FlowPrint

Release 1.0.0

Mar 23, 2020

Contents:

FlowPrint introduces a semi-supervised approach for fingerprinting mobile apps from (encrypted) network traffic. We automatically find temporal correlations among destination-related features of network traffic and use these correlations to generate app fingerprints. These fingerprints can later be reused to recognize known apps or to detect previously unseen apps. The main contribution of this work is to create network fingerprints without prior knowledge of the apps running in the network.

CHAPTER 1

Installation

The most straigtforward way of installing FlowPrint is via pip

```
pip install flowprint
```

If you wish to stay up to date with the latest development version, you can instead download the source code. In this case, make sure that you have all the required *dependencies* installed.

Note: Tshark should always be installed, see *tshark*.

1.1 Dependencies

FlowPrint requires the following python packages to be installed:

- Cryptography: https://pypi.org/project/cryptography/
- Matplotlib: https://matplotlib.org/
- NetworkX: https://networkx.github.io/
- Numpy: https://numpy.org
- Pandas: https://pandas.pydata.org/
- Pyshark: https://pypi.org/project/pyshark/
- Scikit-learn: https://scikit-learn.org/stable/index.html

All dependencies should be automatically downloaded if you install FlowPrint via pip. However, should you want to install these libraries manually, you can install the dependencies using the requirements.txt file

```
pip install -r requirements.txt
```

Or you can install these libraries yourself

```
pip install -U cryptography matplotlib networkx numpy pandas pyshark scikit-learn
```

1.1.1 Tshark

Tshark is required for both the raw tshark backend and the pyshark backend. You can install tshark as a stand alone, but it also comes with the wireshark installation. On ubuntu you can install tshark using

```
sudo apt install tshark
```

or

```
sudo apt install wireshark
```

To test whether tshark is active and in your path, please run

```
tshark --version
```

Which should output the current version you are running.

Note: When tshark is not installed, FlowPrint will give a warning message because it tries to use tshark as a backend by default. If tshark cannot be found it falls back on pyshark, which is a lot slower.

CHAPTER 2

Usage

The FlowPrint package offers both a command-line tool for easy access and a rich API for full customisation. This section gives a high-level overview of the different steps taken by FlowPrint to generate fingerprints. We also include several working examples to guide users through the code. For detailed documentation of individual methods, we refer to the *Reference* guide.

2.1 Overview

This section explains on a high level the different steps taken by FlowPrint to create fingerprints and compare them to recognize apps or detect unseen apps.

1) *Flow extraction*
2) *Fingerprint generation*
3) *Fingerprint application*
    a) *App recognition*
    b) *Unseen app detection*

2.1.1 Flow extraction

FlowPrint itself takes as input an array of *Flow* objects. However, we need to extract these flows from the actual network traffic. Currently, FlowPrint extracts these features from .pcap files using the *Preprocessor* object. This module provides the function `preprocessor.Preprocessor.process()` method in which you specify .pcap files and their lables as input and outputs *Flow* objects and their corresponding labels. The *Preprocessor* class uses the *Reader* and *Flow* classes to produce *Flow* objects. These *Flow* objects can be saved and loaded in files using the `preprocessor.Preprocessor.save()` and `preprocessor.Preprocessor.load()` methods respectively. Figure 1 gives an overview of the flow extraction process.

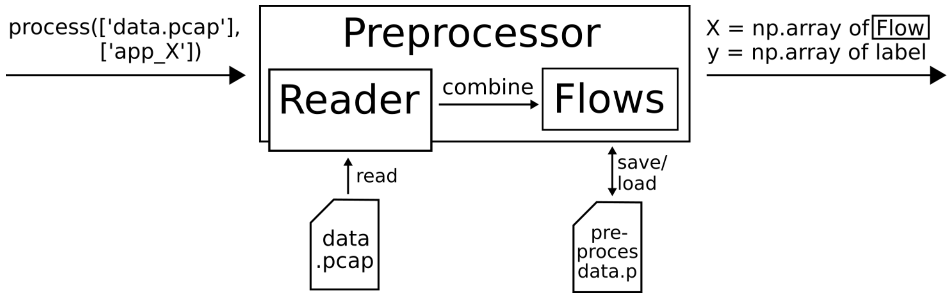

Fig. 1: Figure 1: Overview flow extraction.

2.1.2 Fingerprint generation

After extracting Flows, FlowPrint generates *Fingerprint* objects. We refer to our paper for a detailed overview. The code implements this as described in Figure 2. We see that the entire generation process takes place in the *FingerprintGenerator* object, which uses in order the following classes:

1) *Cluster*
2) *CrossCorrelationGraph*
3) *Fingerprint*

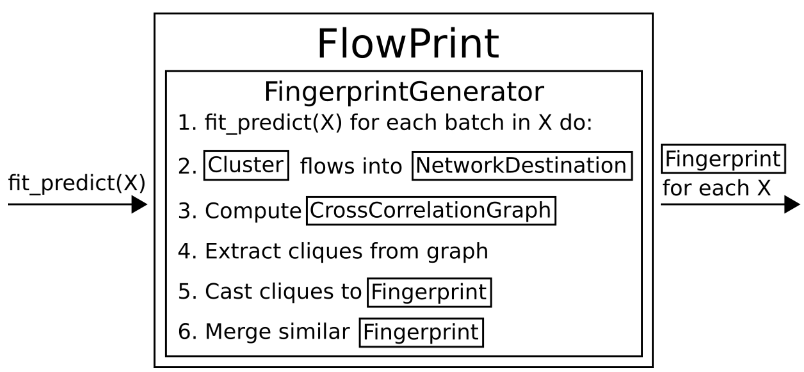

Fig. 2: Figure 2: Overview of fingerprint generation.

2.1.3 Fingerprint application

This library implements FlowPrint's app recognition and unseen app detection applications.

App recognition

To recognize known apps, we simply use *FlowPrint*'s `recognize(X)` method. This method creates new *Fingerprint* objects for the given *Flow* objects `X` and compares them to the fingerprints stored using the `fit()` method. It returns the closest matching fingerprint for each given *Flow* in `X`.

Unseen app detection

To detect unseen apps, we simply use *FlowPrint*'s `detect(X, threshold=0.1)` method. This method creates new *Fingerprint* objects for the given *Flow* objects `X` and compares them to the fingerprints stored using the `fit()`

method. It returns `+1` for each *Flow* in `X` that matches a known fingerprint and `-1` for each *Flow* that does not match known fingerprints.

2.2 Command line tool

When FlowPrint is installed, it can be used from the command line. The `__main__.py` file in the `flowprint` module implements this command line tool. The command line tool provides a quick and easy interface to convert `.pcap` files into *Flow* objects and use these objects to create *Fingerprint*'s. Once generated, the *Fingerprint*'s can be used for app recognition and unseen app detection. The full command line usage is given in its `help` page:

```
usage: flowprint.py [-h]
                  (--detection [FLOAT] | --fingerprint [FILE] | --recognition)
                  [-b BATCH] [-c CORRELATION], [-s SIMILARITY], [-w WINDOW]
                  [-p PCAPS...] [-rp READ...] [-wp WRITE]

Flowprint: Semi-Supervised Mobile-App
Fingerprinting on Encrypted Network Traffic

Arguments:
  -h, --help                 show this help message and exit

FlowPrint mode (select up to one):
  --fingerprint [FILE]       run in raw fingerprint generation mode (default)
                             outputs to terminal or json FILE
  --detection   FLOAT        run in unseen app detection mode with given
                             FLOAT threshold
  --recognition              run in app recognition mode

FlowPrint parameters:
  -b, --batch       FLOAT    batch size in seconds       (default=300)
  -c, --correlation FLOAT    cross-correlation threshold (default=0.1)
  -s, --similarity  FLOAT    similarity threshold        (default=0.9)
  -w, --window      FLOAT    window size in seconds      (default=30)

Flow data input/output (either --pcaps or --read required):
  -p, --pcaps  PATHS...      path to pcap(ng) files to run through FlowPrint
  -r, --read   PATHS...      read preprocessed data from given files
  -o, --write  PATH          write preprocessed data to given file
  -i, --split  FLOAT         fraction of data to select for testing (default= 0)
  -a, --random FLOAT         random state to use for split          (default=42)

Train/test input (for --detection/--recognition):
  -t, --train PATHS...       path to json files containing training fingerprints
  -e, --test  PATHS...       path to json files containing testing fingerprints
```

2.2.1 Examples

Transform `.pcap` files into *flows* and store them in a file.

```
python3 -m flowprint --pcaps <data.pcap> --write <flows.p>
```

Extract `fingerprints` from `flows`, split them into training and testing, and store the fingerprints into a file.

```
python3 -m flowprint --read <flows.p> --fingerprint <fingerprints.json>
```

Use FlowPrint to recognize apps or detect previously unknown apps

```
python3 -m flowprint --train <fingerprints.train.json> --test <fingerprints.test.json>
↪ --recognition
python3 -m flowprint --train <fingerprints.train.json> --test <fingerprints.test.json>
↪ --detection 0.1
```

2.3 Code integration

To integrate FlowPrint into your own project, you can use it as a standalone module. FlowPrint offers rich functionality that is easy to integrate into other projects. Here we show some simple examples on how to use the FlowPrint package in your own python code. For a complete documentation we refer to the *Reference* guide.

2.3.1 Import

To import components from FlowPrint simply use the following format

```
from flowprint.<module> import <Object>
```

For example, the following code imports the *FlowPrint* and *Preprocessor* objects.

```
from flowprint.flowprint import FlowPrint
from flowprint.preprocessor import Preprocessor
```

2.3.2 Flow extraction

To extract *Flow* objects from `.pcap` files, we use the *Preprocessor* object.

```
# Imports
from flowprint.preprocessor import Preprocessor

# Create Preprocessor object
preprocessor = Preprocessor(verbose=True)
# Create Flows and labels
X, y = preprocessor.process(files =['a.pcap', 'b.pcap'],
                            labels=['a', 'b'])

# Save flows and labels to file 'flows.p'
preprocessor.save('flows.p', X, y)
# Load flows from file 'flows.p'
X, y = preprocessor.load('flows.p')
```

2.3.3 Fingerprint generation

To generate fingerprints we use the *FlowPrint* object. We assume that the we have training flows and labels in variables `X_train` and `y_train` respectively, and have testing flows in variable `X_test`.

```python
# Imports
from flowprint.flowprint import FlowPrint

# Create FlowPrint object
flowprint = FlowPrint(
    batch       = 300,
    window      = 30,
    correlation = 0.1,
    similarity  = 0.9
)

# Fit FlowPrint with flows and labels
flowprint.fit(X_train, y_train)
# Predict best matching fingerprints for each flow
y_pred = flowprint.predict(X_test)

# Store fingerprints to file 'fingerprints.json'
flowprint.save('fingerprints.json')
# Load fingerprints from file 'fingerprints.json'
# This returns both the fingerprints and stores them in the flowprint object
fingerprints = flowprint.load('fingerprints.json')
```

2.3.4 App recognition and detection

We can also use FlowPrint to recognize known apps or detect previously unseen apps. Again, we assume that the we have training flows and labels in variables `X_train` and `y_train` respectively, and have testing flows in variable `X_test`.

```python
# Imports
from flowprint.flowprint import FlowPrint

# Create FlowPrint object
flowprint = FlowPrint(
    batch       = 300,
    window      = 30,
    correlation = 0.1,
    similarity  = 0.9
)

# Fit FlowPrint with flows and labels
flowprint.fit(X_train, y_train)

# Recognise which app produced each flow
y_recognize = flowprint.recognize(X_test)
# Detect previously unseen apps
# +1 if a flow belongs to a known app, -1 if a flow belongs to an unknown app
y_detect    = flowprint.detect(X_test)
```

CHAPTER 3

Reference

This is the reference documentation for the classes and methods objects provided by the FlowPrint module.

3.1 BrowserDetector

Note: The BrowserDetector is currently not supported in the command line interface nor is it used in the fingerprint generation of the other classes. Currently, this is only supported as a stand-alone API.

The BrowserDetector class is used as a supervised detector to isolate browser Flows from regular app traffic.

class `browser_detector.`**`BrowserDetector`**(*before=10*, *after=10*, *random_state=42*)

Detector for browser application

`classifier`

Random forest classifier used for classifying individual datapoints

Type sklearn.ensemble.RandomForestClassifier

`before`

Time frame in seconds to remove before detected browser

Type float

`after`

Time frame in seconds to remove after detected browser

Type float

`BrowserDetector.`**`__init__`**(*before=10*, *after=10*, *random_state=42*)

Detector for browser application

Parameters

- **`before`** (*`float, default = 10`*) – Time frame in seconds to remove before detected browser

- **after** (*float, default = 10*) – Time frame in seconds to remove after detected browser
- **random_state** (*int, RandomState instance or None, optional, default:*) – None If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*

3.1.1 Browser Detection

We first need to `browser_detector.BrowserDetector.fit()` (train) the BrowserDetector with Flows from both browser and non-browser apps. Next, we can `browser_detector.BrowserDetector.predict()` whether new *Flow*'s are browser or non-browser flows. Or we can do both in a single step using the `browser_detector.BrowserDetector.fit_predict()` method.

BrowserDetector.**fit**(*X*, *y*)

Fit the classifier with browser and non-browser traffic

Parameters

- **X** (*array-like of shape=(n_samples, n_features)*) – Flows to fit the classifier with
- **y** (*array-like of shape=(n_samples,)*) – Array of labels, -1 for non-browser, 1 for browser

Returns **result** – Returns self for fit_predict method

Return type self

BrowserDetector.**predict**(*X*, *y=None*)

Predict whether samples from X are browser: 1 or non_browser: -1

Parameters

- **X** (*array-like of shape=(n_samples, n_features)*) – Flows to predict with the classifier
- **y** (*ignored*) –

Returns **result** – -1 if sample from X is not from browser, 1 if sample from X is from browser

Return type np.array of shape=(n_samples,)

BrowserDetector.**fit_predict**(*X*, *y*)

Fit and predict the samples with the classifier as browser or non-browser traffic

Parameters

- **X** (*array-like of shape=(n_samples, n_features)*) – Flows to fit the classifier with
- **y** (*array-like of shape=(n_samples,)*) – Array of labels, -1 for non-browser, 1 for browser

Returns **result** – -1 if sample from X is not from browser, 1 if sample from X is from browser

Return type np.array of shape=(n_samples,)

3.1.2 Feature extraction

The BrowserDetector uses several features from each Flow to determine whether a Flow was generated by a browser or non-browser app. The `browser_detector.BrowserDetector.features()` method extracts these features.

`BrowserDetector.`**`features`**`(X)`
Returns flow features for determining whether flows are browser

Parameters X (*array-like of shape=(n_samples, n_features)*) – Flows from which to extract features

Returns result – Features for determining browser flows. Currently the features are [clusters', length incoming', length outgoing', ratio incoming/outgoing'] where the ' indicates the derivative

Return type np.array of shape=(n_samples, n_features)

3.2 Cluster

After performing feature extraction, FlowPrint clusters all *Flow*'s into *NetworkDestination* according equal (destination IP, destination port)-tuple or TLS certificates.

`class` `cluster.`**`Cluster`**`(load=None)`
Cluster object for clustering flows by network destination

`samples`
Samples used to fit Cluster

Type np.array of shape=(n_samples,)

`counter`
Counter for total number of NetworkDestinations generated

Type int

`dict_destination`
Dicationary of (dst IP, dst port) -> NetworkDestination

Type dict

`dict_certificate`
Dicationary of TLS certificate -> NetworkDestination

Type dict

`Cluster.`**`__init__`**`(load=None)`
Cluster flows by network destinations

Parameters load (*string, default=None*) – If given, load cluster from json file from 'load' path.

3.2.1 Generating clusters

We can create clusters from *Flow*'s by fitting the Cluster object `cluster.Cluster.fit()` method. After fitting the cluster, we can use the `cluster.Cluster.predict()` method to get all cluster labels as numbers. The `cluster.Cluster.fit_predict()` method combines both methods into a single action.

`Cluster.fit(X, y=None)`
Fit the clustering algorithm with flow samples X.

Parameters

- **X** (*array-like of shape=(n_samples, n_features)*) – Flow samples to fit cluster object.
- **y** (*array-like of shape=(n_samples,), optional*) – If given, add labels to each cluster.

Returns **result** – Returns self

Return type self

`Cluster.predict(X)`
Predict cluster labels of X.

Parameters **X** (*array-like of shape=(n_samples, n_features)*) – Samples for which to predict NetworkDestination cluster.

Returns **result** – Labels of NetworkDestination cluster corresponding to cluster of fitted samples. Has a value of -1 if no cluster could be matched

Return type array-like of shape=(n_samples,)

`Cluster.fit_predict(X)`
Fit and predict cluster with given samples.

Parameters **X** (*array-like of shape=(n_samples, n_features)*) – Samples to fit cluster object.

Returns **result** – Labels of cluster corresponding to cluster of fitted samples. Has a value of -1 if no cluster could be matched.

Return type array-like of shape=(n_samples,)

3.2.2 Cluster views

We extract the different *NetworkDestination*'s generated by the cluster either as a set or as a dictionary of *identifier -> NetworkDestination*.

`Cluster.clusters()`
Return a set of NetworkDestinations in the current cluster object.

Returns **result** – Set of NetworkDestinations in cluster.

Return type set

`Cluster.cluster_dict()`
Return a dictionary of id -> NetworkDestination.

Returns **result** – Dict of NetworkDestination.identifier -> NetworkDestination

Return type dict

3.2.3 I/O methods

A cluster can be saved and loaded for further analysis. Additionally you can get a copy of the current Cluster.

`Cluster.save(outfile)`
Saves cluster object to json file.

Parameters **outfile** (*string*) – Path to json file in which to store the cluster object.

Cluster.**load**(*infile*)
Loads cluster object from json file.

Parameters **infile** (*string*) – Path to json file from which to load the cluster object.

Cluster.**copy**()
Returns a (semi-deep) copy of self. The resulting cluster is a deep copy apart from the samples X. Has a tremendous speedup compared to copy.deepcopy(self)

Returns **result** – Copy of self

Return type *Cluster*

3.2.4 Visualisation

To get a visual representation of the generated clusters we offer the `cluster.Cluster.plot()` method.

Cluster.**plot**(*annotate=False*)
Plot cluster NetworkDestinations.

Parameters **annotate** (*boolean, default=False*) – If True, annotate each cluster

3.3 CrossCorrelationGraph

The CrossCorrelationGraph is used to compute correlations between different `cluster.NetworkDestination`'s and extract cliques.

class cross_correlation_graph.**CrossCorrelationGraph**(*window=30*, *correlation=0.1*)
CrossCorrelationGraph for computing correlation between clusters

window
Threshold for the window size in seconds

Type float

correlation
Threshold for the minimum required correlation

Type float

graph
Cross correlation graph containing all correlations Note that each node in the graph represents an 'activity signature' to avoid duplicates. The NetworkDestinations corresponding to each signature are stored in the 'mapping' attribute.

Type nx.Graph

mapping
NetworkDestinations corresponding to each node in the graph

Type dict

CrossCorrelationGraph.**__init__**(*window=30*, *correlation=0.1*)
CrossCorrelationGraph for computing correlation between clusters

IMPORTANT: The self.graph object is an optimised graph. Each node does not represent a network destination, but represents an activity fingerprint. E.g. when destinations A and B are both only active at time slices 3 and 7, then these destinations are represented by a single node. We use the self.mapping to extract the network destinations from each graph node. This is a huge optimisation for finding cliques as the number of different network destinations theoretically covers the entire IP space, whereas the number of activity fingerprints is bounded by 2^(batch / window), in our work 2^(300/30) = 2^10 = 1024. If these parameters change, the complexity may increase, but never beyond the original bounds. Hence, this optimisation never has a worse time complexity.

Parameters

- **`window`** (*`float, default=30`*) – Threshold for the window size in seconds
- **`correlation`** (*`float, default=0.1`*) – Threshold for the minimum required correlation

3.3.1 Graph creation

We use the *`cross_correlation_graph.CrossCorrelationGraph.fit()`* method to create the CrossCorrelationGraph. Afterwards, we can detect cliques using the *`cross_correlation_graph.CrossCorrelationGraph.predict()`* method. Or do all in one step using the *`cross_correlation_graph.CrossCorrelationGraph.fit_predict()`* method.

`CrossCorrelationGraph.`**`fit`**(*cluster*, *y=None*)

Fit Cross Correlation Graph.

Parameters

- **`cluster`** (`Cluster`) – Cluster to fit graph, cluster must be populated with flows
- **`y`** (*`ignored`*) –

Returns **result** – Returns self

Return type self

`CrossCorrelationGraph.`**`predict`**(*X=None*, *y=None*)

Fit Cross Correlation Graph and return cliques.

Parameters

- **`X`** (*`ignored`*) –
- **`y`** (*`ignored`*) –

Returns **result** – Generator of all cliques in the graph

Return type Generator of cliques

`CrossCorrelationGraph.`**`fit_predict`**(*cluster*, *y=None*)

Fit cross correlation graph with clusters from X and return cliques.

Parameters

- **`cluster`** (`Cluster`) – Cluster to fit graph, cluster must be populated with flows
- **`y`** (*`ignored`*) –

Returns **result** – Generator of all cliques in the graph

Return type Generator of cliques

3.4 Fingerprint

A Fingerprint object holds the fingerprints as generated by FlowPrint. These fingerprints are sets of (dst ip, dst port)-tuples and TLS certificates. Essentially, it extends the frozenset (i.e., an unchangable set) class with methods useful for comparing, identifying, reading and storing fingerprints.

class fingerprint.**Fingerprint**

FlowPrint fingerprint: a frozenset of NetworkDestinations.

destinations

(IP, port) destination tuples in fingerprint

Note: Only as getter, cannot be set

Type list

certificates

Certificates in fingerprint

Note: Only as getter, cannot be set

Type list

n_flows

Threshold for the window size in seconds

Type int

static Fingerprint.**__new__**(*cls*, **args*)

FlowPrint fingerprint: a frozenset of NetworkDestinations.

3.4.1 Fingerprint comparison

To compare fingerprints using the Jaccard distance as given in the paper we provide the *fingerprint.Fingerprint.compare()* method.

Fingerprint.**compare**(*other*)

Compare fingerprint with other fingerprint

Parameters **other** (Fingerprint) – Fingerprint to compare with

Returns **result** – Jaccard similarity between self and other

Return type float

3.4.2 Fingerprint merging

To merge multiple fingerprints together we provide the *fingerprint.Fingerprint.merge()* method

Fingerprint.**merge**(**other*)

Merge fingerprint with other fingerprint(s)

Parameters ***other** (Fingerprint) – One or more fingerprints to merge with given Fingerprint

Returns **result** – Merged fingerprint

Return type *Fingerprint*

3.4.3 I/O methods

Fingerprints themselves are unchangable, however we can modify them by casting them to and from dictionaries using the following methods.

`Fingerprint.to_dict()`

Return fingerprint as dictionary object

Returns **result** – Fingerprint as dictionary, may be used for JSON export

Return type dict

`Fingerprint.from_dict(dictionary)`

Load fingerprint from dictionary object

Parameters **dictionary** (*dict*) –

Dictionary containing fingerprint object 'certificates' -> list of certificates 'destinations' -> list of destinations 'n_flows' -> int specifying #flows in fingerprint.

Returns **result** – Fingerprint object as read from dictionary

Return type *Fingerprint*

3.5 Flow

The Flow class is FlowPrint's representation of each individual Flow in the network traffic. A Flow object represents a TCP/UDP flow and all corresponding features that are used by FlowPrint to generate fingerprints. We use the *FlowGenerator* class for generating Flow objects from all packets extracted by *Reader*.

`class flows.Flow`

Flow object extracted from pcap file that can be used for fingerprinting

src

Source IP

Type string

sport

Source port

Type int

dst

Destination IP

Type string

dport

Destination port

Type int

source

(Source IP, source port) tuple

Type tuple

destination
(Destination IP, destination port) tuple

Type tuple

time_start
Timestamp of first packet in flow

Type int

time_end
Timestamp of last packet in flow

Type int

certificate
Certificate of flow, if any

Type Object

lengths
List of packet length for each packet in flow

Type list

timestamps
List of timestamps corresponding to each packet in flow

Type list

`Flow.__init__()`
Initialise an empty Flow.

3.5.1 Add packets

Once created, a Flow is still empty and needs to be populated by packets. We can add packets to a flow using the *flows.Flow.add()* method.

`Flow.add(packet)`
Add a new packet to the flow.

Parameters **packet** (*np.array of shape=(n_features,)*) – Packet from Reader.

Returns **self** – Returns self

Return type self

3.6 FlowGenerator

The FlowGenerator class generates Flow objects from packets extracted by *Reader*. To convert features from individual packets to Flows, we use the `flows.Flows.combine()` method.

class `flow_generator.FlowGenerator`
Generator for Flows from packets extraced using reader.Reader.read()

combine(*packets*)
Combine individual packets into a flow representation

Parameters **packets** (*np.array of shape=(n_samples_packets, n_features_packets)*) – Output from Reader.read

Returns **flows** – Dictionary of flow_key -> Flow()

Return type dict

3.7 FlowPrint

The FlowPrint object that is used to generate *Fingerprint*'s. Note that this is mainly a wrapper method, the actual Fingerprint generation is done in the *FingerprintGenerator*.

class flowprint.**FlowPrint**(*batch=300, window=30, correlation=0.1, similarity=0.9, threshold=0.1*)

FlowPrint object for creating fingerprints from mobile network traffic

batch
Threshold for the batch size in seconds

Type float

window
Threshold for the window size in seconds

Type float

correlation
Threshold for the minimum required correlation

Type float

similarity
Threshold for the minimum required similarity

Type float

threshold
Threshold for anomaly detection

Type float

fingerprinter
FingerprintGenerator used for generating fingerprints

Type *fingerprints.FingerprintGenerator*

fingerprints
Dictionary of Fingerprint -> label, containing all fingerprints generated by FlowPrint

Type dict

FlowPrint.**__init__**(*batch=300, window=30, correlation=0.1, similarity=0.9, threshold=0.1*)

FlowPrint object for creating fingerprints from mobile network traffic

Parameters

- **batch** (*float, default=300*) – Threshold for the batch size in seconds
- **window** (*float, default=30*) – Threshold for the window size in seconds
- **correlation** (*float, default=0.1*) – Threshold for the minimum required correlation
- **similarity** (*float, default=0.9*) – Threshold for the minimum required similarity
- **threshold** (*float, default=0.1*) – Threshold for anomaly detection

3.7.1 Generating fingerprints

`FlowPrint.fit(X, y=None)`
Fit FlowPrint object with fingerprints from given flows.

Parameters

- **X** (*np.array of shape=(n_samples,)*) – Flows for fitting FlowPrint.
- **y** (*np.array of shape=(n_samples,), optional*) – If given, attach labels to fingerprints from X.

Returns **self** – Returns FlowPrint object

Return type self

`FlowPrint.predict(X, y=None, default='common')`
Find closest fingerprint to trained fingerprints

Parameters

- **X** (*Array-like of Fingerprint of shape=(n_fingerprints,)*) – Fingerprints to compare against training set.
- **y** (*ignored*) –
- **default** ('common'|'largest'|other, default='common') –

  Default to this strategy if no match is found

  – 'common' : return the fingerprint with most flows

  – 'largest': return the largest fingerprint

  – other: return <other> as match, e.g. Fingerprint()/None

Returns **result** – Closest matching fingerprints to original. If no match is found, fall back on default

Return type np.array of shape=(n_fingerprints,)

`FlowPrint.fit_predict(X, y=None, default='common')`
Fit FlowPrint with samples and labels and return the predictions of the same samples after running them through FlowPrint.

Parameters

- **X** (*np.array of shape=(n_samples,)*) – Flows for fitting FlowPrint.
- **y** (*np.array of shape=(n_samples,), optional*) – If given, attach labels to fingerprints from X.
- **default** ('common'|'largest'|other, default='common') –

  Default to this strategy if no match is found

  – 'common' : return the fingerprint with most flows

  – 'largest': return the largest fingerprint

  – other: return <other> as match, e.g. Fingerprint()/None

Returns **result** – Closest matching fingerprints to original. If no match is found, fall back on default

Return type np.array of shape=(n_fingerprints,)

3.7.2 App Recognition

Once FlowPrint is trained using the `fit()`, you can use FlowPrint to label unknown Flows with known apps.

`FlowPrint.`**`recognize`**(*X*, *y=None*, *default='common'*)

Return labels corresponding to closest matching fingerprints

Parameters

- **X** (*Array-like of Fingerprint of shape=(n_fingerprints,)*) – Fingerprints to compare against training set.
- **y** (*ignored*) –
- **default** ('common'|'largest'|other, default='common') –

  Default to this strategy if no match is found

  – 'common' : return the fingerprint with most flows

  – 'largest': return the largest fingerprint

  – other: return <other> as match, e.g. Fingerprint()/None

Returns **result** – Label of closest matching fingerprints to original

Return type np.array of shape=(n_fingerprints,)

3.7.3 Unseen app detection

Once FlowPrint is trained using the `fit()`, you can use FlowPrint to detect if unknown Flows are in the set of known (trained) apps or if they are a previously unseen app.

`FlowPrint.`**`detect`**(*X*, *y=None*, *threshold=None*)

Predict whether samples of X are anomalous or not.

Parameters

- **X** (*np.array of shape=(n_samples,)*) – Flows for fitting FlowPrint.
- **y** (*Ignored*) –
- **threshold** (*float, default=None*) – Minimum required threshold to consider point benign. If None is given, use FlowPrint default

Returns **result** – Prediction of samples in X: +1 if benign, -1 if anomalous.

Return type np.array of shape=(n_samples,)

3.7.4 I/O methods

FlowPrint provides methods to save and load a FlowPrint object, including its fingerprints to a json file.

`FlowPrint.`**`save`**(*file*, *fingerprints=None*)

Save fingerprints to file.

Parameters

- **file** (*string*) – File in which to save flowprint fingerprints.
- **fingerprints** (*iterable of Fingerprint (optional)*) – If None export fingerprints from fitted FlowPrint object, otherwise, export given fingerprints.

`FlowPrint.`**`load`**(**files*, *store=True*, *parameters=False*)

Load fingerprints from files.

Parameters

- **`file`** (*`string`*) – Files from which to load fingerprints.
- **`store`** (*`boolean, default=True`*) – If True, store fingerprints in FlowPrint object
- **`parameters`** (*`boolean, default=False`*) – If True, also update FlowPrint parameters from file

Returns **result** – Fingerprints imported from file.

Return type dict of Fingerprint -> label

3.8 FingerprintGenerator

This generator performs all steps to transform *Flow*'s into *Fingerprint*'s. These steps include

1) Batch data
2) Clustering (also see *Cluster*)
3) Cross correlation (also see *CrossCorrelationGraph*)
4) Finding cliques (also see *CrossCorrelationGraph*)
5) Transforming cliques into Fingerprints. (also see *Fingerprint*)

`class` `fingerprints.`**`FingerprintGenerator`**(*batch=300*, *window=30*, *correlation=0.1*, *similarity=0.9*)

Generator of FlowPrint Fingerprint objects from flows

`batch`

Threshold for the batch size in seconds

Type float

`window`

Threshold for the window size in seconds

Type float

`correlation`

Threshold for the minimum required correlation

Type float

`similarity`

Threshold for the minimum required similarity

Type float

`FingerprintGenerator.`**`__init__`**(*batch=300*, *window=30*, *correlation=0.1*, *similarity=0.9*)

Generate FlowPrint Fingerprint objects from flows

Parameters

- **`batch`** (*`float, default=300`*) – Threshold for the batch size in seconds
- **`window`** (*`float, default=30`*) – Threshold for the window size in seconds
- **`correlation`** (*`float, default=0.1`*) – Threshold for the minimum required correlation

- **similarity** (*float, default=0.9*) – Threshold for the minimum required similarity

3.8.1 Fingerprint generation

The method `fingerprints.FingerprintGenerator.fit_predict()` performs all steps required for fingerprint generation.

`FingerprintGenerator.`**`fit_predict`**(*X*, *y=None*)
Create fingerprints from given samples in X.

Parameters

- **X** (*array-like of shape=(n_samples,)*) – Samples (Flow objects) from which to generate fingerprints.
- **y** (*array-like of shape=(n_samples,), optional*) – Labels corresponding to X. If given, they will be encorporated into each fingerprint.

Returns **result** – Resulting fingerprints.

Return type np.array of shape=(n_samples,)

3.9 NetworkDestination

A NetworkDestination represents a cluster of flows that communicate with the same destination.

class `network_destination.`**`NetworkDestination`**(*identifier*, *samples=[]*)
NetworkDestination object for flow samples

identifier
Unique identifier for NetworkDestination

Type object

samples
List of flows stored in NetworkDestination

Type list

destinations
Set of destination (IP, port) tuples related to NetworkDestination

Type set

certificates
Set of TLS certificates related to NetworkDestination

Type set

labels
Labels related to NetworkDestination

Type Counter

`NetworkDestination.`**`__init__`**(*identifier*, *samples=[]*)
NetworkDestination object for flow samples

Parameters

- **identifier** (*object*) – Identifier for NetworkDestination Important: identifier must be unique!
- **samples** (*iterable of Flow*) – Samples to store in this NetworkDestination.

3.9.1 Adding Flows

We add new Flows using the `network_destination.NetworkDestination.add()` method.

`NetworkDestination.`**`add`**(*X*, *y=None*)
Add flow X to NetworkDestination object.

Parameters

- **X** (`Flow`) – Datapoint to store in this NetworkDestination.
- **y** (*object*) – Label for datapoint

3.9.2 Merging destinations

When merging two network destinations, we use the `network_destination.NetworkDestination.merge()` method.

`NetworkDestination.`**`merge`**(*other*)
Merge NetworkDestination with other NetworkDestination object.

Parameters **other** (`NetworkDestination`) – Other NetworkDestination object to merge with.

3.10 Preprocessor

The Preprocessor object transforms data to from .pcap files to *Flow*.

class `preprocessor.`**`Preprocessor`**(*verbose=False*)
Preprocessor object for preprocessing flows from pcap files

reader
pcap Reader object for reading .pcap files

Type *reader.Reader*

flow_generator
Flow generator object for generating Flow objects

Type flows.FlowGenerator

`Preprocessor.`**`__init__`**(*verbose=False*)
Preprocessor object for preprocessing flows from pcap files

3.10.1 Process data

The process method extracts all flows and labels (currently the file name) from a given input .pcap file.

`Preprocessor.`**`process`**(*files*, *labels*)
Extract data from files and attach given labels.

Parameters

- **files** (*iterable of string*) – Paths from which to extract data.

- **labels** (*iterable of int*) – Label corresponding to each path.

Returns

- **X** (*np.array of shape=(n_samples, n_features)*) – Features extracted from files.
- **y** (*np.array of shape=(n_samples,)*) – Labels for each flow extracted from files.

3.10.2 I/O methods

As this process can take a long time, especially when using the pyshark backend (see *Reader*), the Preprocessor offers methods to save and load data through the means of pickling.

Preprocessor.**save**(*outfile*, *X*, *y*)
Save data to given outfile.

Parameters

- **outfile** (*string*) – Path of file to save data to.
- **X** (*np.array of shape=(n_samples, n_features)*) – Features extracted from files.
- **y** (*np.array of shape=(n_samples,)*) – Labels for each flow extracted from files.

Preprocessor.**load**(*infile*)
Load data from given infile.

Parameters **infile** (*string*) – Path of file from which to load data.

Returns

- **X** (*np.array of shape=(n_samples, n_features)*) – Features extracted from files.
- **y** (*np.array of shape=(n_samples,)*) – Labels for each flow extracted from files.

3.11 Reader

The Reader object extracts raw features from .pcap files that can be turned into *Flow* using the *Preprocessor* class.

class reader.**Reader**(*verbose=False*)
Reader object for extracting features from .pcap files

verbose
Boolean indicating whether to be verbose in reading

Type boolean

Reader.**__init__**(*verbose=False*)
Reader object for extracting features from .pcap files

Parameters **verbose** (*boolean, default=False*) – Boolean indicating whether to be verbose in reading

3.11.1 Read data

Reader provides the read() method which reads flow features from a .pcap file. This method automatically chooses the optimal available backend to use.

`Reader.read(path)`
Read TCP and UDP packets from .pcap file given by path. Automatically choses fastest available backend to use.

Parameters **path** (*string*) – Path to .pcap file to read.

Returns

result – Where features consist of:

0) Filename of capture
1) Protocol TCP/UDP
2) TCP/UDP stream identifier
3) Timestamp of packet
4) Length of packet
5) IP packet source
6) IP packet destination
7) TCP/UDP packet source port
8) TCP/UDP packet destination port
9) SSL/TLS certificate if exists, else None

Return type np.array of shape=(n_packets, n_features)

> **Warning:**
>
> **warning** Method throws warning if tshark is not available.

3.11.2 Cutsom Backend

Alternatively, you can choose your own backend using one of the following methods.

`Reader.read_tshark(path)`
Read TCP and UDP packets from file given by path using tshark backend

Parameters **path** (*string*) – Path to .pcap file to read.

Returns

result – Where features consist of:

0) Filename of capture
1) Protocol TCP/UDP
2) TCP/UDP stream identifier
3) Timestamp of packet
4) Length of packet
5) IP packet source
6) IP packet destination
7) TCP/UDP packet source port
8) TCP/UDP packet destination port

9) SSL/TLS certificate if exists, else None

Return type np.array of shape=(n_packets, n_features)

`Reader.`**`read_pyshark`**(*path*)

Read TCP and UDP packets from file given by path using pyshark backend

Parameters **path** (*string*) – Path to .pcap file to read.

Returns

result – Where features consist of:

0) Filename of capture
1) Protocol TCP/UDP
2) TCP/UDP stream identifier
3) Timestamp of packet
4) Length of packet
5) IP packet source
6) IP packet destination
7) TCP/UDP packet source port
8) TCP/UDP packet destination port
9) SSL/TLS certificate if exists, else None

Return type np.array of shape=(n_packets, n_features)

CHAPTER 4

Roadmap

This part of the documentation keeps track of desired features in future releases.

- None at the moment

4.1 Nice to haves

Features that are listed here would be nice to have for FlowPrint. I probably won't implement them myself, but feel free to send me a pull request.

- Read from a live capture
- Visualisation module that plots the Clusters, CrossCorrelationGraph and Fingerprints live while running.

CHAPTER 5

License

CHAPTER 6

Citing

To cite FlowPrint please use the following publication:

van Ede, T., Bortolameotti, R., Continella, A., Ren, J., Dubois, D. J., Lindorfer, M., Choffnes, D., van Steen, M. & Peter, A. (2020, February). FlowPrint: Semi-Supervised Mobile-App Fingerprinting on Encrypted Network Traffic. In 2020 NDSS. The Internet Society.

[PDF]

6.1 Bibtex

```
@inproceedings{vanede2020flowprint,
  title={{FlowPrint: Semi-Supervised Mobile-App Fingerprinting on Encrypted Network Traffic}},
  author={van Ede, Thijs and Bortolameotti, Riccardo and Continella, Andrea and Ren, Jingjing and Dubois, Daniel J. and Lindorfer, Martina and Choffness, David and van Steen, Maarten, and Peter, Andreas},
  booktitle={NDSS},
  year={2020},
  organization={The Internet Society}
}
```

Index

Symbols

A

B

C

D

F

G

I

L

M

N

P

R

S

T

V

W