1 = Field(Raw(nbBytes=2))
>>> f2 = Field(Raw(nbBytes=(10, 20)))
>>> f3 = Field(Raw(nbBytes=2))
>>> s = Symbol(fields=[f1, f2, f3])
>>> s.messages = [RawMessage(s.specialize()) for x in range(1000)]
>>> bytes_entropy = [byte_entropy for byte_entropy in EntropyMeasurement.measure_
↪values_entropy(f2.getValues())]
>>> print(min(bytes_entropy[:10]) > 7)
True
```

static `measure_entropy`(*messages*)

This method returns the entropy of bytes found at each position of the messages.

```
>>> [x for x in EntropyMeasurement.measure_entropy(messages=None)]
Traceback (most recent call last):
```

```
...
Exception: Messages cannot be None
```

```
>>> from netzob.all import *
>>> [x for x in EntropyMeasurement.measure_entropy(messages=[RawMessage()])]
Traceback (most recent call last):
...
Exception: At least two messages must be provided
```

static measure_values_entropy(*values*)

This method returns the entropy of bytes found at each position of the specified values.

```
>>> [x for x in EntropyMeasurement.measure_values_entropy(values=None)]
Traceback (most recent call last):
...
Exception: values cannot be None
```

```
>>> from netzob.all import *
>>> [x for x in EntropyMeasurement.measure_values_entropy(values=[])]
Traceback (most recent call last):
...
Exception: At least one value must be provided
```

netzob.Inference.Vocabulary.Format module

netzob.Inference.Vocabulary.RelationFinder module

netzob.Inference.Vocabulary.all module

Module contents

Submodules

netzob.Inference.all module

Module contents

netzob.Model package

Subpackages

netzob.Model.Grammar package

Subpackages

netzob.Model.Grammar.States package

Submodules

netzob.Model.Vocabulary.Domain.Variables.Leafs.AbstractVariableLeaf module

netzob.Model.Vocabulary.Domain.Variables.Leafs.Data module

netzob.Model.Vocabulary.Domain.Variables.Leafs.InternetChecksum module

netzob.Model.Vocabulary.Domain.Variables.Leafs.Size module

netzob.Model.Vocabulary.Domain.Variables.Leafs.Value module

netzob.Model.Vocabulary.Domain.Variables.Leafs.all module

Module contents

netzob.Model.Vocabulary.Domain.Variables.Nodes package

Submodules

netzob.Model.Vocabulary.Domain.Variables.Nodes.AbstractVariableNode module

netzob.Model.Vocabulary.Domain.Variables.Nodes.Agg module

netzob.Model.Vocabulary.Domain.Variables.Nodes.Alt module

netzob.Model.Vocabulary.Domain.Variables.Nodes.Repeat module

netzob.Model.Vocabulary.Domain.Variables.Nodes.all module

Module contents

Submodules

netzob.Model.Vocabulary.Domain.Variables.AbstractVariable module

netzob.Model.Vocabulary.Domain.Variables.Memory module

netzob.Model.Vocabulary.Domain.Variables.SVAS module

class SVAS

Bases: `object`

Represents a State Variable Assignment Strategy

The SVAS of a variable defines how its value is used while abstracting and specializing. The SVAS impacts the memorization strategy.

CONSTANT = 'Constant SVAS'

EPHEMERAL = 'Ephemeral SVAS'

PERSISTENT = 'Persistent SVAS'

VOLATILE = 'Volatile SVAS'

netzob.Model.Vocabulary.Domain.Variables.all module

Module contents

Submodules

netzob.Model.Vocabulary.Domain.DomainFactory module

netzob.Model.Vocabulary.Domain.GenericPath module

netzob.Model.Vocabulary.Domain.all module

Module contents

netzob.Model.Vocabulary.Functions package

Subpackages

netzob.Model.Vocabulary.Functions.EncodingFunctions package

Submodules

netzob.Model.Vocabulary.Functions.EncodingFunctions.Base64EncodingFunction module

netzob.Model.Vocabulary.Functions.EncodingFunctions.DomainEncodingFunction module

netzob.Model.Vocabulary.Functions.EncodingFunctions.TypeEncodingFunction module

netzob.Model.Vocabulary.Functions.EncodingFunctions.ZLibEncodingFunction module

netzob.Model.Vocabulary.Functions.EncodingFunctions.all module

Module contents

netzob.Model.Vocabulary.Functions.VisualizationFunctions package

Submodules

netzob.Model.Vocabulary.Functions.VisualizationFunctions.HighlightFunction module

class **HighlightFunction**(*start*, *end*)

Bases: *netzob.Model.Vocabulary.Functions.VisualizationFunction.VisualizationFunction*

Represents a function which applies to modify the visualiation attributes of a data

TAG_END = '\x1b[1;m'

TAG_START = '\x1b[1;41m'

TYPE = 'HighlightFunction'

getTags()

netzob.Model.Vocabulary.Functions.VisualizationFunctions.all module

Module contents

Submodules

netzob.Model.Vocabulary.Functions.EncodingFunction module

class EncodingFunction

Bases: *netzob.Common.Utils.SortableObject.SortableObject*

Represents a function which applies to modify the encoding of a data.

The application of these functions is priorized using a SortedTypedList, hence every filter needs to set their application priority.

static getDefaultEncodingFunction()

Default encoding function applied when the raw data needs to be encoded and when no specific filter is specified by the user.

priority()

Returns the priority of the current encoding filter.

netzob.Model.Vocabulary.Functions.FunctionApplicationTable module

class FunctionApplicationTable(*splittedData*)

Bases: object

applyFunction(*function*, *i_start*, *i_end*)

getInitialConversionAddressingTable()

getResult()

getSegments(*i_start*, *i_end*)

getTags(*col*, *i_local*)

insertTagInEncoded(*col*, *i_local*, *i_global*, *tag*, *currentValue*)

registerTag(*i_col*, *idTag*, *i*, *tag*)

updateConversionAddressingTable(*old_start*, *old_end*, *new_start*, *new_end*)

updateConversionAddressingTableWithTable(*table*)

netzob.Model.Vocabulary.Functions.TransformationFunction module

class `TransformationFunction`
Bases: `object`

Represents a function which applies to transform the data

netzob.Model.Vocabulary.Functions.VisualizationFunction module

class `VisualizationFunction`(*start*, *end*)
Bases: `object`

Represents a function which applies to modify the visualiation attributes of a data

`TYPE` = 'VisualizationFunction'

`getTags`()

netzob.Model.Vocabulary.Functions.all module

Module contents

netzob.Model.Vocabulary.Messages package

Submodules

netzob.Model.Vocabulary.Messages.AbstractMessage module

netzob.Model.Vocabulary.Messages.FileMessage module

netzob.Model.Vocabulary.Messages.L2NetworkMessage module

netzob.Model.Vocabulary.Messages.L3NetworkMessage module

netzob.Model.Vocabulary.Messages.L4NetworkMessage module

netzob.Model.Vocabulary.Messages.RawMessage module

netzob.Model.Vocabulary.Messages.all module

Module contents

netzob.Model.Vocabulary.Types package

Submodules

netzob.Model.Vocabulary.Types.ASCII module

netzob.Model.Vocabulary.Types.AbstractType module

netzob.Model.Vocabulary.Types.BitArray module

netzob.Model.Vocabulary.Types.HexaString module

netzob.Model.Vocabulary.Types.IPv4 module

netzob.Model.Vocabulary.Types.Integer module

netzob.Model.Vocabulary.Types.Raw module

netzob.Model.Vocabulary.Types.Timestamp module

netzob.Model.Vocabulary.Types.TypeConverter module

netzob.Model.Vocabulary.Types.all module

Module contents

Submodules

netzob.Model.Vocabulary.AbstractField module

netzob.Model.Vocabulary.ApplicativeData module

class **ApplicativeData**(*name*, *value*, *_id=None*)

Bases: `object`

An applicative data represents an information used over the application that generated the captured flows. It can be the player name or the user email address if these informations are used somehow by the protocol.

An applicative data can be created out of any information. >>> from netzob.all import * >>> app = ApplicativeData("Username", ASCII("toto")) >>> print(app.name) Username

```
>>> app1 = ApplicativeData("Email", ASCII("contact@netzob.org"))
>>> print(app1.value)
ASCII=contact@netzob.org ((0, 144))
```

id

The unique id of the applicative data.

Type `uuid.UUID`

name

The name of the applicative data.

Type `str`

value

The value of the applicative data.

Type object

netzob.Model.Vocabulary.ChannelDownSymbol module

netzob.Model.Vocabulary.EmptySymbol module

netzob.Model.Vocabulary.Field module

netzob.Model.Vocabulary.Session module

netzob.Model.Vocabulary.Symbol module

netzob.Model.Vocabulary.UnknownSymbol module

netzob.Model.Vocabulary.all module

Module contents

Submodules

netzob.Model.Protocol module

netzob.Model.all module

Module contents

netzob.Simulator package

Subpackages

netzob.Simulator.Channels package

Submodules

netzob.Simulator.Channels.AbstractChannel module

class **AbstractChannel** (*isServer*, *_id=UUID('b2245510-f6b2-4f5a-9dc3-d2d42a1ab302')*)

Bases: `object`

DEFAULT_WRITE_COUNTER_MAX = -1

TYPE_IPCLIENT = 2

TYPE_RAWETHERNETCLIENT = 3

TYPE_RAWIPCLIENT = 1

TYPE_SSLCLIENT = 4

TYPE_TCPCLIENT = 5

TYPE_TCPSERVER = 6

TYPE_UDPCLIENT = 7

TYPE_UDPSERVER = 8

TYPE_UNDEFINED = 0

channelType
Returns if the communication channel type

Returns the type of the communication channel

Type int

clearWriteCounter()
Reset the writings counter.

close()
Close the communication channel.

static getLocalIP(*remoteIP*)
Retrieve the source IP address which will be used to connect to the destination IP address.

static getLocalInterface(*localIP*)
Retrieve the network interface name associated with a specific IP address.

id
the unique identifier of the channel

Type uuid.UUID

isOpen
Returns if the communication channel is open

Returns the status of the communication channel

Type bool

isServer
isServer indicates if this side of the channel plays the role of a server.

Type bool

open(*timeout=None*)
Open the communication channel. If the channel is a server, it starts to listen and will create an instance for each different client.

Parameters **timeout** – the maximum time to wait for a client to connect

read(*timeout=None*)
Read the next message on the communication channel.

@keyword timeout: the maximum time in millisecond to wait before a message can be reached @type timeout: int

sendReceive(*data*, *timeout=None*)
Write on the communication channel the specified data and returns the corresponding response

Parameters **data** (*binary object*) – the data to write on the channel

@type timeout: int

setWriteCounterMax(*maxValue*)
Change the max number of writings. When it is reached, no packet can be sent anymore until clearWriteCounter() is called. if maxValue==-1, the sending limit is deactivated.

Parameters **maxValue** (*int*) – the new max value

write(*data*, *rate=None*, *duration=None*)
Write on the communication channel the specified data

Parameters

- **data** (*bytes object*) – the data to write on the channel
- **rate** (*int*) – specifies the bandwidth in octets to respect during traffic emission (should be used with duration= parameter)
- **duration** (*int*) – tells how much seconds the symbol is continuously written on the channel
- **duration** – tells how much time the symbol is written on the channel

writePacket(*data*)
Write on the communication channel the specified data

Parameters **data** (*binary object*) – the data to write on the channel

exception ChannelDownException
Bases: Exception

netzob.Simulator.Channels.IPClient module

netzob.Simulator.Channels.RawEthernetClient module

netzob.Simulator.Channels.RawIPClient module

netzob.Simulator.Channels.SSLClient module

class SSLClient(*remoteIP*, *remotePort*, *localIP=None*, *localPort=None*, *timeout=2*, *server_cert_file=None*, *alpn_protocols=None*)
Bases: *netzob.Simulator.Channels.AbstractChannel.AbstractChannel*

An SSLClient is a communication channel that relies on SSL. It allows to create client connecting to a specific IP:Port server over a TCP/SSL socket.

When the actor execute an OpenChannelTransition, it calls the open method on the ssl client which connects to the server.

close()
Close the communication channel.

localIP
IP on which the server will listen.

Type str

localPort
TCP Port on which the server will listen. Its value must be above 0 and under 65535.

Type int

open(*timeout=None*)
Open the communication channel. If the channel is a client, it starts to connect to the specified server.

read(*timeout=None*)
Read the next message on the communication channel.

@keyword timeout: the maximum time in millisecond to wait before a message can be reached @type timeout: `int`

remoteIP
IP on which the server will listen.

Type `str`

remotePort
TCP Port on which the server will listen. Its value must be above 0 and under 65535.

Type `int`

sendReceive(*data*, *timeout=None*)
Write on the communication channel the specified data and returns the corresponding response.

timeout

writePacket(*data*)
Write on the communication channel the specified data

Parameters **data** (*binary object*) – the data to write on the channel

netzob.Simulator.Channels.TCPClient module

class TCPClient(*remoteIP*, *remotePort*, *localIP=None*, *localPort=None*, *timeout=5*)
Bases: *netzob.Simulator.Channels.AbstractChannel.AbstractChannel*

A TCPClient is a communication channel. It allows to create client connecting to a specific IP:Port server over a TCP socket.

When the actor execute an OpenChannelTransition, it calls the open method on the tcp client which connects to the server.

```
>>> from netzob.all import *
>>> import time
>>> client = TCPClient(remoteIP='127.0.0.1', remotePort=9999)
```

```
>>> symbol = Symbol([Field("Hello everyone!")])
>>> s0 = State()
>>> s1 = State()
>>> s2 = State()
>>> openTransition = OpenChannelTransition(startState=s0, endState=s1)
>>> mainTransition = Transition(startState=s1, endState=s1, inputSymbol=symbol,␣
↪outputSymbols=[symbol])
>>> closeTransition = CloseChannelTransition(startState=s1, endState=s2)
>>> automata = Automata(s0, [symbol])
```

```
>>> channel = TCPServer(localIP="127.0.0.1", localPort=8885)
>>> abstractionLayer = AbstractionLayer(channel, [symbol])
>>> server = Actor(automata = automata, initiator = False,␣
↪abstractionLayer=abstractionLayer)
```

```
>>> channel = TCPClient(remoteIP="127.0.0.1", remotePort=8885)
>>> abstractionLayer = AbstractionLayer(channel, [symbol])
>>> client = Actor(automata = automata, initiator = True,␣
↪abstractionLayer=abstractionLayer)
```

```
>>> server.start()
>>> client.start()
```

```
>>> time.sleep(1)
>>> client.stop()
>>> server.stop()
```

close()
Close the communication channel.

localIP
IP on which the server will listen.

Type str

localPort
TCP Port on which the server will listen. Its value must be above 0 and under 65535.

Type int

open(*timeout=None*)
Open the communication channel. If the channel is a client, it starts to connect to the specified server.

read(*timeout=None*)
Reads the next message on the communication channel. Continues to read while it receives something.

@keyword timeout: the maximum time in millisecond to wait before a message can be reached @type timeout: int

remoteIP
IP on which the server will listen.

Type str

remotePort
TCP Port on which the server will listen. Its value must be above 0 and under 65535.

Type int

sendReceive(*data*, *timeout=None*)
Write on the communication channel the specified data and returns the corresponding response.

timeout

writePacket(*data*)
Write on the communication channel the specified data

Parameters **data** (*binary object*) – the data to write on the channel

netzob.Simulator.Channels.TCPServer module

class TCPServer(*localIP*, *localPort*, *timeout=5*)
Bases: *netzob.Simulator.Channels.AbstractChannel.AbstractChannel*

A TCPServer is a communication channel. It allows to create server listening on a specified IP:Port over a TCP socket.

When the actor execute an OpenChannelTransition, it calls the open method on the tcp server which starts the server. The objective of the server is to wait for the client to connect.

```
>>> from netzob.all import *
>>> import time
>>> server = TCPServer(localIP='127.0.0.1', localPort=9999)
```

```
>>> symbol = Symbol([Field("Hello everyone!")])
>>> s0 = State()
>>> s1 = State()
>>> s2 = State()
>>> openTransition = OpenChannelTransition(startState=s0, endState=s1)
>>> mainTransition = Transition(startState=s1, endState=s1, inputSymbol=symbol,␣
↪outputSymbols=[symbol])
>>> closeTransition = CloseChannelTransition(startState=s1, endState=s2)
>>> automata = Automata(s0, [symbol])
```

```
>>> channel = TCPServer(localIP="127.0.0.1", localPort=8886)
>>> abstractionLayer = AbstractionLayer(channel, [symbol])
>>> server = Actor(automata = automata, initiator = False,␣
↪abstractionLayer=abstractionLayer)
```

```
>>> channel = TCPClient(remoteIP="127.0.0.1", remotePort=8886)
>>> abstractionLayer = AbstractionLayer(channel, [symbol])
>>> client = Actor(automata = automata, initiator = True,␣
↪abstractionLayer=abstractionLayer)
```

```
>>> server.start()
>>> client.start()
```

```
>>> time.sleep(1)
>>> client.stop()
>>> server.stop()
```

close()
Close the communication channel.

localIP
IP on which the server will listen.

Type str

localPort
TCP Port on which the server will listen. Its value must be above 0 and under 65535.

Type int

open(*timeout=None*)
Open the communication channel. If the channel is a server, it starts to listen and will create an instance for each different client

read(*timeout=None*)
Read the next message on the communication channel.

@keyword timeout: the maximum time in millisecond to wait before a message can be reached @type timeout: int

sendReceive(*data*, *timeout=None*)
Write on the communication channel the specified data and returns the corresponding response.

timeout

writePacket(*data*)
Write on the communication channel the specified data

Parameters **data** (*binary object*) – the data to write on the channel

netzob.Simulator.Channels.UDPClient module

class UDPClient(*remoteIP*, *remotePort*, *localIP=None*, *localPort=None*, *timeout=5*)
Bases: *netzob.Simulator.Channels.AbstractChannel.AbstractChannel*

A UDPClient is a communication channel. It allows to create client connecting to a specific IP:Port server over a UDP socket.

When the actor executes an OpenChannelTransition, it calls the open method on the UDP client which connects to the server.

```
>>> from netzob.all import *
>>> import time
>>> client = UDPClient(remoteIP='127.0.0.1', remotePort=9999)
>>> client.open()
>>> client.close()
```

```
>>> symbol = Symbol([Field("Hello everyone!")])
>>> s0 = State()
>>> s1 = State()
>>> s2 = State()
>>> openTransition = OpenChannelTransition(startState=s0, endState=s1)
>>> mainTransition = Transition(startState=s1, endState=s1, inputSymbol=symbol,␣
↪outputSymbols=[symbol])
>>> closeTransition = CloseChannelTransition(startState=s1, endState=s2)
>>> automata = Automata(s0, [symbol])
```

```
>>> channel = UDPServer(localIP="127.0.0.1", localPort=8883)
>>> abstractionLayer = AbstractionLayer(channel, [symbol])
>>> server = Actor(automata = automata, initiator = False,␣
↪abstractionLayer=abstractionLayer)
```

```
>>> channel = UDPClient(remoteIP="127.0.0.1", remotePort=8883)
>>> abstractionLayer = AbstractionLayer(channel, [symbol])
>>> client = Actor(automata = automata, initiator = True,␣
↪abstractionLayer=abstractionLayer)
```

```
>>> server.start()
>>> client.start()
```

```
>>> time.sleep(2)
>>> client.stop()
>>> server.stop()
```

close()
Close the communication channel.

localIP
IP on which the server will listen.

Type str

localPort
UDP Port on which the server will listen. Its value must be above 0 and under 65535.

Type int

open(*timeout=None*)
Open the communication channel. If the channel is a client, it starts to connect to the specified server.

read(*timeout=None*)
Read the next message on the communication channel.

@keyword timeout: the maximum time in millisecond to wait before a message can be reached @type timeout: int

remoteIP
IP on which the server will listen.

Type str

remotePort
UDP Port on which the server will listen. Its value must be above 0 and under 65535.

Type int

sendReceive(*data*, *timeout=None*)
Write on the communication channel the specified data and returns the corresponding response.

timeout

writePacket(*data*)
Write on the communication channel the specified data

Parameters **data** (*binary object*) – the data to write on the channel

netzob.Simulator.Channels.UDPServer module

class UDPServer(*localIP*, *localPort*, *timeout=5*)
Bases: *netzob.Simulator.Channels.AbstractChannel.AbstractChannel*

A UDPServer is a communication channel. It allows to create a server that listen to a specific IP:Port over a UDP socket.

When the actor executes an OpenChannelTransition, it calls the open method on the UDP server which makes it to listen for incomming messages.

```
>>> from netzob.all import *
>>> import time
>>> server = UDPServer(localIP='127.0.0.1', localPort=9999)
>>> server.open()
>>> server.close()
```

```
>>> symbol = Symbol([Field("Hello everyone!")])
>>> s0 = State()
>>> s1 = State()
>>> s2 = State()
>>> openTransition = OpenChannelTransition(startState=s0, endState=s1)
>>> mainTransition = Transition(startState=s1, endState=s1, inputSymbol=symbol,␣
↪outputSymbols=[symbol])
>>> closeTransition = CloseChannelTransition(startState=s1, endState=s2)
>>> automata = Automata(s0, [symbol])
```

```
>>> channel = UDPServer(localIP="127.0.0.1", localPort=8884)
>>> abstractionLayer = AbstractionLayer(channel, [symbol])
>>> server = Actor(automata = automata, initiator = False,␣
↪abstractionLayer=abstractionLayer)
```

```
>>> channel = UDPClient(remoteIP="127.0.0.1", remotePort=8884)
>>> abstractionLayer = AbstractionLayer(channel, [symbol])
>>> client = Actor(automata = automata, initiator = True,␣
↪abstractionLayer=abstractionLayer)
```

```
>>> server.start()
>>> client.start()
```

```
>>> time.sleep(1)
>>> client.stop()
>>> server.stop()
```

close()
Close the communication channel.

localIP
IP on which the server will listen.

Type str

localPort
UDP Port on which the server will listen. Its value must be above 0 and under 65535.

Type int

open(*timeout=None*)
Open the communication channel. This will open a UDP socket that listen for incomming messages.

read(*timeout=None*)
Read the next message on the communication channel.

@keyword timeout: the maximum time in millisecond to wait before a message can be reached @type timeout: int

sendReceive(*data*, *timeout=None*)
Write on the communication channel the specified data and returns the corresponding response.

timeout

writePacket(*data*)
Write on the communication channel the specified data

Parameters **data** (*binary object*) – the data to write on the channel

netzob.Simulator.Channels.all module

Module contents

Submodules

netzob.Simulator.AbstractionLayer module

netzob.Simulator.Actor module

netzob.Simulator.all module

Module contents

Submodules

netzob.NetzobInteractiveSessionController module

class NetzobIPythonShellController
Bases: *netzob.NetzobInteractiveSessionController.NetzobInteractiveSessionController*

Execute Netzob in a IPython embedded shell

start()

class NetzobInteractiveSessionController
Bases: object

Execute Netzob in an Interactive Session

DEFAULT_INTERPRETOR = 'python -i'

getBanner()
getBanner: Computes and returns a string which includes the banner to display on the interpretor startup.
@return L{str}

start()

class NetzobSessionControllerFactory
Bases: object

netzob.NetzobResources module

netzob.all module

netzob.release module

keywords = ['Protocol', 'Inference', 'Networking', 'Reverse Engineering', 'Fuzzing', 'Security']
@deprecated: the official long description is now the full README.rst file

Module contents

CHAPTER 5

Developer Guide

See how you can contribute to Netzob

CHAPTER 6

Indices and tables

- Table of content
- genindex
- modindex
- search

CHAPTER 7

Licences

Netzob code in provided under the GPLv3 licence.

The documentation is under the CC-BY-SA licence.

Python Module Index

n

Index

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W