nfstream

Release 3.2.0

Feb 20, 2020

Table of Contents:

nfstream is a Python package providing fast, flexible, and expressive data structures designed to make working with **online** or **offline** network data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, **real world** network data analysis in Python. Additionally, it has the broader goal of becoming **a common network data processing framework for researchers** providing data reproducibility across experiments.

Main Features

- **Performance: nfstream** is designed to be fast (x10 faster with pypy3 support) with a small CPU and memory footprint.
- **Layer-7 visibility: nfstream** deep packet inspection engine is based on nDPI library. It allows nfstream to perform reliable encrypted applications identification and metadata extraction (e.g. TLS, SSH, DNS, HTTP).
- **Flexibility:** add a flow feature in 2 lines as an NFPlugin.
- **Machine Learning oriented:** add your trained model as an NFPlugin.

CHAPTER 1

Installing nfstream

1.1 Prerequisites

```
apt-get install libpcap-dev
```

1.2 Installation

using pip

Binary installers for the latest released version are available:

```
pip3 install nfstream
```

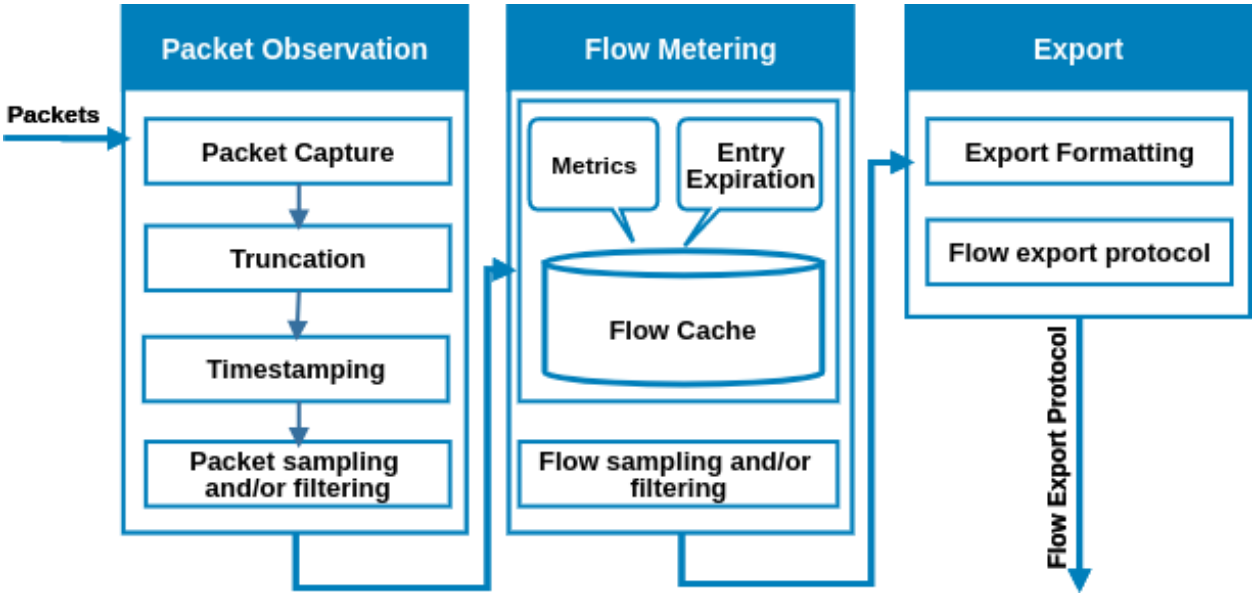
from source

If you want to build nfstream on your local machine:

```
apt-get install autogen
git clone https://github.com/aouinizied/nfstream.git
cd nfstream
python3 setup.py install
```

CHAPTER 2

Architecture



A step by step walk through each process involved when performing flow monitoring is developed in this section. Our aim is to provide you with a reminder about how things works in theory. Consequently, an easier understanding of nfstream features and implementation is possible.

2.1 Packet observation

Packet observation is a key stage in a flow monitoring architecture as it is the starting point. Consequently, we detail in the following each step involved at this phase:

Packet capture: This step is performed on the Network Interface Card (NIC) level. After passing various checks such as checksum error, packets stored in on-card reception buffers are moved to the hosting device memory. Several libraries are available to capture network traffic such as libpcap for UNIX based operating systems Winpcap for

Windows. These libraries are running on the top of the operating system stack which may reduce performances passing through several layers. To overcome such limitation in a high speed network context, software optimization technique are proposed and could be considered (e.g. Intel DPDK, PF-RING, netmap).

Timestamping: As packets may come from several observation points, reordering process is based on packet's timestamp. While hardware timestamping provides a high accuracy up to 100 nanoseconds in case of the IEEE 1588 protocol, it's not supported by most of commodity NIC. Software timestamping is widely used to outcome this lack providing an accuracy up to 100 microseconds.

Truncation (optional): Defining a snapshot length, the process selects precise bytes from the packet. It is performed in some cases to reduce the amount of data captured by the probe and therefore CPU and bus bandwidth load.

Packet sampling (optional): is generally performed to reduce load on subsequent stages. It can be systematic (periodic sampling scheme) or random. The latter is recommended as periodic scheme may introduce unwanted correlation in the observed network data.

Packet filtering (optional): performs filtering of packets to separate packets having specific properties from those not having them. A packet is selected if some specific fields are equal or in the range of given values. Another technique is a hash based filtering, applying a hash function on a portion of the packet, the result is compared to a value or a range of values.

2.2 Flow metering

It includes packets aggregation into flows and flow entry expiration management. Second, the metering process associates a packet to a flow entry using a defined key. Third, it performs the aggregation of packets into flow entry based on a set of metrics. Then, a flow entry is cached until it is considered as terminated (entry expiration). Finally, optional steps such as flow sampling and filtering may be performed.

Flow Cache: Flow cache consist of table in which the metering process stores information regarding active flows in the network. A flow key (typically IP source and destination addresses, source and destination ports, protocol and the VLAN identifier) determines whether a packet is matching an existing flow entry in the cache or not. In the first case, flow's counters are updated. In the latter one, a new entry is created. Non-key fields are utilized to collect flow metrics (e.g. packets/bytes count, etc.). If IP addresses are part of flows key, and that traffic between two pairs generates flows on both directions. We define a flow as bidirectional when we consider that pair and it reverse belongs to same entry.The cache's size depends on exporter device memory capacity and should be configured based on criteria such as key/non-key fields, maximum number of flows expected and expiration policy.

Entry expiration: Cache entries are maintained in the cache table until they are considered as terminated. Termination of a flow is triggered by an expiration event. The metering process should consider an entry as expired based on:

- Natural expiration: observed TCP packet belonging to a flow with FIN/RST flag.
- Emergency expiration: flush a certain number of entries to free some space when the cache become full.
- Active timeout: a flow entry expires after being considered active during a certain period (range from 120 seconds to 30 minutes). Counters are reset while start/end timestamp are updated.
- Idle timeout: a flow entry expires if no packets belonging to it are observed during a specific period (range from 15 seconds to 5 minutes).
- Resource constraints: special heuristics such as dynamic timeouts configuration at runtime.
- Cache flush: flush of all the entries due to unexpected situations.

It is possible to configure our metering process based on expiration policy to reduce the amount of records exported.

Flow Sampling and Filtering: Flow sampling and filtering processes are quite like packet sampling and filtering process explained above. The major differences are the processed unit; while packet sampling and filtering process packets, flow sampling and filtering process flow records coming from the metering process.

2.3 Export

Export involves two steps which are mainly **formatting** and **export protocol**. While the first decide how an export is formatted (number of flow per export, json or other, etc.), the latter determine the used export protocol (file, mqtt, zmq, etc.).

CHAPTER 3

Get started with nfstream

Dealing with a big pcap file and just want to aggregate it as network flows? nfstream make this path easier in few lines:

```
from nfstream import NFStreamer
my_capture_streamer = NFStreamer(source="facebook.pcap",
                                 snaplen=65535,
                                 idle_timeout=30,
                                 active_timeout=300,
                                 plugins=(),
                                 dissect=True,
                                 max_tcp_dissections=10,
                                 max_udp_dissections=16)

my_live_streamer = NFStreamer(source="eth1")  # or capture from a network interface
for flow in my_capture_streamer:  # or for flow in my_live_streamer
    print(flow)  # print, append to pandas Dataframe or whatever you want :)!
```

3.1 NFStreamer object

- `source` [default= `None`]
  - Source of packets. Can be `live_interface_name` or `pcap_file_path`.
- `snaplen` [default= `65535`]
  - Packet capture length.
- `idle_timeout` [default= `30`]
  - Flows that are inactive for more than this value in seconds will be exported.
- `active_timeout` [default= `300`]
  - Flows that are active for more than this value in seconds will be exported.

- `plugins` [default= `()`]
  - Set of user defined NFPlugins.
- `dissect` [default= `True`]
  - Enable nDPI deep packet inspection library for Layer 7 visibility.
- `max_tcp_dissections` [default= `10`]
  - Maximum per flow TCP packets to dissect (ignored when dissect=False).
- `max_udp_dissections` [default= `16`]
  - Maximum per flow UDP packets to dissect (ignored when dissect=False).

NFStreamer returns an iterator of **NFEntry** object.

3.2 NFEntry object

Table 1: NFEntry object

attribute name	attribute type	attribute description
id	int	Flow identifier.
first_seen	int	First packet timestamp in milliseconds.
last_seen	int	Last packet timestamp in milliseconds.
version	int	IP version.
src_port	int	Transport layer source port.
dst_port	int	Transport layer destination port.
protocol	int	Transport layer protocol.
vlan_id	int	Virtual LAN identifier.
src_ip	str	Source IP address string representation.
dst_ip	str	Destination IP address string representation.
ip_src	int	Source IP address int value. [`volatile`]
ip_dst	int	Destination IP address int value. [`volatile`]
total_packets	int	Flow packets accumulator.
total_bytes	int	Flow bytes (full packet lentgh) accumulator.
duration	int	Flow duration in milliseconds.
src2dst_packets	int	Flow packets accumulator (source->destination).
src2dst_bytes	int	Flow bytes (full packet lentgh) accumulator (source->destination).
dst2src_packets	int	Flow packets accumulator (destination->source).
dst2src_bytes	int	Flow bytes (full packet lentgh) accumulator (destination->source).
expiration_id	int	Identifier of flow expiration trigger. Can be `0` for idle_timeout, `1` for active_timeout or 'negative' for custom expiration.
master_protocol	int	nDPI master protocol identifier.
app_protocol	int	nDPI app protocol identifier.
application_name	str	nDPI application name.
category_name	str	nDPI application category name.
client_info	str	Dissected client informations. Can be `http_detected_os` for HTTP, `client_signature` for SSH or `ssl_client_hello_sni` for SSL.
client_info	str	Dissected server informations. Can be `host_server_name` for HTTP or DNS, `server_signature` for SSH or `ssl_server_hello_sni` for SSL.
j3a_client	str	J3A client fingerprint.
j3a_server	str	J3A server fingerprint.

NFEntry is an aggregation of **NFPacket** objects.

3.3 NFPacket object

Table 2: NFPacket object

attribute name	attribute type	attribute description
time	int	Packet timestamp in milliseconds.
capture_length	int	Packet capture length.
length	int	Packet size.
ip_src	int	Source IP address int value.
ip_dst	int	Destination IP address int value.
src_port	int	Transport layer source port.
dst_port	int	Transport layer destination port.
protocol	int	Transport layer protocol.
vlan_id	int	Virtual LAN identifier.
version	int	IP version.
tcp_flags	int	Packet observed TCP flags.
raw	bytes	Raw content starting from IP Header.
direction	int	Packet direction: `0` for src_to_dst and `1` for dst_to_src.

CHAPTER 4

Extending nfstream

nfstream is designed to be flexible and machine learning oriented. In the following section, we depict the use of NFPlugin in both cases.

```
from nfstream import NFPlugin

class my_awesome_plugin(NFPlugin):
    def on_update(self, obs, entry):
        if obs.length >= 666:
            entry.my_awesome_plugin += 1


streamer_awesome = NFStreamer(source='devil.pcap', plugins=[my_awesome_plugin()])
for flow in streamer_awesome:
   print(flow.my_awesome_plugin) # now you will see your dynamically created metric␣
↪in generated flows
```

4.1 NFPlugin parameters

- `name` [default= `class name`]
  - Plugin name. Must be unique as it's dynamically created as a flow attribute.
- `volatile` [default= `False`]
  - Volatile plugin is available only when flow is processed. At flow expiration level, plugin is automatically removed (will not appear as flow attribute).
- `user_data` [default= `None`]
  - user_data passed to the plugin. Example: external module, pickled sklearn model, etc.

4.2 NFPlugin methods

- `on_init(self, obs)` [default= `return 0`]
  - Method called at entry creation). When aggregating packets into flows, this method is called on `NFEntry` object creation based on first `NFPacket` object belonging to it.
- `on_update(self, obs, entry)` [default= `pass`]
  - Method called to update each entry with its belonging obs. When aggregating packets into flows, the entry is an `NFEntry` object and the obs is an `NFPacket` object.
- `on_expire(self, entry)` [default= `pass`]
  - Method called at entry expiration. When aggregating packets into flows, the entry is an `NFEntry`
- `cleanup(self)` [default= `pass`]
  - Method called for plugin cleanup.

In the following, we want to run an early classification of flows based on a trained machine learning model than takes as features the 3 first packets size of a flow.

4.3 Computing required features

```
from nfstream import NFPlugin

class feat_1(NFPlugin):
    def on_update(self, obs, entry):
        if entry.total_packets == 1:
            entry.feat_1 == obs.length

class feat_2(NFPlugin):
    def on_update(self, obs, entry):
        if entry.total_packets == 2:
            entry.feat_2 == obs.length

class feat_3(NFPlugin):
    def on_update(self, obs, entry):
        if entry.total_packets == 3:
            entry.feat_3 == obs.length
```

4.4 Trained model prediction

```
class model_prediction(NFPlugin):
    def on_update(self, obs, entry):
        if entry.total_packets == 3:
            entry.model_prediction = self.user_data.predict_proba([entry.feat_1 ,␣
↪entry.feat_2 , entry.feat_3])
            # optionally we can force NFStreamer to immediately expires the flow
            # entry.expiration_id = -1
```

4.5 Start your new streamer

```
my_model = function_to_load_your_model() # or whatever
ml_streamer = NFStreamer(source='devil.pcap',
                         plugins=[feat_1(volatile=True),
                                  feat_2(volatile=True),
                                  feat_3(volatile=True),
                                  model_prediction(user_data=my_model)
                                  ])
for flow in ml_streamer:
    print(flow.model_prediction) # now you will see your trained model prediction as␣
↪part of the flow :)
```

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of contribution

Report bugs

Report bugs at https://github.com/aouinizied/nfstream/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.
- pcap file if you are reporting a bug on offline mode

Fix bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write documentation

nfstream could always use more documentation, whether as part of the official nfstream docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit feedback

The best way to send feedback is to file an issue at https://github.com/aouinizied/nfstream/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome.

5.2 Get started

Ready to contribute? Here's how to set up nfstream for local development.

1. Fork the nfstream repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/nfstream.git
```

3. Install your local copy into a virtualenv. This is an example how you set up your fork for local development for Python3.6:

```
$ cd nfstream
$ virtualenv venv-nfstream-py36 -p /usr/bin/python3.6
$ source venv-nfstream-py36/bin/activate
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

5. When you're done making changes, check that your changes pass the tests (run it as root to trigger live capture testing):

```
$ python tests.py
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull request guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for 3.6 and 3.7 and 3.8 Check https://travis-ci.org/aouinizied/nfstream/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in /docs/source/changelog.rst). Then run:

```
$ bumpversion patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 6

Changelog

3.2.0 (2020-02-02)

- nDPI 3.2 support.
- Fix metadata extraction issues.

3.1.2 (2020-01-07)

- Fix tests workflow.
- Update nDPI (commit: 73c7ccdb65a1e13e3fb1726af7882dd34534906f).

3.1.1 (2019-12-29)

- Fix generated wheels. (drop sdist)

3.1.0 (2019-12-29)

- Initial support for nDPI3.1 (commit: 73c7ccdb65a1e13e3fb1726af7882dd34534906f).
- Add wrapping for pandas.
- pypy7.2 support.
- Add py36, py38 for macOS wheels.
- Move continous integration to GitHub Actions.

3.0.4 (2019-12-18)

- Fix pypi description rendering.

3.0.3 (2019-12-18)

- MacOS Catalina support.
- Implement random port selection for zmq.

3.0.2 (2019-12-06)

- ether type double stacking implementation.
- Minor fixes.

3.0.1 (2019-12-04)

- Fix macOS wheels 10.14

3.0.0 (2019-12-04)

- Sync with nDPI major.minor versions.
- New NFPlugin API definition.
- Fix macOS wheels for 10.13 and 10.14

2.0.1 (2019-11-29)

- Fix pypy3 wheel.

2.0.0 (2019-11-28)

- Pypy support.
- Major performances improvements.
- NFPlugin as main extension API.
- nDPI memory usage improved.
- nDPI implemented using cffi.
- tcp_max_dissections, udp_max_dissections options.
- NFFlow dynamic attributes creation.
- HTTP, SSH, DNS client and server informations extraction.
- FlowCache management implemented in pure Python.

1.2.1 (2019-11-15)

- Fix ndpi padding and alignement issues.
- nDPI3.1 compatibility.

1.2.0 (2019-11-14)

- Fix ndpi bindings.
- Add TLS dissection features (server sni, client sni, version, organization, expiration dates)
- Improve documentation.

1.1.8 (2019-11-07)

- Fix ndpi wrap missing fields.
- Add host_server_name metric.
- Update doc.

1.1.7 (2019-11-07)

- Fix minor bugs.

1.1.6 (2019-11-03)

- TCP flags extraction.
- Minor bug fixes.

1.1.5 (2019-11-02)

- Add BPF filtering feature.

- Fix radiotap parsing.

1.1.2-3-4 (2019-11-01)

- Fix broken macos wheels on pypi.

1.1.1 (2019-11-01)

- Fix broken linux wheels on pypi.
- Py38 compatibility.

1.1.0 (2019-11-01)

- Add OSX support.

1.0.1-2-3 (2019-10-31)

- Fix deployment CI

1.0.0 (2019-10-30)

- cffi based packet capture.
- fast parsing mechanism.
- Minor bug fixes.
- auto-generate binaries.

0.5.0 (2019-10-21)

- Classifier mechanism introduced.
- Custom export_reason.
- Fix minor bugs.
- Improve documentation.

0.4.0 (2019-10-20)

- Pypi package description readable.

0.3.1 (2019-10-20)

- Add category_name as flow feature.

0.3.0 (2019-10-20)

- Add user defined callbacks feature.
- Fix live capture handling.
- Fix library loading path.
- Json support for flow printing.
- Add examples.

0.2.0 (2019-10-19)

- Add nDPI bindings as part of the released package
- Documentation improvement

0.1.0 (2019-10-19)

- First release on PyPI.