University of Padua

Department of Mathematics

Bachelor's Degree in Computer Science

BFF: a modular framework for botnet detectors evaluation and a practical application

Author

Nicola De Cao

Supervisors

Mauro Conti

Gilberto Filè

Academic Year 2015-2016

Contents

List of Figures

List of Tables

Chapter 1

Overview

In this work we present Botnet Finder Framework (BFF) a modular framework for botnet detectors evaluation and a practical application of the framework for research purpose.

A botnet is a network of machines that are infected by a malware controlled by an attacker in order to perform illegitimate actions. Once a machine becomes infected it joins the malware network and the owner of the botnet (called botmaster) can perform several malicious or illegal actions remotely controlling the bot.

In order to prevent botnet spreading or at least to block malicious traffic, detection algorithm has been developed. Detectors should distinguish between benign and malicious traffic accessing to network data. Most of state of the art detectors use statistical approaches to extract relevant features from data to identify malicious behaviors. Thereafter, they can detect malicious communications between botmasters and infected machines analyzing those features. Security research in this field is still on looking for developing more precise algorithms.

As detectors are improving their performances even new botnets are be-

coming more elusive. New decentralized approaches are quite difficult to detect. Besides some botnets use covert channels to communicate in order to hide malicious traffic into benign one and become undetectable so.

A new botnet which claim to be undetectable by modern detectors is ELISA [1]. ELISA is a botnet that propagates itself using social network (i.e. Facebook). In particular it uses social media accounts available in infected machines. ELISA hides malicious traffic into social traffic generated by victims. For instance when an user interacts with his social network sending messages, ELISA appends invisible contents to these messages. These contents are used by the botnet to internal communications within bots and botmasters. As future work, authors planned to prove the undetectability of ELISA's covert channel.

We want to test their covert channel using a real world IP traffic dataset containing malicious packets generated by ELISA and two detector algorithms. We choose Disclosure [2] and BotTrack [3] for our evaluation because they are state of the art botnet detectors. Their implementation were unavailable so we reimplemented them on the best of our knowledge.

As we wanted to evaluate our reimplementation and to plot graphs with measurements we decided to do not develop detectors only but also a number of tools. We decided to develop reusable software and to deploy it as a modular and extensible framework. This choice was motivated by the fact that in such a way future implementation of botnet detectors would be easier using our framework. Moreover, we develop an evaluation module which can help researcher to compare several detectors and to evaluate the quality of their work. One of the main purpose of this work is to discuss this framework implementation.

After we had built our system, we tested Disclosure and BotTrack to

validate our correct reimplementations on an real dataset containing real botnets.

Thereafter we performed an experiment recording network traffic containing ELISA malicious traffic. We evaluated ELISA undetectability showing that both Disclosure and BotTrack detectors failed to detect ELISA. That failure is due to ELISA covert channel and how both detectors are built. They both classify malicious traffic whether it comes form or goes to an IP address classified as malicious. So they failed classifying social network server machines as malicious because only few communications from or to them are actually malicious. We consider this result as a success and an empirical proof of ELISA undetectability.

Chapter 2

Introduction

We now will discuss briefly about what a botnet is (Section 2.1) and how botnet detectors work (Section 2.2). Then follows a short description of ELISA (Section 2.3) and finally an introduction to our framework Botnet Finder Framework (BFF) (Section 2.4).

2.1 Botnet Command and Control (C&C)

A botnet is a network of machines that are infected by a malware controlled by an attacker in order to perform illegitimate actions.

Botnets are becoming one of the biggest security threats, responsible for a large volume of malicious activities. Once a botnet is established, the owner (called botmaster) can perform identity theft, DoS attacks, spread new dangerous malwares or steal information.

First appearance of botnets were in 1990 and now botnets are considered one of the most serious threats against cyber-security. They are difficult to detect, hard to prevent, and their dimension can be as big as millions of infected machines worldwide [4]. Botmasters use a Command and Control

(C&C) channel that consist of servers and different technical infrastructure used to control and send commands to infected machines. Botmasters use botnets to coordinate them fraudulent activities.

In the past, C&C channels were often built over the well known Internet Relay Chat (IRC) and Hypertext Transfer Protocol (HTTP) protocols, which provided a centralized mechanism. Today, centralized C&C channels are considered ineffective and the new generation of botnets adopt a decentralized communication mechanism. Peer-To-Peer (P2P) botnets [5] are the first attempt of decentralized C&C communication. They use the P2P protocol as C&C channel, which avoids the single point of failure of centralized botnets that is the C&C server.

2.2 Botnet detectors

Botnet detectors largely follow one of two major approaches: detecting C&C channels used by botmasters to communicate with infected machines or detecting patterns of infected machines behavior in response to botmaster commands.

Unavailability of raw network data is due to legal and administrative restrictions and it does not facilitate botnet detection on a large scale. Fortunately, another an alternative data source as NetFlow [6] data is widely available today. This source is extremely attractive but imposes several challenges for performing accurate botnet detection.

In particular NetFlow records do not include packet payloads, in fact NetFlow data contains flow records metadata aggregation such the flow duration and number of bytes transferred. Besides NetFlow data is often collected by sampling a monitored network removing some traffic.

Detectors try to distinguish between benign and malicious network traffic accessing to NetFlow data and trying to extract relevant features in order to detect malicious behavior.

Detection of C&C channel is one of the most important aspect of botnets detection. Understanding and analyzing botnet behavior is a preliminary step for identification of C&C [7]. There are many approaches detectors can use, most of them use statistical approaches to detect C&C communication. Machine learning is used for network traffic classification, in particular supervised learning. It involves identification of features and using these to train a classifier on a relevant dataset.

The first machine learning algorithms have been used to classify network traffic into IRC and non-IRC traffic and then to identify botnet and non-botnet traffic within the IRC traffic [8]. Then, modern approaches have improved methods and gained accuracy and precision [9], [10] [2].

Graph-based models are other approaches and they are used to represent malicious connections. Machine learning techniques are then used to generate graph for C&C activity and trained to classify set of graphs that are associated with malicious C&C [3].

As explained in the overview of this work (see Section 1), in order to evaluate ELISA detectability there is the need to test it in a controlled environment while observed by botnet detectors. We choose Disclosure [2] and BotTrack [3] detectors for our evaluation because they are state of the art botnet detectors. We discuss how we use them in the Section 3.6.

2.3 Boten ELISA

Elusive Social Army (ELISA) [1] is a Online Social Networks (OSN) based botnet that propagates itself and the C&C messages using social accounts available in infected machines.

In ELISA, differently from C&C server-based botnets there are no components specifically built to manage the C&C channel. The social relationships between victims form an overlay network over the OSN which connects the bots (infected machines) and the botmaster together. Hence, botmaster and bots communicate by exchanging command and control messages on the overlay network. Command and control messages are piggybacked on the benign content the victim posts on the OSN, therefore the usual victims interactions on the OSN deliver the messages to the whole botnet.

In particular ELISA uses Unicode steganography to build a covert channel, by injecting non-printable characters into the user-generated content that he post on OSNs. This technique exploits non-printable characters and characters with identical visual representation in order to hide information within text messages which remains human-readable. So that prevents ELISA from generating detectable network traffic, and guarantees ELISA to be unnoticed by OSN users.

For instance when an user, on an infected machine, publishes benign content on a OSN (i.e. Facebook), ELISA transform the user input appending some malicious information. In particular ELISA encode a message into non-printable characters and append it to the original one since user will be unaware of that. Once the encrypted message spreads on the OSN, other bots or the botmaster itself can decode the message and perform actions.

ELISA was implemented as a prototype, and authors ran a thorough set of experiments that confirm the feasibility of ELISA and the effectiveness of

its communication channel.

2.4 Botnet Finder Framework

We decided to design a framework for easily develop, test and evaluate performance of detector algorithms in order to easily reimplement two detectors and give a contribution to the field.

Thus, we developed Botnet Finder Framework (BFF): a modular and extensible framework which provides features to facilitate developers work. In particular, it exposes many abstract and concrete base classes to handle with datasets, features extractors, detector algorithms, metrics computations and results plotting. We divide BFF into completely independent modules in order handle complexity, simplify user interactions and facilitate future development work. We discuss of BFF implementation in the Chapter 3.

Chapter 3

Framework

In this chapter we describe requirements, testing systems, architectural and detailed design of BFF and of the two implemented algorithms. We briefly explain what BFF and algorithms should do and how in the requirements Section (3.1). Then, we present a short overview about technologies and tools chosen to develop them (Section 3.2). Next, we describe BFF and algorithms architectural (Section 3.3) and detailed design (Section 3.4). Thereafter we explain what design patterns we used and why, followed by the description of how we implement Disclosure and BotTrack algorithms (Section 3.6). At the end of the chapter we present how we verify, test and also validate on real traffic BFF the two implemented algorithms.

3.1 Requirements

We identify four requirement classes: functional requirements which are what the system must do, performance requirements which are how the system must be efficient, design requirements which are bounds about how the system must be designed and quality requirements which are how the system

must be tested before releases.

Functional requirements. The framework must provide modules, function skeletons, interfaces and methods to easily implement, develop and evaluate different botnet detectors. In particular, it must have interfaces and abstract base classes in order to help developers to interact easily with the framework while developing their feature extractor modules or detector algorithms. It must supply some build-in components to compute evaluation metrics and plot results but it must be possible and easy to extend those component adding functionalities. So it must have components to:

- build data structures from NetFlow files;
- use those structures to extract features;
- use those features to learn from a training set and later predict target values on a test set;
- use predictions to compute evaluation metrics;
- use evaluation metrics to plot results;
- save and load from files partial results before or after each atomic step described above.

Performance requirements. Due to the huge dataset users can work with either BFF and the algorithms must have performance requirements. In particular BFF requires memory to load dataset files and build large data structures with them but it does not need computational power to process data. On the contrary algorithms seldom have particular memory requirements, because they use data structure provided by BFF, but they often

require much more CPU power to execute, for example during features extraction.

Due to the wide diversity of the algorithms that could be implemented in the future we identify performance requirements for BFF and the two implemented algorithms only (see Section 3.6.1 and 3.6.2).

Feature extraction computing of both implemented algorithms needs intense CPU work. In order to prevent long waitings and maximize the use of modern multi-cores CPU power, we impose that during feature extractions each independent feature must be computed concurrently. Besides, BFF does not require particular CPU time requirements as a matter of fact it only deals with data structure creation and management and execute basic operations but it does not compute procedures with high computational complexity.

BFF uses a lot of memory to build and store data structures but we do not impose fixed strict bounds on memory usage. This choice is motivated by the fact that the purpose of this work is not to build a real time system which detects botnets but it has to statically analyze detectors performance so we do not give particular attention on memory optimization.

Design requirements. One of the main purposes of BFF is to be modular so it is fundamental that BFF has strict design requirements. BFF must be spitted into separate modules which can be called separately, if needed. BFF must be designed in order to minimize the changes needed if someone what to add or modify modules or components. Moreover BFF must maximize dynamic parameters loading in order to maximize customization for future configurations and user cases.

Furthermore, we impose to develop BFF using Python [11] programming language due to the fact that it is wide used in scientific computing and

machine learning applications.

Quality requirements. In order to test whether algorithms perform as expected we imposed that development activity must be test driven. Test driven development is develop test first and than develop source code periodically testing whether outputs are as expected. In particular we do it so for feature extractions components. At this time we do not give particular attention on testing components interactions and interfaces but we consider it as a must for future work (see Section 5).

3.2 Technologies and tools

We now present a brief description of technologies and tools used to build BFF and the implemented algorithms. We explain how we use those technologies and tools and why we choose them.

Wireshark. Wireshark [12] is a free and open source network packet analyzer. It capture network packets and display high detailed data of them. The following are some of the many features Wireshark provides:

- capture live packet data from a network interface;
- open files containing packet data captured with tcpdump/WinDump, Wireshark, and a number of other packet capture programs;
- export some or all packets in a number of capture file formats;
- save packet data captured.

We used tools provided by Wireshark as the exporter to convert `.pcap` files provided by The Information Security Centre of Excellence (ISCX)

(a research center within Faculty of Computer Science, University of New Brunswick, Canada) [13] in `.csv` files with NetFLow data only as described in Section 3.3.1.

Python. BFF source code is in Python due to the fact that it is wide used in scientific computing adding the reasons explained below.

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. It offers high-level built-in data structures, dynamic typing and dynamic binding. Those features make it very attractive for Rapid Application Development, or for use it as a scripting or glue language to connect existing components together. Python's simple syntax maximize readability and reduces the cost of program maintenance. Python supports modules and packages, which are useful features in order to perform program modularity and code reuse.

NumPy. Disclosure algorithm intensively use very large arrays so due to high-level language Python is arrays are not managed with maximum efficiency. Therefore it uses NumPy [14] library to do it. NumPy is a package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, basic linear algebra and basic statistical operations.

SciPy. Both implemented algorithms an BFF itself need to compute complex mathematical operations so a support library such SciPy [15] is used. SciPy is a collection of mathematical algorithms and convenience functions built on the NumPy extension of Python. It adds significant power to the

interactive Python session by providing the user with high-level commands and classes for manipulating and visualizing data.

Matplotlib. BFF has a module (see Section 3.3.5) which plot results graphically. To do it so it use an external plotting library such Matplotlib [16]. It is a Python library which produces publication quality figures in a variety of hard-copy formats.

SciKit-learn. Both implemented algorithm (see Section 3.6.1 and 3.6.2) use machine learning on some features in order to find patterns over malicious traffic. In order to perform such algorithm SciKit-learn [17] library is used. It provide many tools and various classification, regression and clustering algorithms such random forests, support vector machines, DBSCAN, and k-means. It also provides functions to perform stratified k fold cross-validation and compute evaluation metrics on predictions. SciKit-learn is built on top of SciPy and Numpy.

NetworkX. BotTrack algorithm (see Section 3.6.2) is based on network graph analysis. NetworkX library is used in order to perform such analysis and features extraction after graph generation. In fact NetworkX [18] is a Python library for the creation, manipulation, and study of the structure and dynamics of networks.

3.3 Architectural design

In this section we present the architectural design of BFF and how we organize modules and packages. Due to modularity of BFF, we divide the framework in independent modules. Each module needs a different input

and generates a different output. Each module receives some input from another module and compute output for the next one. Figure 3.1 shows how BFF transforms data from NetFlow to metrics and plots.

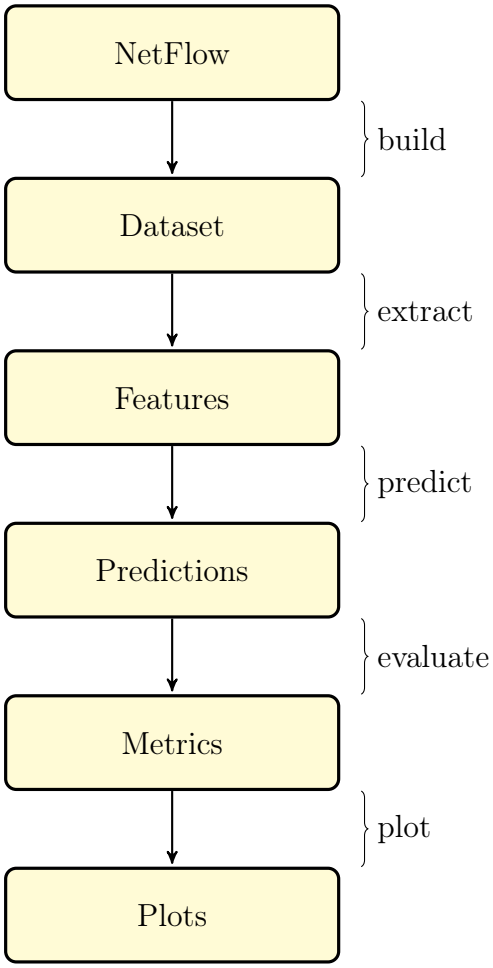

Figure 3.1: Data flow of BFF

Initially build module (see Section 3.3.1) receive NetFlow as input, then it provides to build internal data representation of the data. Then the extract module performs features extraction from datasets (see Section 3.3.2). Follows the predict and evaluate modules which make predictions from features and compute evaluation metrics respectively (see sections 3.3.3 and 3.3.4).

Finally the plot module use processed metrics to plot visual representations of them (see Section 3.3.5).

BFF always save output data into files for later processing as input of the following module except for the build module. In order to provide human readability we use JSON and CSV files. Now we describe each module and its input and output.

3.3.1 Build module

The build module provides to load records from a NetFlow file and build internal data representation of the data which are later used to extract features from. BFF use NetFlow as initial input for the traffic analysis.

Dataset builders extract information required to build a dataset from NetFlow files. In particular they build a data structure which is later used by the extract module. In order to maximize separation of concern and handle the creation of dataset we use a builder pattern (see Section 3.5). Moreover we use an exhaustive classes and interfaces hierarchy in order to easily allow future extensions. See the detailed design for a more exhaustive explanation of those designs (Section 3.4).

NetFlow. NetFlow is a network protocol developed by Cisco for collecting IP traffic information and monitoring network traffic. Devices or softwares that support or implement NetFlow can collect IP traffic data and then export those data as NetFlow records. A NetFlow record contains information about the traffic in a given flow. For instance it contains input and output interface indexes, date and durations of flows, number of bytes and packets, source and destination IP addresses and ports and used protocol [6].

NetFlow data are expected to be in stored into CSV files which every row

is a record and every column information about it.

Datasets. Datasets are outputs of the build module. They consist in an internal representation of NetFlow. In particular they are very large arrays where row are NetFlow records and columns are information as described above. Besides, both of the two implemented algorithms need additional views of information, for example what IPs are labeled as botnet. We create special dataset classes in the class hierarchy to do it so and we collocate them into a proper package `datasets` (see Section 3.4 for further information).

We do not need to save dataset into files since it does not produce any kind of advantage so there are not methods to do it so at the moment.

3.3.2 Extract module

Extract module is the component which extracts features form datasets provided by the build module. Each extractor implements the same interface but each of them extracts different features. We implement extractors using strategy design pattern. Besides in order to easily call an extractor from command line we create a class `Extractor` which bind a string parameter given as input to a particular extractor. In this case `Extractor` is a proxy as described in 3.5 for each extractor. See the detailed design for a more exhaustive explanation of this design (Section 3.4).

Features. After a features extraction algorithm is used to extract features from a dataset, it generates, for each dataset sample, a number of futures. Features are stored into JSON files with samples as keys (often IPs) and features as values. Values are dictionaries with feature name as keys and the feature value as values.

3.3.3 Predict module

Since features extraction is often very time consuming, BFF predict module use features previously generated to preform predictions. Predictions are made by each detectors looking features of a samples without know *a priori* whether it is botnet or not and trying to predict it using trained classifiers. Predictions are computed by different algorithm and as for extractors we use a proxy (`Algorithm`) to bind command line parameter and call the proper algorithm.

Predictions. Predictions are eventually saved in a JSON file. Predictions are saved as vectors of integer where a zero correspond to predict a samples as benign traffic and one as malicious one. Into files BFF saves either ground-truth and predictions.

3.3.4 Evaluate module

In order to evaluate how well detector algorithms work, BFF gives as output some metrics which are commonly used to measure binary classifier performance [19]. We choose to compute metrics like accuracy, precision, sensitivity and ROC AUC scores. Due to modularity of BFF, we create a particular package (`metrics`) with the purpose of containing current implemented metrics but also additional future contributions and new metrics. Now we present a brief description of metrics file and of chosen metrics.

Metrics. Metrics are saved in a JSON file. It contains, for each selected botnet class, a dictionary with name of metrics as keys and their computed values as values logically.

Accuracy score. Accuracy is the proportion of true results (both True Positives $TP = number\ of\ true\ positives$ and True Negatives $TN = number\ of\text{-}true\ negatives$) among the total number of correct or incorrect classifications. It can also be seen as the probability that a classification is made correctly.

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Precision score. Precision or Positive Predictive Values (PPV) is the proportion of the samples that the classifier predict positive and are actually positive (True Positive, $TP = number\ of\ true\ positives$) among all the samples that it classifies as positive (False Positive, $FP = number\ of\ false\ positives$). It can also be seen as the probability that the test is malicious traffic given that it is classified as positive. With higher precision, fewer benign traffic is classified as malicious.

$$precision = \frac{TP}{TP + FP}$$

Sensitivity score. Sensitivity or True Positive Rate (TPR), also known as recall, is the proportion of the samples that the classifier predict positive and are actually positive (True Positive, $TP = number\ of\ true\ positives$) among all the samples that are positive (False Negative, $FN = number\ of\ false\ negatives$, Condition Positive, $CP = TP + FN$). It can also be seen as the probability that the test is classified as positive given that the sample is labeled as actually malicious traffic. With higher sensitivity, fewer actual malicious traffic go undetected.

$$sensitivity = \frac{TP}{TP + FN}$$

ROC AUC score. Receiver Operating Characteristic (ROC) curve, is a graphical representation that shows the performance of a binary classifier.

The curve is created by plotting the TPR and the False Positive Rate (FPR) at various threshold settings.

When using normalized units (so from 0 to 1), the Area Under the Curve (AUC) is the probability that the classifier ranks a randomly chosen positive instance higher than a randomly chosen negative one [20].

$$A = \int_{\infty}^{-\infty} \mathrm{TPR}(T)\mathrm{FPR}'(T)\,dT = P(X_1 > X_0)$$

where X_1 is the score for a positive instance and X_0 is the score for a negative one.

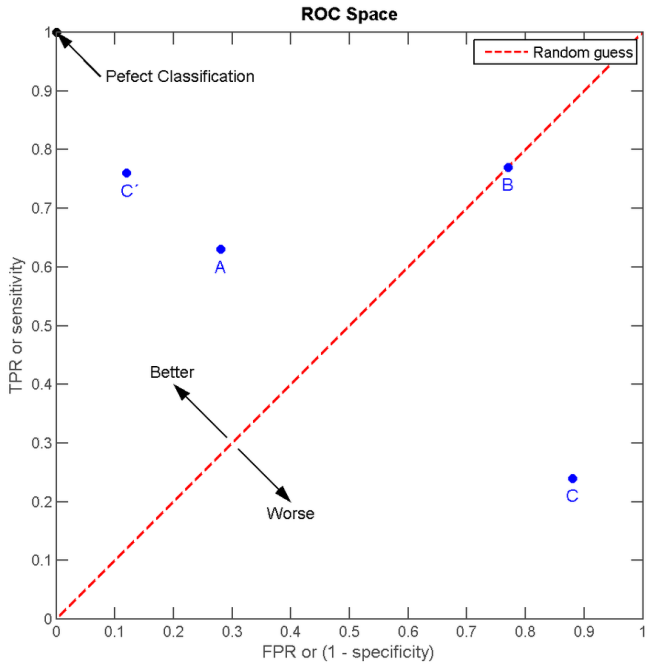

Figure 3.2: ROC AUC score examples: C is the worst classifier since it performs worse than the random guess; B performs as well as the random guess; A and C perform well; the perfect classifier is at the top left corner.

3.3.5 Plot module

After metrics computation often users desire a visual representation of them. Plot module provide to load a metrics scores file and performs some plots. Plots have several parameter and because of that BFF loads a configuration file before in order to load all those parameters.

Plots. Plots are saved in EPS vectorial format in order be easily exported by users in other formats without a quality loss. Implemented plots are two kind of bar chart: one which shows total performance of a detector on various metrics and another which shows evaluations metrics for each kind of botnet class detected.

3.4 Detailed design

3.4.1 algorithms package

This package provides classes to implement detector algorithms

algorithms::AlgorithmInterface

Description

This is the interface for each detection algorithm

Children classes

- algorithms::Algorithm
- algorithms::AbstractAlgorithm

Methods

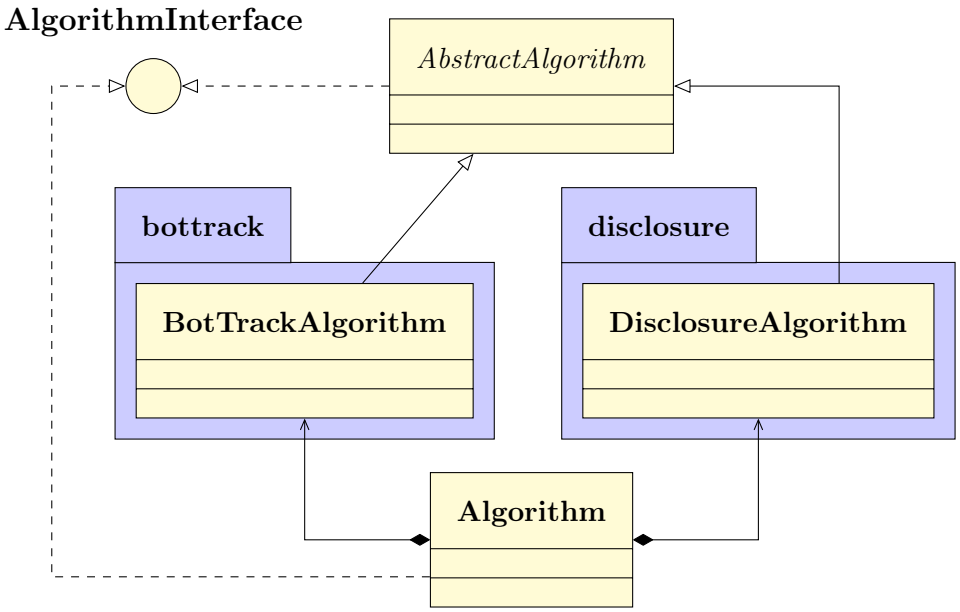

Figure 3.3: Package diagram: algorithms

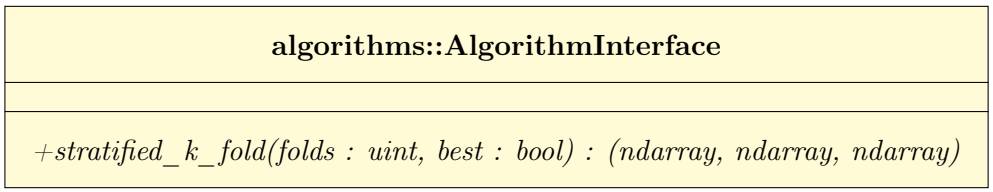

Figure 3.4: Class diagram - algorithms::AlgorithmInterface

- stratified_k_fold(folds : uint, best : bool) : (ndarray, ndarray, ndarray) → This method computes a stratified k-fold cross validation of the samples
  - folds → How many k-folds to perform
  - best → If use the best classifier for the given samples

algorithms::Algorithm

Description

This class binds to many detector algorithms depending on parameters

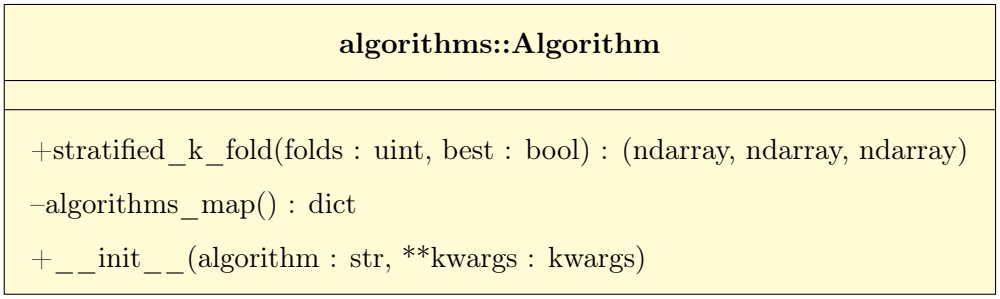

Figure 3.5: Class diagram - algorithms::Algorithm

Parent classes

- algorithms::AlgorithmInterface

Methods

- stratified_k_fold(folds : uint, best : bool) : (ndarray, ndarray, ndarray) → This method computes a stratified k-fold cross validation of the samples
  - folds → How many k-folds to perform
  - best → If use the best classifier for the given samples
- algorithms_map() : dict → This method provides a map to bind a string to an algorithm
- __init__(algorithm : str, **kwargs : kwargs) → This method is the constructor of the class
  - algorithm → Which algorithm to build
  - **kwargs → Other keyword arguments

algorithms::AbstractAlgorithm

Description

This class is an abstract base class for each detection algorithm

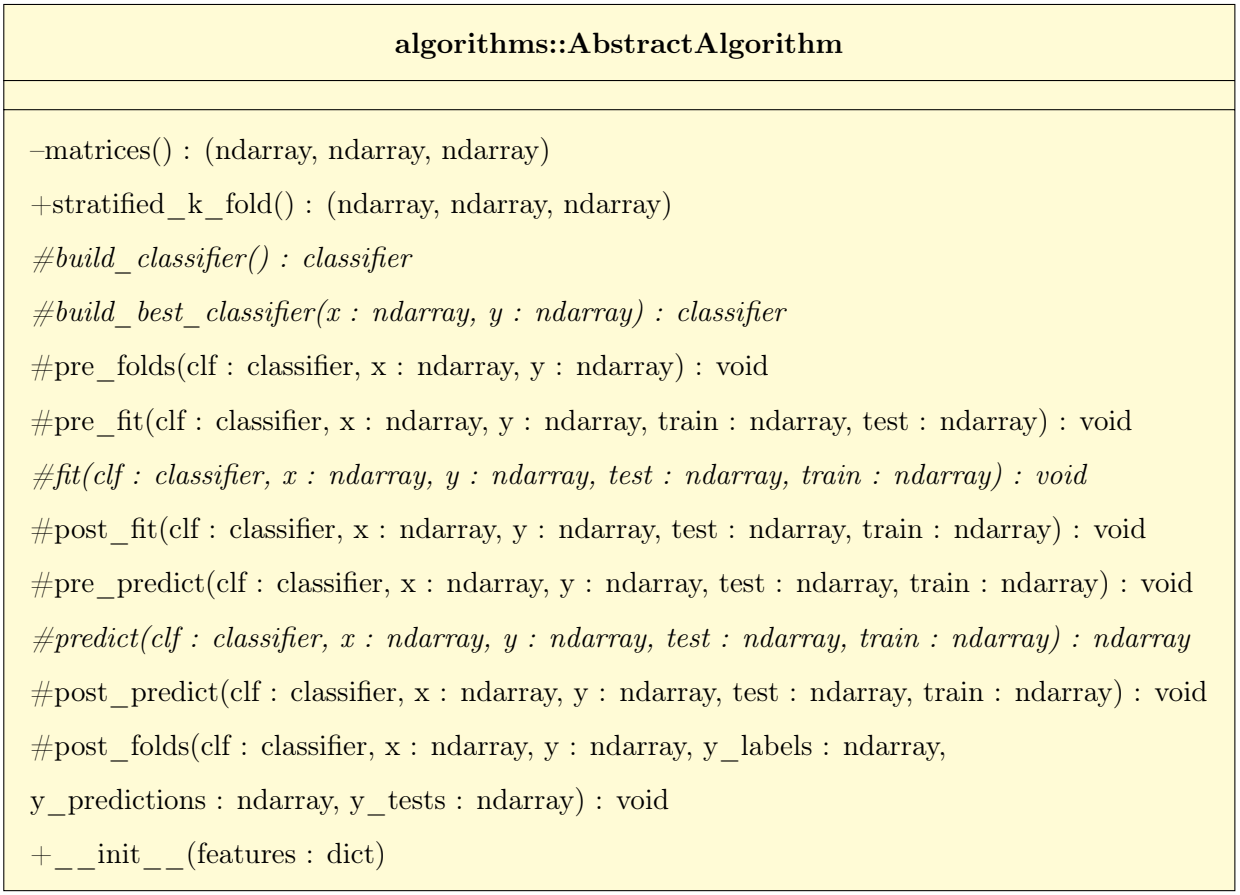

Figure 3.6: Class diagram - algorithms::AbstractAlgorithm

Parent classes

- algorithms::AlgorithmInterface

Children classes

- algorithms::bottrack::BotTrackAlgorithm
- algorithms::disclosure::DisclosureAlgorithm

Methods

- matrices() : (ndarray, ndarray, ndarray) → This method gets sample features matrix, labels vector and named labels vector

- stratified_k_fold() : (ndarray, ndarray, ndarray) → This method computes a stratified k-fold cross validation of the samples
- build_classifier() : classifier → This method builds a classifier
- build_best_classifier(x : ndarray, y : ndarray) : classifier → This method builds the best classifier based on features
  - x → Features matrix
  - y → Labels matrix
- pre_folds(clf : classifier, x : ndarray, y : ndarray) : void → This method executes operations before the k-fold computation
  - clf → The classifier to use
  - x → Features matrix
  - y → Labels matrix
- pre_fit(clf : classifier, x : ndarray, y : ndarray, train : ndarray, test : ndarray) : void → This method executes operations before the fit computation
  - clf → The classifier to use
  - x → Features matrix
  - y → Labels matrix
  - train → Train matrix
  - test → Test matrix
- fit(clf : classifier, x : ndarray, y : ndarray, test : ndarray, train : ndarray) : void → This method provides to fit the classifier
  - clf → The classifier to use
  - x → Features matrix
  - y → Labels matrix

  - test → Test matrix
  - train → Train matrix

- post_fit(clf : classifier, x : ndarray, y : ndarray, test : ndarray, train : ndarray) : void → This method executes operations after the fit computation
  - clf → The classifier to use
  - x → Features matrix
  - y → Labels matrix
  - test → Test matrix
  - train → Train matrix

- pre_predict(clf : classifier, x : ndarray, y : ndarray, test : ndarray, train : ndarray) : void → This method executes operations before the prediction computation
  - clf → The classifier to use
  - x → Features matrix
  - y → Labels matrix
  - test → Test matrix
  - train → Train matrix

- predict(clf : classifier, x : ndarray, y : ndarray, test : ndarray, train : ndarray) : ndarray → This method performs predictions
  - clf → The classifier to use
  - x → Features matrix
  - y → Labels matrix
  - test → Test matrix
  - train → Train matrix

- post_predict(clf : classifier, x : ndarray, y : ndarray, test : ndarray, train : ndarray) : void → This method executes operations after the prediction computation
  - clf → The classifier to use
  - x → Features matrix
  - y → Labels matrix
  - test → Test matrix
  - train → Train matrix
- post_folds(clf : classifier, x : ndarray, y : ndarray, y_labels : ndarray, y_predictions : ndarray, y_tests : ndarray) : void → This method executes operations after the k-fold computation
  - clf → The classifier to use
  - x → Features matrix
  - y → Labels matrix
  - y_labels → Many labels vectors
  - y_predictions → Many prediction labels vectors
  - y_tests → Many test labels vectors
- __init__(features : dict) → This method is the constructor of the class
  - features → Dictionary with samples as keys and another dictionary as values which has features as keys and values

algorithms::disclosure

This package provides classes to implement Disclosure detector algorithm

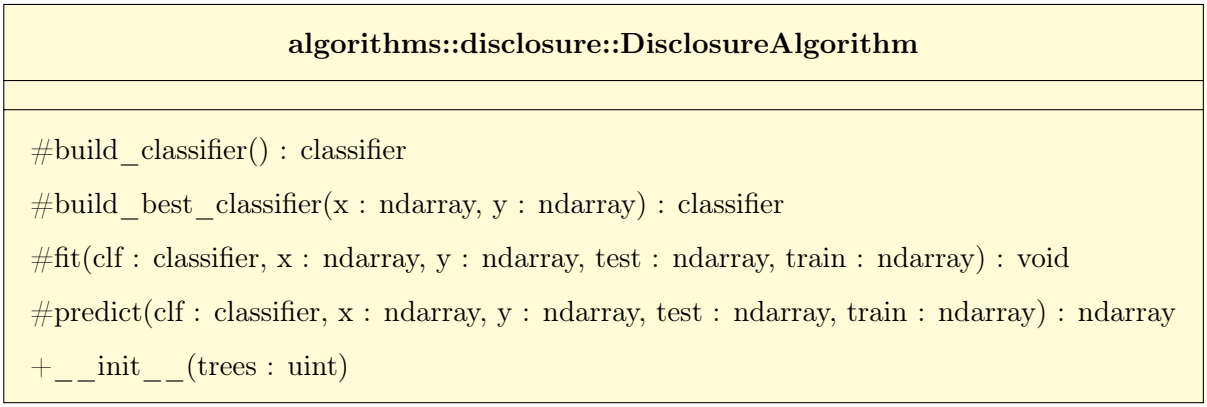

Figure 3.7: Class diagram - algorithms::disclosure::DisclosureAlgorithm

algorithms::disclosure::DisclosureAlgorithm

Description

This class implements Disclosure detection algorithm

Parent classes

- algorithms::AbstractAlgorithm

Methods

- build_classifier() : classifier → This method builds a classifier
- build_best_classifier(x : ndarray, y : ndarray) : classifier → This method builds the best classifier based on features
  - x → Features matrix
  - y → Labels matrix
- fit(clf : classifier, x : ndarray, y : ndarray, test : ndarray, train : ndarray) : void → This method provides to fit the classifier
  - clf → The classifier to use
  - x → Features matrix

  - y → Labels matrix
  - test → Test matrix
  - train → Train matrix
- predict(clf : classifier, x : ndarray, y : ndarray, test : ndarray, train : ndarray) : ndarray → This method performs predictions
  - clf → The classifier to use
  - x → Features matrix
  - y → Labels matrix
  - test → Test matrix
  - train → Train matrix
- __init__(trees : uint) → This function is the constructor method
  - trees → Number of trees in the Random Forest classifier

algorithms::bottrack

This package provides classes to implement BotTrack detector algorithm

algorithms::bottrack::BotTrackAlgorithm

Description

This class implements BotTrack detection algorithm

Parent classes

- algorithms::AbstractAlgorithm

Methods

- build_classifier() : classifier → This method builds a classifier
- build_best_classifier(x : ndarray, y : ndarray) : classifier → This method builds the best classifier based on features

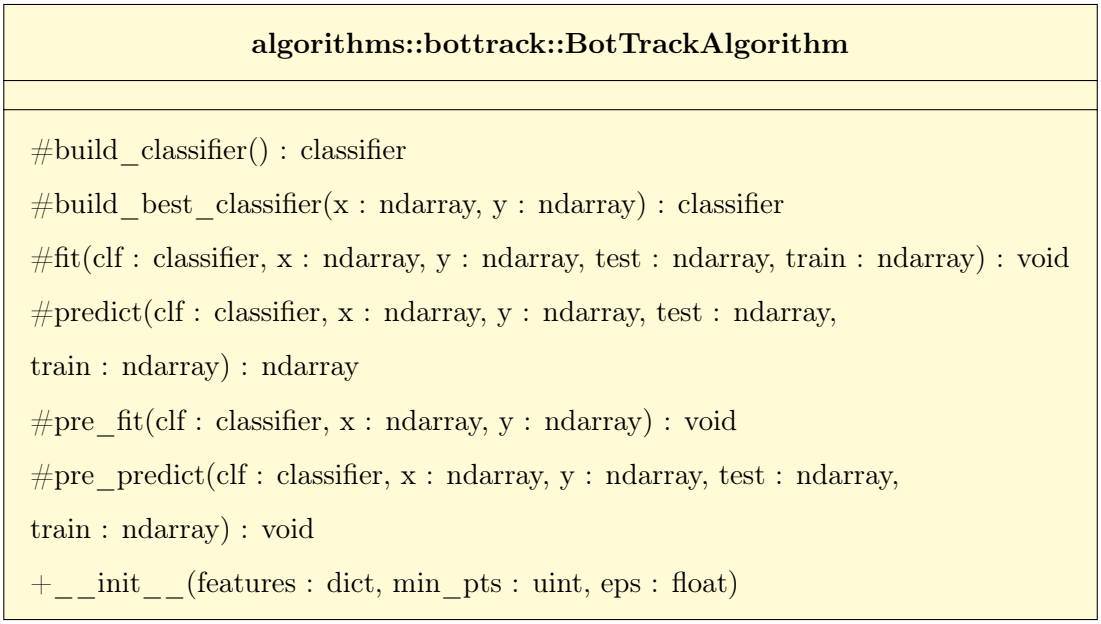

Figure 3.8: Class diagram - algorithms::bottrack::BotTrackAlgorithm

  - x → Features matrix
  - y → Labels matrix

- fit(clf : classifier, x : ndarray, y : ndarray, test : ndarray, train : ndarray) : void → This method provides to fit the classifier
  - clf → The classifier to use
  - x → Features matrix
  - y → Labels matrix
  - test → Test matrix
  - train → Train matrix
- predict(clf : classifier, x : ndarray, y : ndarray, test : ndarray, train : ndarray) : ndarray → This method performs predictions
  - clf → The classifier to use
  - x → Features matrix

  - y → Labels matrix
  - test → Test matrix
  - train → Train matrix

- pre_fit(clf : classifier, x : ndarray, y : ndarray) : void → This method executes operations before the k-fold computation
  - clf → The classifier to use
  - x → Features matrix
  - y → Labels matrix

- pre_predict(clf : classifier, x : ndarray, y : ndarray, test : ndarray, train : ndarray) : void → This method executes operations before the prediction computation
  - clf → The classifier to use
  - x → Features matrix
  - y → Labels matrix
  - test → Test matrix
  - train → Train matrix

- __init__(features : dict, min_pts : uint, eps : float) → This function is the constructor method
  - features → Dictionary with samples as keys and another dictionary as values which has features as keys and values
  - min_pts → Minimum point such a cluster is not considered as noise in DBSCAN
  - eps → Maximum euclidean distance from a point to be considered neighbor in DBSCAN

3.4.2 datasets package

This package provides classes to handle with datasets

datasets::AbstractDataset

Description

This class is the abstract base class for each dataset

Children classes

- datasets::UnlabeledDataset

datasets::UnlabeledDatasetInterface

Description

This is the interface for each unlabeled dataset

Children classes

- datasets::LabeledDatasetInterface
- datasets::UnlabeledDataset

Methods

- netflow() : ndarray → This method gets the netflow matrix

datasets::LabeledDatasetInterface

Description

This is the interface for each labeled dataset

Parent classes

- datasets::UnlabeledDatasetInterface

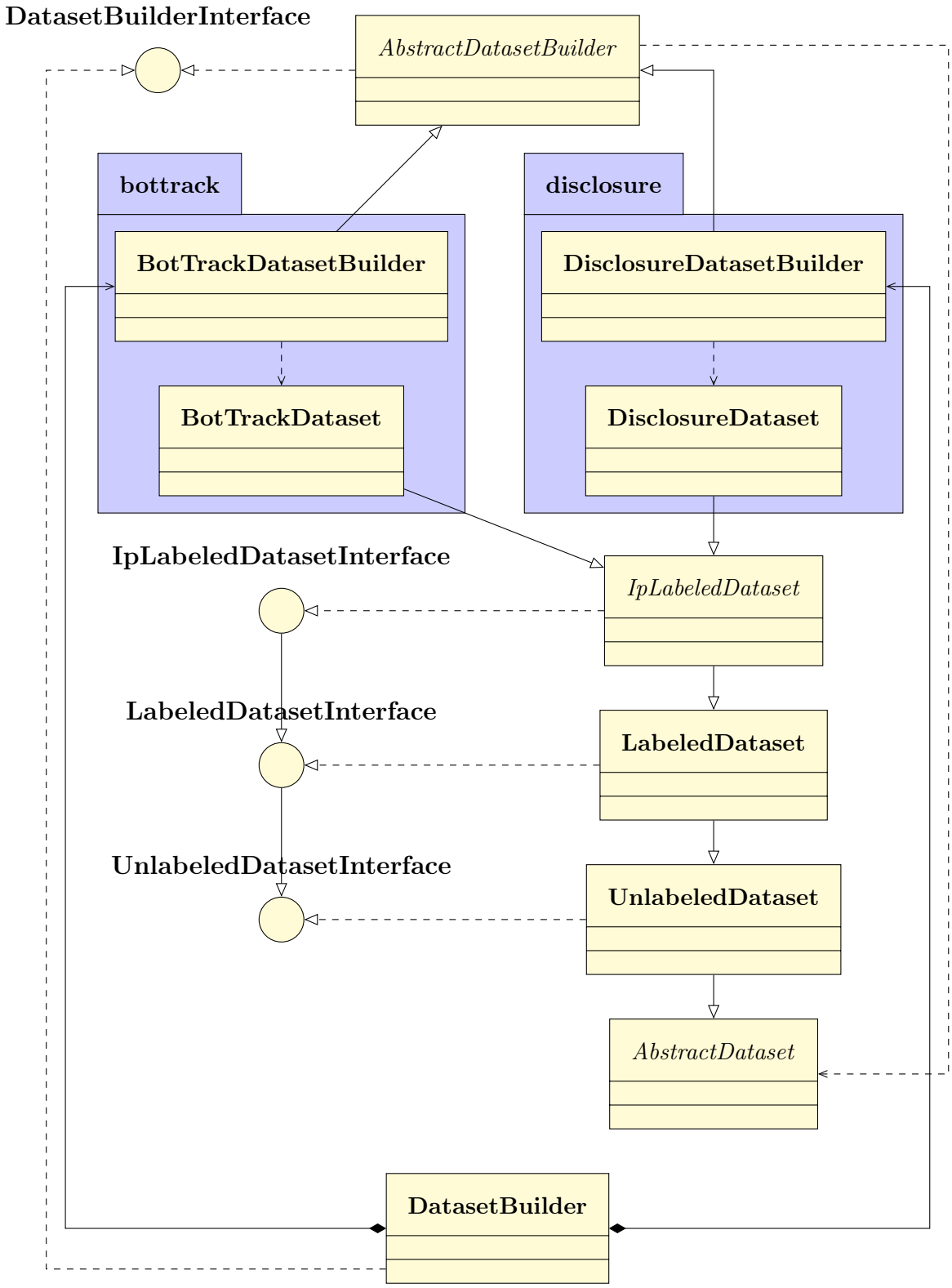

Figure 3.9: Package diagram: datasets

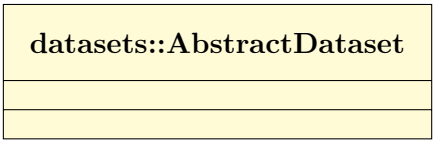

Figure 3.10: Class diagram - datasets::AbstractDataset

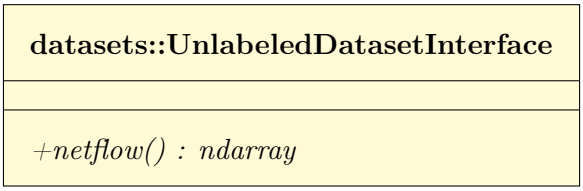

Figure 3.11: Class diagram - datasets::UnlabeledDatasetInterface

Children classes

- datasets::IpLabeledDatasetInterface
- datasets::LabeledDataset

Methods

- labels() : ndarray → This method gets the labels matrix

datasets::IpLabeledDatasetInterface

Description

This is the interface for each IP-labeled dataset

Parent classes

- datasets::LabeledDatasetInterface

Children classes

- datasets::IpLabeledDataset

Methods

- labeled_ips() : dict → This method gets labels for each samples

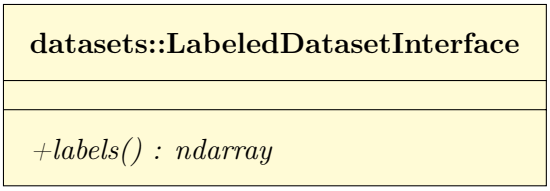

Figure 3.12: Class diagram - datasets::LabeledDatasetInterface

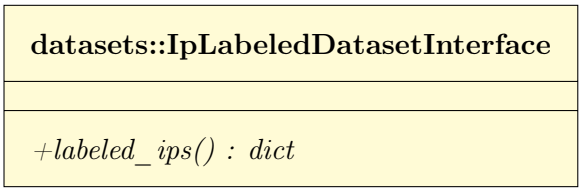

Figure 3.13: Class diagram - datasets::IpLabeledDatasetInterface

datasets::UnlabeledDataset

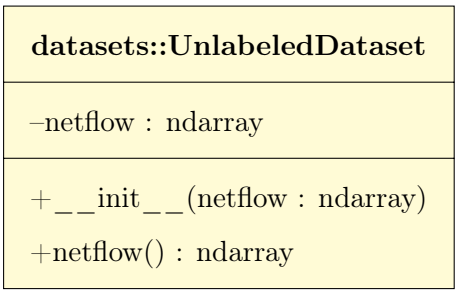

Figure 3.14: Class diagram - datasets::UnlabeledDataset

Description

This class represents an unlabeled dataset

Parent classes

- datasets::AbstractDataset
- datasets::UnlabeledDatasetInterface

Children classes

- datasets::LabeledDataset

Attributes

- netflow : ndarray (Netflow matrix)

Methods

- __init__(netflow : ndarray) → This method is the constructor of the class
  - netflow → Netflow matrix
- netflow() : ndarray → This method gets the netflow matrix

datasets::LabeledDataset

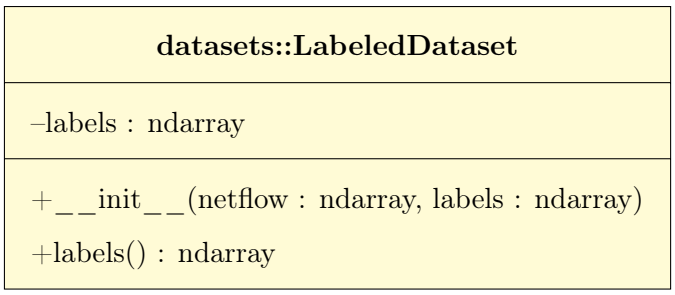

Figure 3.15: Class diagram - datasets::LabeledDataset

Description

This class represents a labeled dataset

Parent classes

- datasets::LabeledDatasetInterface
- datasets::UnlabeledDataset

Children classes

- datasets::IpLabeledDataset

Attributes

- labels : ndarray (Labels matrix)

Methods

- __init__(netflow : ndarray, labels : ndarray) → This method is the constructor of the class
  - netflow → Netflow matrix
  - labels → Labels matrix
- labels() : ndarray → This method gets the labels matrix

datasets::IpLabeledDataset

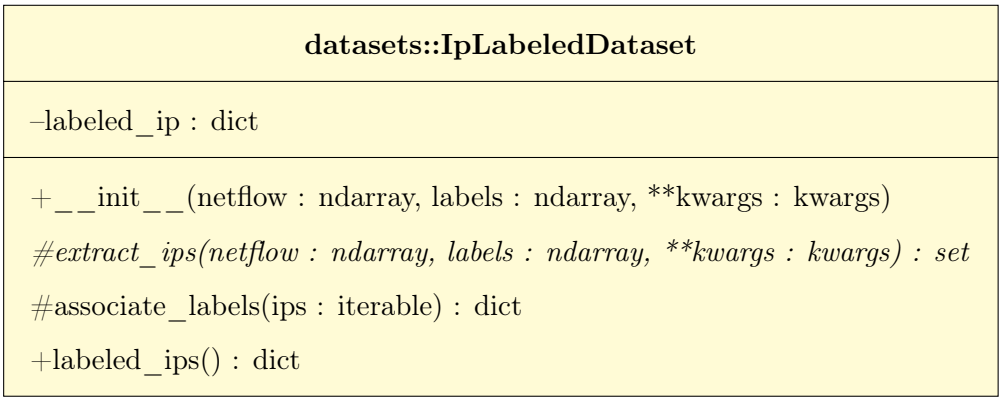

Figure 3.16: Class diagram - datasets::IpLabeledDataset

Description

This class is an abstract base class for each IP-labeled dataset

Parent classes

- datasets::IpLabeledDatasetInterface

- datasets::LabeledDataset

Children classes

- datasets::disclosure::DisclosureDataset
- datasets::bottrack::BotTrackDataset

Attributes

- labeled_ip : dict (Labels for each samples)

Methods

- __init__(netflow : ndarray, labels : ndarray, **kwargs : kwargs) → This method is the constructor of the class
  - netflow → Netflow matrix
  - labels → Labels matrix
  - **kwargs → Other keyword arguments
- extract_ips(netflow : ndarray, labels : ndarray, **kwargs : kwargs) : set → This method extracts IPs
  - netflow → Netflow matrix
  - labels → Labels matrix
  - **kwargs → Other keyword arguments
- associate_labels(ips : iterable) : dict → This method associates labels to each IP
  - ips → IPs
- labeled_ips() : dict → This method gets labels for each samples

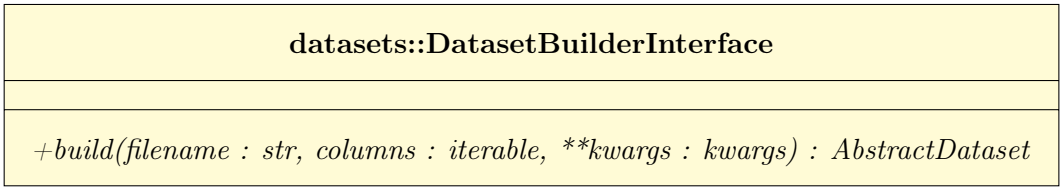

Figure 3.17: Class diagram - datasets::DatasetBuilderInterface

datasets::DatasetBuilderInterface

Description

This is the interface for each dataset builder

Children classes

- datasets::AbstractDatasetBuilder
- datasets::DatasetBuilder

Methods

- build(filename : str, columns : iterable, **kwargs : kwargs) : AbstractDataset → This method build the dataset
  - filename → File to load netflow from
  - columns → Columns of the netflow
  - **kwargs → Other keyword arguments

datasets::AbstractDatasetBuilder

Description

This class is an abstract base class for each dataset builder

Parent classes

- datasets::DatasetBuilderInterface

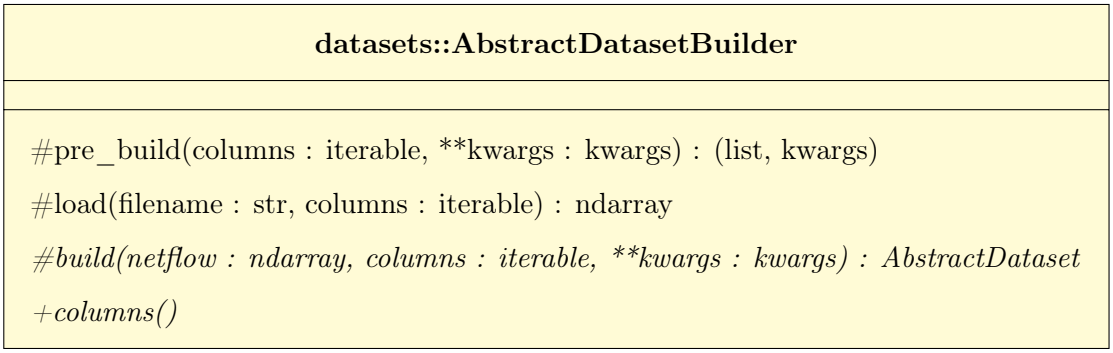

Figure 3.18: Class diagram - datasets::AbstractDatasetBuilder

Children classes

- datasets::disclosure::DisclosureDatasetBuilder
- datasets::bottrack::BotTrackDatasetBuilder

Methods

- pre_build(columns : iterable, **kwargs : kwargs) : (list, kwargs) → This method performs operations before build method call
  - columns → Columns of the netflow
  - **kwargs → Other keyword arguments
- load(filename : str, columns : iterable) : ndarray → This method loads the netflow from file
  - filename → File to load netflow from
  - columns → Columns of the netflow
- build(netflow : ndarray, columns : iterable, **kwargs : kwargs) : AbstractDataset → This method build the dataset and it is accessed from build public method
  - netflow → Netflow matrix
  - columns → Columns of the netflow

  - **kwargs → Other keyword arguments

- columns() → This method gets the columns to extract from the netflow

datasets::DatasetBuilder

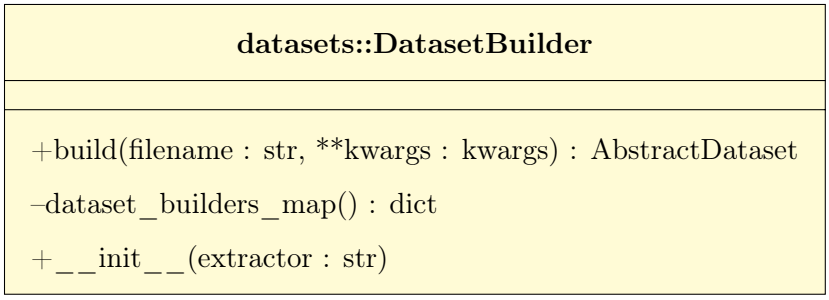

Figure 3.19: Class diagram - datasets::DatasetBuilder

Description

This class binds to many dataset builders depending on parameters

Parent classes

- datasets::DatasetBuilderInterface

Methods

- build(filename : str, **kwargs : kwargs) : AbstractDataset → This method build the dataset
  - filename → File to load netflow from
  - **kwargs → Other keyword arguments
- dataset_builders_map() : dict → This method provides a map to bind a string to a dataset builder

- __init__(extractor : str) → This method is the constructor of the class
  - extractor → Which dataset builder to build

datasets::disclosure

This package provides classes to handle with Disclosure datasets

datasets::disclosure::DisclosureDataset

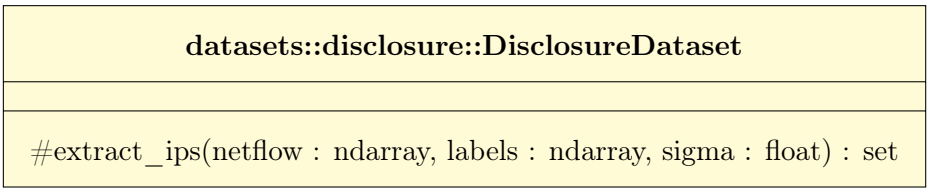

Figure 3.20: Class diagram - datasets::disclosure::DisclosureDataset

Description

This class represents a Disclosure dataset

Parent classes

- datasets::IpLabeledDataset

Methods

- extract_ips(netflow : ndarray, labels : ndarray, sigma : float) : set → This method extracts server IPs which are nodes with number of incoming connections around a certain standard deviation from the maximum
  - netflow → Netflow matrix
  - labels → Labels matrix
  - sigma → Standard deviation

datasets::disclosure::DisclosureDatasetBuilder

datasets::disclosure::DisclosureDatasetBuilder
*#build(netflow : ndarray, columns : iterable, **kwargs : kwargs) : DisclosureDataset* *+columns()*

Figure 3.21: Class diagram - datasets::disclosure::DisclosureDatasetBuilder

Description

This class provides a builder for Disclosure datasets

Parent classes

- datasets::AbstractDatasetBuilder

Methods

- build(netflow : ndarray, columns : iterable, **kwargs : kwargs) : DisclosureDataset → This method build the dataset and it is accessed from build public method
  - netflow → Netflow matrix
  - columns → Columns of the netflow
  - **kwargs → Other keyword arguments
- columns() → This method gets the columns to extract from the netflow

datasets::bottrack

This package provides classes to handle with BotTrack datasets

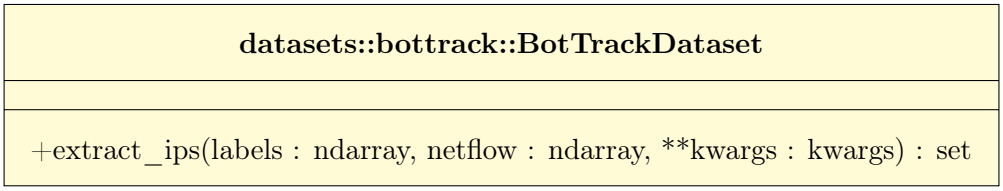

Figure 3.22: Class diagram - datasets::bottrack::BotTrackDataset

datasets::bottrack::BotTrackDataset

Description

This class represents a BotTrack dataset

Parent classes

- datasets::IpLabeledDataset

Methods

- extract_ips(labels : ndarray, netflow : ndarray, **kwargs : kwargs) : set → This method extracts every unique IP
  - labels → Labels matrix
  - netflow → Netflow matrix
  - **kwargs → Other keyword arguments

datasets::bottrack::BotTrackDatasetBuilder

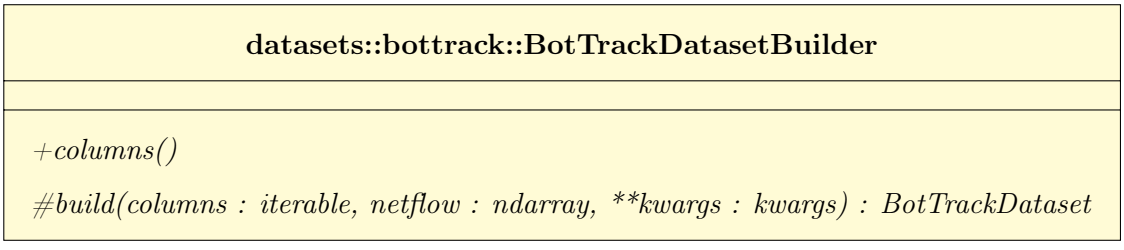

Figure 3.23: Class diagram - datasets::bottrack::BotTrackDatasetBuilder

Description

This class provides a builder for BotTrack datasets

Parent classes

- datasets::AbstractDatasetBuilder

Methods

- columns() → This method gets the columns to extract from the netflow
- build(columns : iterable, netflow : ndarray, **kwargs : kwargs) : BotTrackDataset → This method build the dataset and it is accessed from build public method
  - columns → Columns of the netflow
  - netflow → Netflow matrix
  - **kwargs → Other keyword arguments

3.4.3 extractors package

This package provides classes to implement features extractors

extractors::ExtractorInterface

Description

This is the interface for each extractor

Children classes

- extractors::Extractor
- extractors::disclosure::DisclosureExtractor
- extractors::bottrack::BotTrackExtractor

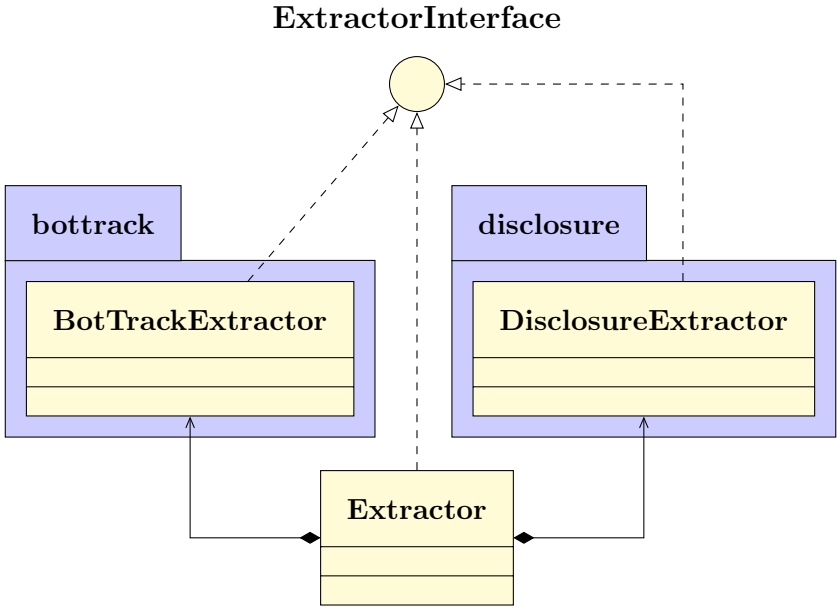

Figure 3.24: Package diagram: extractors

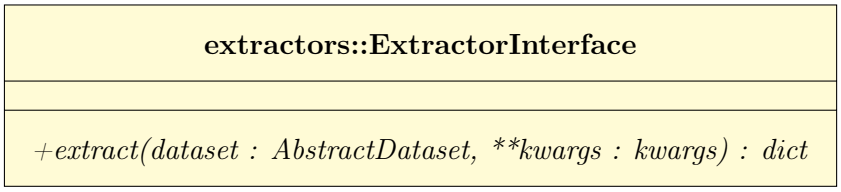

Figure 3.25: Class diagram - extractors::ExtractorInterface

Methods

- extract(dataset : AbstractDataset, **kwargs : kwargs) : dict → This method provides to extract features from the dataset
  - dataset → Dataset which extract features from
  - **kwargs → Other keyword arguments

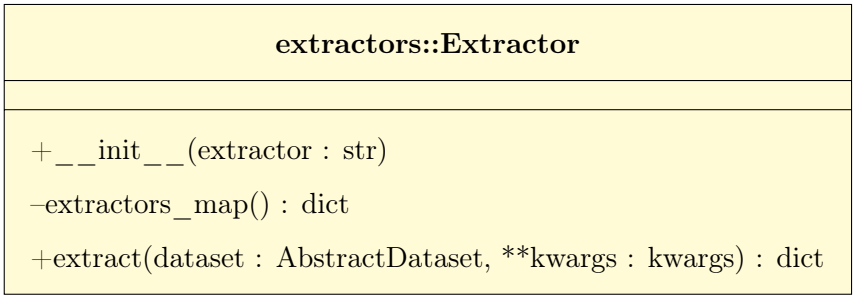

Figure 3.26: Class diagram - extractors::Extractor

extractors::Extractor

Description

This class binds to many features extractors depending on parameters

Parent classes

- extractors::ExtractorInterface

Methods

- __init__(extractor : str) → This method is the constructor of the class
  - extractor → Which extractor to build
- extractors_map() : dict → This method provides a map to bind a string to an extractor
- extract(dataset : AbstractDataset, **kwargs : kwargs) : dict → This method provides to extract features from the dataset
  - dataset → Dataset which extract features from
  - **kwargs → Other keyword arguments

extractors::disclosure

This package provides classes to implement Disclosure features extractor algorithm

extractors::disclosure::DisclosureExtractor

extractors::disclosure::DisclosureExtractor
+extract(dataset : DisclosureDataset, autocorrelation : uint, unmatched : uint) : dict –extract_flows(dataset : DisclosureDataset) : dict –extract_statistical_features(q : Queue, flows_by_server : dict) : dict –extract_autocorrelation_features(q : Queue, flows_by_server : dict, dataset : DisclosureDataset, autocorrelation : uint) : dict –extract_unique_flow_sizes_features(q : Queue, flows_by_server : dict) : dict –extract_kurtosis_and_entropy(flow_sizes : ndarray) : dict –regular_access_patterns_features(q : Queue, flows_by_server : dict) : dict –unmatched_flow_density_features(q : Queue, flows_by_server : dict, dataset : DisclosureDataset, unmatched : uint) : dict

Figure 3.27: Class diagram - extractors::disclosure::DisclosureExtractor

Description

This class is a features extractor for Disclosure algorithm

Parent classes

- extractors::ExtractorInterface

Methods

- extract(dataset : DisclosureDataset, autocorrelation : uint, unmatched : uint) : dict → This method provides to extract features from the dataset
  - dataset → Dataset which extract features from
  - autocorrelation → Number of time series for computing autocorrelation features
  - unmatched → Number of time series for computing unmatched flow features
- extract_flows(dataset : DisclosureDataset) : dict → This method provide to create of an easy to iterate data structure from the dataset
  - dataset → Dataset which extract features from
- extract_statistical_features(q : Queue, flows_by_server : dict) : dict → This method extracts statistical features
  - q → Queue where put result in
  - flows_by_server → Easy to iterate data structure with flows
- extract_autocorrelation_features(q : Queue, flows_by_server : dict, dataset : DisclosureDataset, autocorrelation : uint) : dict → This method extracts autocorrelation features
  - q → Queue where put result in
  - flows_by_server → Easy to iterate data structure with flows
  - dataset → Dataset which extract features from
  - autocorrelation → Number of time series for computing autocorrelation features
- extract_unique_flow_sizes_features(q : Queue, flows_by_server : dict) : dict → This method extracts unique flow sizes features

  - q → Queue where put result in
  - flows_by_server → Easy to iterate data structure with flows

- extract_kurtosis_and_entropy(flow_sizes : ndarray) : dict → This method extracts kurtosis of given flow sizes and entropy on unique flow size
  - flow_sizes → Vector of flow sizes
- regular_access_patterns_features(q : Queue, flows_by_server : dict) : dict → This method extracts regular access patterns features
  - q → Queue where put result in
  - flows_by_server → Easy to iterate data structure with flows
- unmatched_flow_density_features(q : Queue, flows_by_server : dict, dataset : DisclosureDataset, unmatched : uint) : dict → This method extracts unmatched flow density features
  - q → Queue where put result in
  - flows_by_server → Easy to iterate data structure with flows
  - dataset → Dataset which extract features from
  - unmatched → Number of time series for computing unmatched flow features

extractors::bottrack

This package provides classes to implement BotTrack features extractor algorithm

extractors::bottrack::BotTrackExtractor

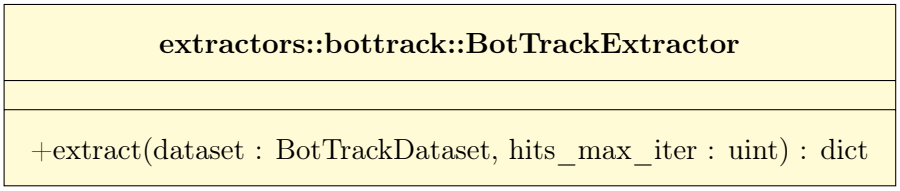

Figure 3.28: Class diagram - extractors::bottrack::BotTrackExtractor

Description

This class is a features extractor for BotTrack algorithm

Parent classes

- extractors::ExtractorInterface

Methods

- extract(dataset : BotTrackDataset, hits_max_iter : uint) : dict
  → This method provides to extract features from the dataset
  - dataset → Dataset which extract features from
  - hits_max_iter → Maximum number of iterations of HITS function

3.4.4 main package

This package contains the main program

main::App

Description

This class is the main class of the application

Attributes

- app_dir : str (The location of the application)

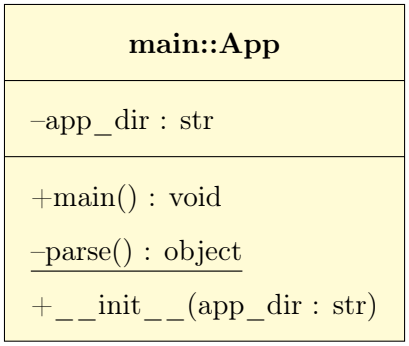

Figure 3.29: Class diagram - main::App

Methods

- main() : void → This method is the main method of the application
- parse() : object → This method provide to parse the command line input
- __init__(app_dir : str) → This method is the constructor of the class
  - app_dir → The location of the application

3.4.5 metrics package

This package provides classes to compute evaluation metrics

metrics::Metrics

Description

This class contains build-in methods to compute evaluation metrics

Methods

- accuracy(y_tests : ndarray, y_predictions : ndarray) : tuple → This method performs accuracy score metric

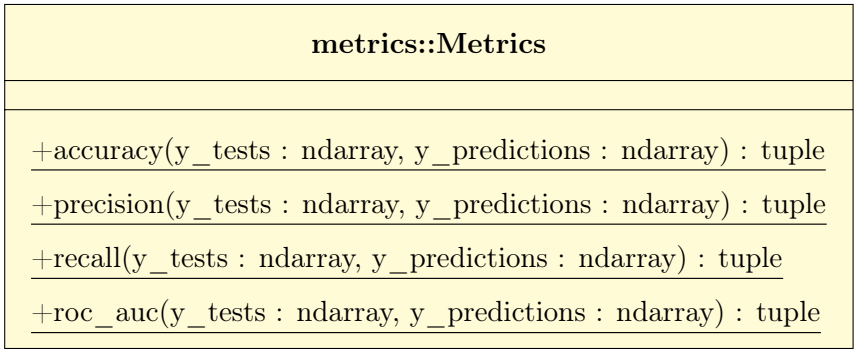

Figure 3.30: Class diagram - metrics::Metrics

  - y_tests → Many test labels vectors
  - y_predictions → Many prediction labels vectors

- precision(y_tests : ndarray, y_predictions : ndarray) : tuple → This method performs precision score metric
  - y_tests → Many test labels vectors
  - y_predictions → Many prediction labels vectors
- recall(y_tests : ndarray, y_predictions : ndarray) : tuple → This method performs recall score metric
  - y_tests → Many test labels vectors
  - y_predictions → Many prediction labels vectors
- roc_auc(y_tests : ndarray, y_predictions : ndarray) : tuple → This method performs ROC AUC score metric
  - y_tests → Many test labels vectors
  - y_predictions → Many prediction labels vectors

3.4.6 outputs package

This package provides classes to handle with outputs

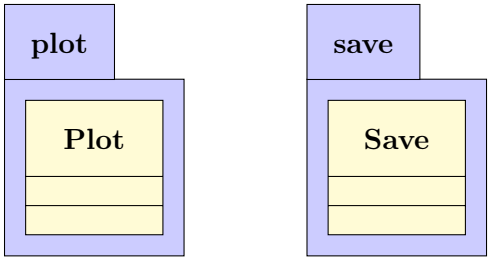

Figure 3.31: Package diagram: outputs

outputs::save

This package provides classes to save outputs into files

outputs::save::Save

outputs::save::Save
+metrics(filename : str, y_tests : ndarray, y_predictions : ndarray, y_labels : ndarray, which : iterable, bots : iterable, sort_keys : bool, indent : uint) : void +features(filename : str, features : dict, which : iterable, bots : iterable, sort_keys : bool, indent : uint) : void +predictions(filename_tests : str, filename_predictions : str, filename_labels : str, y_tests : ndarray, y_predictions : ndarray, y_labels : ndarray, bots : iterable, indent : uint) : void

Figure 3.32: Class diagram - outputs::save::Save

Description

This class contains build-in methods to save outputs into files

Methods

- metrics(filename : str, y_tests : ndarray, y_predictions : ndarray, y_labels : ndarray, which : iterable, bots : iterable, sort_keys : bool, indent : uint) : void → This method save evaluation metrics into a JSON file
  - filename → File where save data into
  - y_tests → Tests labels
  - y_predictions → Predictions labels
  - y_labels → Labels of samples
  - which → Which metrics save into the file
  - bots → Which botnets save into the file
  - sort_keys → If JSON should have sorted keys
  - indent → Indent of the JSON file
- features(filename : str, features : dict, which : iterable, bots : iterable, sort_keys : bool, indent : uint) : void → This method save extracted features into a JSON file
  - filename → File where save data into
  - features → Dictionary with samples as keys and another dictionary as values which has features as keys and values
  - which → Which features save into the file
  - bots → Which samples save into the file
  - sort_keys → If JSON should have sorted keys
  - indent → Indent of the JSON file
- predictions(filename_tests : str, filename_predictions : str, filename_labels : str, y_tests : ndarray, y_predictions : ndarray, y_labels : ndarray, bots : iterable, indent : uint) : void → This method save predictions into a JSON file

- filename_tests → File where save tests labels into
- filename_predictions → File where save predictions labels into
- filename_labels → File where save labels of samples into
- y_tests → Many test labels vectors
- y_predictions → Many prediction labels vectors
- y_labels → Many labels vectors
- bots → Which botnets save into the file
- indent → Indent of the JSON file

outputs::plot

This package provides classes to save plots into files

outputs::plot::Plot

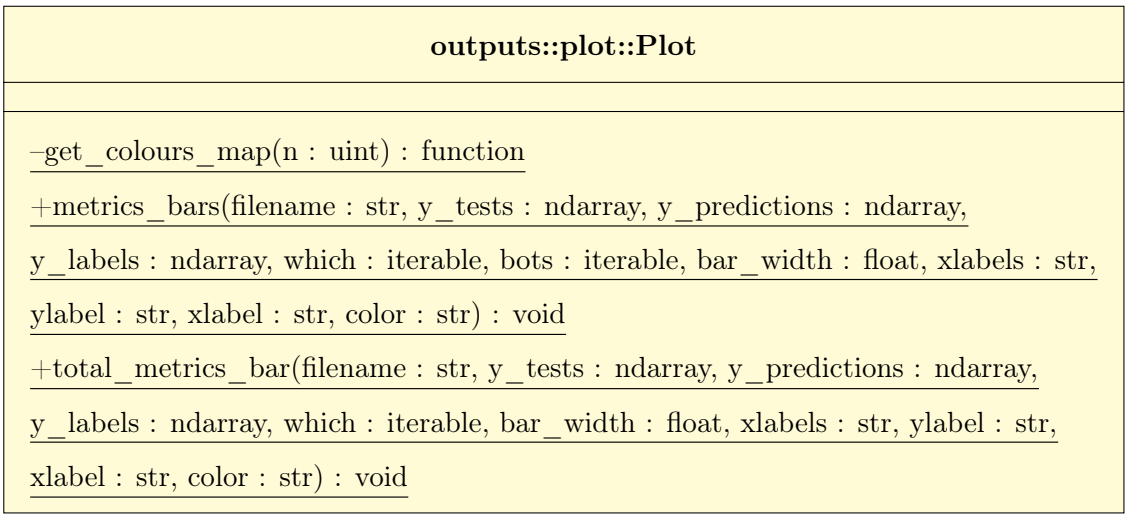

Figure 3.33: Class diagram - outputs::plot::Plot

Description

This class contains build-in methods to save plots into files

Methods

- get_colours_map(n : uint) : function → This method gets a function which generates colors in a range
  - n → Range of colors to generate
- metrics_bars(filename : str, y_tests : ndarray, y_predictions : ndarray, y_labels : ndarray, which : iterable, bots : iterable, bar_width : float, xlabels : str, ylabel : str, xlabel : str, color : str) : void → This method plots a bar chart with metrics for each detected botnet
  - filename → File where save data into
  - y_tests → Tests labels
  - y_predictions → Predictions labels
  - y_labels → Labels of samples
  - which → Which are metrics to plot
  - bots → Which are botnets to plot
  - bar_width → How wide plot the bars
  - xlabels → Labels of the X axis ticks
  - ylabel → Label of the Y axis
  - xlabel → Label of the X axis
  - color → Bars color
- total_metrics_bar(filename : str, y_tests : ndarray, y_predictions : ndarray, y_labels : ndarray, which : iterable, bar_width : float, xlabels : str, ylabel : str, xlabel : str, color : str) : void → This method plots a bar chart with metrics of total detected botnets
  - filename → File where save data into

- y_tests → Tests labels
- y_predictions → Predictions labels
- y_labels → Labels of samples
- which → Which are metrics to plot
- bar_width → How wide plot the bars
- xlabels → Labels of the X axis ticks
- ylabel → Label of the Y axis
- xlabel → Label of the X axis
- color → Bars color

3.5 Design patterns

Some of the purposes of object-oriented software are building reusable systems, handling complexity and trying to achieve the best maintainability level. [21]

We need, during the designing process, to find pertinent objects, implement them into classes at the right granularity, define interfaces and inheritance hierarchies, and establish relationships among them. The design should be specific to the problem but also general enough to address future problems and requirements. Generalization is a good practice when used with judgment which avoid future problems and maximize reusability. The main purpose is to avoid redesign. Design patterns help to choose the right design that make a system reusable and avoid custom alternatives that compromise reusability. [21]

The main purpose of BFF is to be modular and reusable so proper design patters are use to achieve this fundamental goal.

Strategy. Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it. [21]

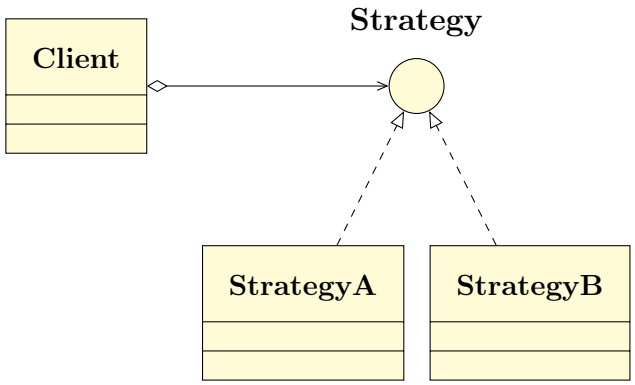

Figure 3.34: Design pattern diagram - Strategy

We use strategy pattern in botnet detectors algorithms. Indeed each algorithm implements the same public interface `tAlgorithmInterface` and it allowed the client to call known methods even if the implementations are different.

Proxy. Proxy patter provides a surrogate or placeholder for another object to control access to it. [21]

Both `Algorithm`, `DatasetBuilder` and `Extractor` classes are proxies. We use proxy pattern in those classes because they have to control access to the right algorithm, dataset builder or extractor. For instance the creation of `Algorithm` class needs a string parameter which identify what kind of algorithm is required. The string is a key in a dictionary of classes which are algorithms and `Algorithm` provide to control every access to itself binding requests to the proper object.

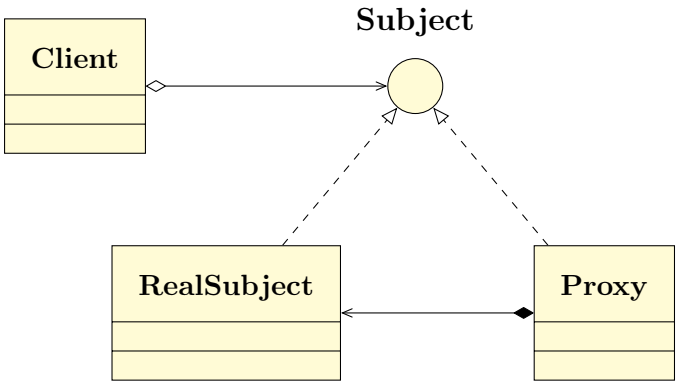

Figure 3.35: Design pattern diagram - Proxy

Template Method. Template Method pattern defines the skeleton of an algorithm through operations that are different on some steps into subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. [21]

We use template method pattern in feature extractors and in botnet detectors algorithms. For instance we implement `stratified_k_fold(folds : int)` method in `AbstractAlgorithm` using this pattern. We define the skeleton of the method but we leave some abstract operation which are only defined in `DisclosureAlgorithm` or in `BotTrackAlgorithm`.

Builder. Builder pattern purpose is to separate the construction of a complex object from its representation [21]. In this way the responsibility of object construction is delegated to an external class and the same construction process can create different representations of the product.

We use builder pattern to create each dataset class. In particular `AbstractDatasetBuilder` is the base class for every builder and every implemented dataset has a particular builder class to handle the creation of such a complex object.

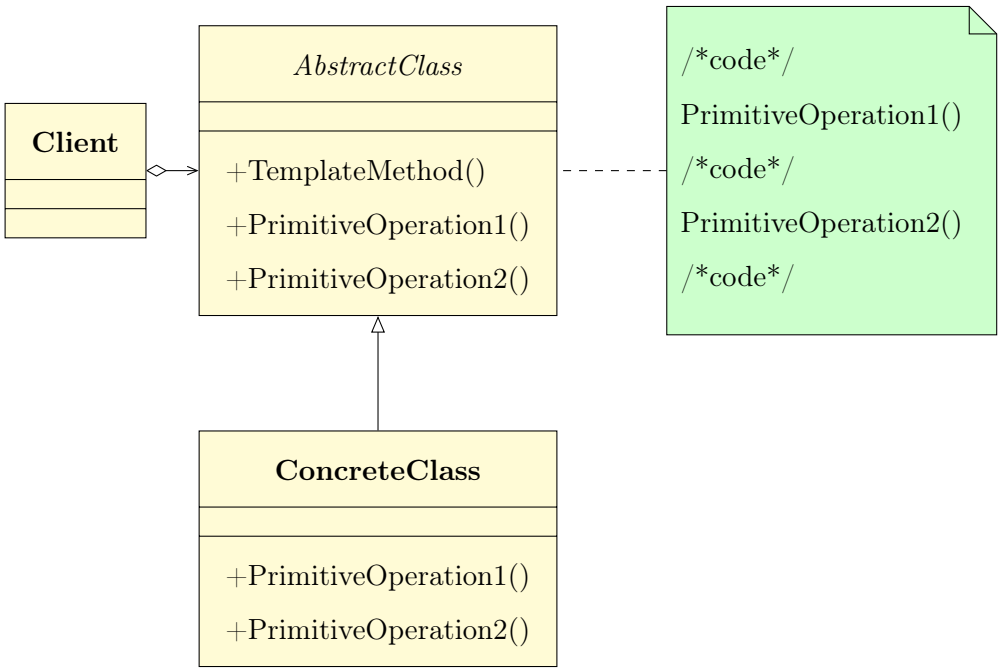

Figure 3.36: Design pattern diagram - Template Method

3.6 Implemented algorithms

As explained in the overview of this work (see Section 1), in order to evaluate ELISA detectability there is the need to test it in a controlled environment while observed by botnet detectors.

We choose Disclosure [2] and BotTrack [3] detectors for our evaluation because they are state of the art botnet detectors. Both of them were unavailable so we reimplemented them on the best of our knowledge. In order to demonstrate their efficacy and our correct implementation they are validated on real traffic (see Section 3.7). We now describe briefly both algorithms.

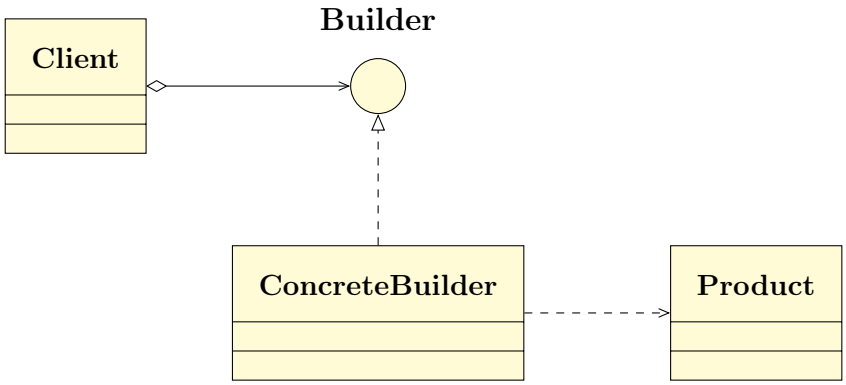

Figure 3.37: Design pattern diagram - Adapter

3.6.1 Disclosure

Disclosure aims to discover C&C servers but not infected machines so it restricts the analysis to servers only. Disclosure applies an heuristic in order to separate servers from the clients. Preliminary tests have shown that heuristic used in the original algorithm do not work so well so we use a different one. This new heuristic counts incoming connections for each IP in the NetFlow and if an IP is within a given variance around the IP with most incoming connections then it is a server. After that, Disclosure extracts features from NetFlow attributes (features aim to best capture botnet behavior without referring to a particular C&C protocol). Besides we decide to ignore temporal features of Disclosure because there are not applicable to our study. At the end a supervised machine learning algorithm as Random Forests Classifier [22] is used to train models on benign and C&C servers. Now we describe which features are extracted and what is the used machine learning algorithm.

Features. Disclosure identifies several groups of features which are divided in two main classes: flow size-based features and client access patterns-based

features.

Flow size-based features are based on flow sizes, which is the number of bytes transferred in one direction between two endpoints in a particular flow. From flow sizes it extracts statistical features like mean and standard deviation for each server. These features characterizes the regularity of a flow size behavior over time. Unique flow sizes features are also computed by Disclosure which counts number of unique flow sizes observed, and then it performs statistical measurements of occurrence density for each of them. Finally, Disclosure transforms a series of connections in a time series and it extracts autocorrelation features for identifying repeating patterns in time.

Client access patterns-based features are based on how clients communicate with servers. A common property of botnets is that infected machines frequently establish connections with a C&C channel. Therefore Disclosure extracts features in order to distinguish malicious client access patterns which are very different from benign ones. It extracts regular access patterns features measuring regularity over time and unmatched flow density features which show if each incoming flow has correspondent outgoing connections and how regularly. High regularly means high probability that the traffic is not generated by a human.

Random Forests Classifier. Random Forests Classifier [22] is an estimator that fits a number of decision tree classifiers on various sub-samples of the dataset. It uses averaging and randomness to improve the predictive accuracy and avoid over-fitting. As described by the authors of Disclosure this is the best classifier for the selected features.

3.6.2 BotTrack

BotTrack detector [3] is designed to detect stealthy botnets which use peer-to-peer communication infrastructures. It analyzes communication behavioral patterns and tries to detect potential botnet activities through linkage analysis and clustering techniques that identify groups of hosts sharing similar behavioral patterns. In particular BotTrack uses HITS algorithm (described below) to compute two features (hubs and authorities which are used to score nodes communication behavior) and then use DBSCAN [23] clustering classifier algorithm (see below) in order to classify similar pattern clusters. Finally it needs to know that some nodes are botnets so it classify all nodes in the same cluster as malicious.

Hubs and Authorities (HITS). Hyperlink-Induced Topic Search (HITS) also known as hubs and authorities is a link analysis algorithm that rates nodes in an Internet network [24]. The basic idea is that certain nodes in an Internet network, known as hubs, served large redirectors that are not necessary authoritative in information they held. They are used as catalogs of information that lead users direct to other nodes. Besides authoritative nodes provide qualitative information. High authority score means being linked from nodes that are recognized as hubs for information. Instead, high hub score means being linked to nodes that are considered to be authorities for information. Therefore a good hub represents a node that is pointed to many other nodes, and a good authority represented a node that is linked to many hubs. [25]

The algorithm computes two scores for each node: its authority score, which is the value of information the node provides, and its hub score, which is the value of how the node links to other nodes.

At the begin the rankings are $\forall p$, $auth(p) = 1$ and $hub(p) = 1$, then algorithm performs a series of iterations consisting of two basic steps: update of authority values and then hub values for each node. First $\forall p$, $auth(p)$ is updated as:

$$auth(p) = \sum_{i=1}^{n} hub(i)$$

where n is the total number of nodes connected to p and i is a node connected to p. Thus the authority score of a node is the sum of all the hub scores of nodes that have a link to it.

Then $\forall p$, $hub(p)$ is updated as:

$$hub(p) = \sum_{i=1}^{n} auth(i)$$

where n is the total number of nodes p connects to and i is a node which p connects to. Thus the hub score of a node is the sum of the authority scores of nodes that have a link from it.

The final scores of each node is determined after infinite repetitions of the process described above. Applying directly this algorithm it leads to diverging values, so we need to normalize the matrix after every iteration. The values obtained from this process will finally converge. [26]

DBSCAN. Density-Based Spatial Alustering of Applications with Noise (DBSCAN) is a density-based data clustering algorithm that given a set of sample in some n dimensions space, it groups them together if they are closely packed (many nearby neighbors) but also exclude outliers points considering them as noise (whose nearest neighbors are too far away).

The algorithm is very simple and it need only two parameters: μ which is how many sample a cluster has to contain at least in order not to be

considered as noise and a distance ϵ which is the distance within samples are considered neighbors.

Then for each sample in the dataset it looks whether it has neighbors searching within ϵ distance. Often and also in our case it performs euclidean distance so the distance from two points is the inner product of two features vectors. So two points a and b are neighbor if $\|a - b\| <= \epsilon$. Distance can also be seen as:

$$\|a - b\| = \sqrt{(a - b) \cdot (a - b)} = \sqrt{(a - b)^T (a - b)} = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$$

where n is the number of dimensions of the data space.

If there are neighbors it proceed considering them as part of the same cluster and then it computes neighbors search again until there are not reachable samples within ϵ distance. If there are not at least $minSamples$ samples for each cluster then they are considered as noise because they do not have enough neighbors.

3.7 Tests

Since we want to ensure quality in our work (see Section 3.1) we need to test our framework. We used unit test to ensure that features extractors do what expected (see 3.7.1). Finally, we validate Disclosure and BotTrack detector algorithms to ensure a correct reimplementation (see 3.7.2). We do not build integration tests due to strict deadlines but we think it will be a future work 5.

3.7.1 Unit test

We use unit testing during software development process in order to test smallest parts of our framework. In particular we use unit testing for features extraction evaluation since it is one of the most important modules and it require particular attention.

We used external tool [27] for manually computing features extractions of a test dataset and the tests were built comparing extraction module output with the expected output. Our software increment were always checked before integration. So we develop only safe and tested source code. Our unit testing was automated in order to speed up development work.

3.7.2 Algorithms validation

Because of our reimplementation of Disclosure and BotTrack we need to validate them with proper tests. We need to assessing performance with data that is heterogeneous enough to simulate real traffic to an acceptable level. Due to the lack of such datasets available we use a constructed dataset provided by ISCX. We modified the dataset because of its fragmentation. In particular we manipulate the starting dates of the flows since the dataset was built merging multiple flow records. We did it in order to eliminate wide pauses between traces since Disclosure for example needs the flow to be continuous.

CNS2014 ISCX dataset. The provided dataset pays a close attention to the following features [13]:

- **Generality:** detectors are often evaluated against a particular or few botnets only, but this approach imposes limitations because these detectors only detect a small number of characteristics describing a very

specific botnet behavior. This dataset aims to be general in order to train detectors against multiple botnets behavior.

- **Realism:** the dataset provides realistic botnet traffic traces because authors claim that botnet traffic is usually generated/captured in a controlled environment and this approach does not demonstrate effective results in real applications.

- **Representativeness:** authors claims that another problem with botnet datasets is the ability of collected network traffic traces in order to reflect real environment. To do it so they combined non overlapping subsets of the following data: ISOT dataset [28], ISCX 2012 IDS dataset [29] and Botnet traffic generated by the Malware Capture Facility Project.

As we said above we modified these non overlapping traces for the approach Disclosure uses. Authors produced two datasets: one for training and one for testing. Testing dataset provides more botnet samples in order to test detectors against botnet which have not samples in the training set. We do not have to test our detector but only validate our reimplementation effectiveness so we merged these two datasets. Methods for retrieve this dataset are described in detail in [13]. Table 3.1 shows distribution and type of botnets in the dataset.

Botnet name	**Type**	**Portion of flows in dataset**
Menti	IRC	24 407 (0.170%)
Murlo	IRC	81 134 (0.567%)
Neris	IRC	497 856 (3.480%)
Rbot	IRC	738 901 (5.166%)

Smoke Bot	P2P	2 708 (0.019%)
Sogou	HTTP	20 614 (0.144%)
Tbot	IRC	37 749 (0.264%)
Virut	HTTP	483 333 (3.379%)
Weasel	P2P	458 283 (3.204%)
Zero Access	P2P	19 758 (0.138%)
Zeus	P2P	16 803 (0.117%)
Others	P2P	5 047 148 (35.28%)

Table 3.1: Distribution of botnet types in the test dataset - total flows are 14 304 458 and others include ISCX bot [29] (P2P), NSIS (P2P), SMTP Spam (P2P) and UDP Storm (P2P)

BotTrack results. As we show in figures 3.38 we achieve excellent result implementing BotTrack detector algorithm.

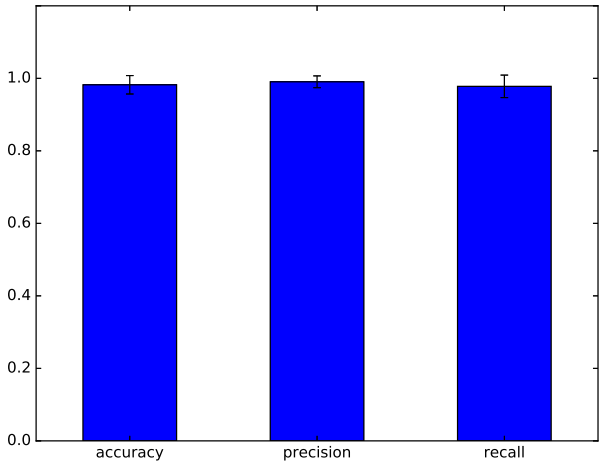

Figure 3.38: Total evaluation metrics

We evaluate it with BFF evaluation module, which perform a strati-

fied k-folds validation. BotTrack achieves a total accuracy score of 98.22% (standard deviation of 2.54%), a total precision score of 99.05% (standard deviation of 1.63%) and a total recall score of 97.78% (standard deviation of 3.11%). Such high scores show how our reimplementation is correct.

Analyzing in detail evaluation metrics for each botnet (as figure A.1 in Appendix A shows) we see that BotTrack does not achieve the same results on every botnet. This is caused by the fact that some botnets (as table 3.1 shows) are too few to train the detector well enough to detect them.

Overall we consider our reimplementation a success since total evaluation scores show us that BotTrack performs very well.

Disclosure results. As we show in figures 3.39 we achieve excellent result implementing BotTrack detector algorithm.

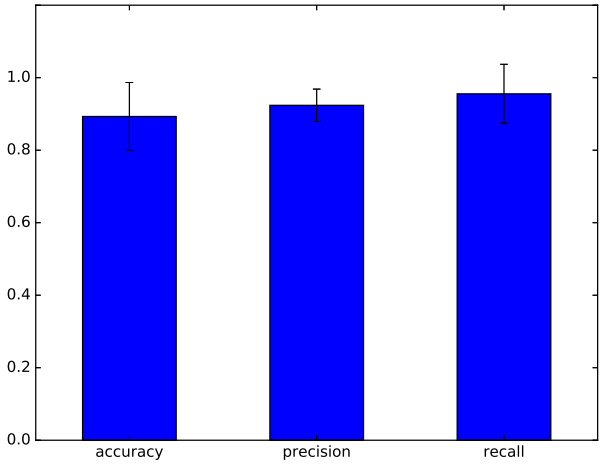

Figure 3.39: Total evaluation metrics

We evaluate it with BFF evaluation module, which perform a stratified k-folds validation. Disclosure achieves a total accuracy score of 89.30% (standard deviation of 9.38%), a total precision score of 92.38% (standard deviation of 4.47%) and a total recall score of 95.56% (standard deviation of

8.14%). Such high scores show how our reimplementation is correct.

Analyzing in detail evaluation metrics for each botnet (as figure A.2 in Appendix A shows) we see that Disclosure does not achieve the same results on every botnet. Besides seems that Disclosure performs quite bad. As for BotTrack this is caused by the fact that some botnets (as table 3.1 shows) are too few to train the detector well enough to detect them or selected features are not relevant enough for some type of botnet.

Overall we consider our reimplementation a success since total evaluation scores show us that Disclosure performs quite well.

Chapter 4

ELISA detectability evaluation

The purpose of the experiment is to evaluate ELISA detectability. We want to know whether the two implemented detector algorithms are capable of spot ELISA among benign traffic and with what precision. In particular we expect either that algorithms will detect ELISA but with a extremely low precision since they will probably flag all Facebook traffic as malicious or that algorithms will not spot anything at all since their inability of discriminating traffic correctly because of the covert channel.

For those reason we performed an experiment, described below.

4.1 Experiment design

The experiment consisted in recording network data such packets information from a student computer laboratory at our University department.

During the recording volunteers generated benign traffic normally using PCs for didactic purposes, studying, working or other activities. Meanwhile we used 4 PCs performing an usual web navigation but also generating malicious traffic when accessing Facebook website.

We recorded traffic for 9 hours consecutively and more than 50 volunteers participated to our experiment. We overall recorded about 38 million flow records.

We only stored information about date, time and duration of flows, used protocols, source and destination IP addresses and length of packets. All of data were stored into NetFlow format.

Because of privacy regulations we anonymized each internal IP address of the laboratory except 4 machines which were used only by us to generate malicious traffic. We needed only to know IP addresses of infected machines in order to flag some of their traffic as malicious. This does not influence result since each other IP were randomly mapped in another but maintaining flows consistency.

After we generated such dataset we applied detection over it.

4.2 Experimental results

We evaluate ELISA detectability with BFF evaluation module on both BotTrack and Disclosure detectors. BFF performed a stratified k-folds validation and both failed to find ELISA.

BotTrack. BotTrack achieved a total accuracy score of 98.49% (standard deviation of 0.946%). With such a result seems that BotTrack is able to find ELISA traffic among benign one but it is not true. Looking at predictions we saw that BotTrack flagged almost all traffic as benign so that implies an high total accuracy score because most of the traffic is benign.

As a matter of fact the BotTrack's total precision score and total recall score were 0% that means that there were no true positive samples that is traffic classified as malicious. So we demonstrated that BotTrack was unable

to detect ELISA and we consider this result as a success and an empirical proof of ELISA undetectability.

That failure is due to ELISA covert channel and how BotTrack detector works. It classifies malicious traffic whether it comes form or goes to an IP address classified as malicious. So it failed classifying social network server machines as malicious because only few communications from or to them are actually malicious.

In fact both BotTrack is trained to detect botnet behavior but after that it performs predictions. BotTrack considers any incoming or outgoing flow through an IP that is considered malicious as malicious traffic.

ELISA malicious traffic do not use unique IP addresses for C&C communications since social networks website use Content Delivery Network (CDN) or multiple addresses to handle the great number of connections for example. Besides not all incoming or outgoing traffic of a social network website is or should be considered as malicious since most of user interact with the website without generating botnet traffic. For those reasons both BotTrack and Disclosure failed to find ELISA among all benign social traffic.

Disclosure. To test ELISA with Disclosure we sampled the traffic to reduce the dimension, because Disclosure needs too much computational time. Moreover we modified the heuristic that decides which IP addresses are servers or clients to reduce the extreme high number of selected servers.

Unlike BotTrack, Disclosure was able to find ELISA. It seemed to perform quite well since in one of the k-folds it achieved a precision of 100%. However, since the Disclosure classifier learns the botnet behavior, it had, in fact, learned how Facebook traffic behaves. Like BotTrack, Disclosure considers any incoming or outgoing flow through an IP that is considered malicious as

malicious traffic.

As a matter of fact 92.42% of the analyzed traffic was benign but Disclosure classified that traffic as malicious since it come from or go to a malicious IP. In such a way, Disclosure flagged too many benign communications as malicious. We considered that result as a Disclosure failure.

Chapter 5

Conclusions and future work

In this work we presented BFF, a new framework designed as a development tool for botnet development and evaluation. BFF was design to be modular, extensible and easy to use. One of the purposes of our work is that it can be used as a common tool for compute evaluations by researcher. This would be useful because it could permit to compute comparisons between botnets. Besides, as framework, it offer many tools to speed up researcher work.

Thereafter we tested and used BFF implementing two botnet detectors as BotTrack and Disclosure. We validated our reimplementation testing them on an real dataset containing real botnets. We considered our reimplementation a success since we used BFF evaluation module to see how well they performed. However the original datasets which both of them were tested with were unavailable so we can not compare original performances evaluations with ours.

Finally we showed that both BotTrack and Disclosure were unable to find a new elusive botnet as ELISA is. Both of them failed because ELISA uses a covert channel for C&C communications. Our claim is supported by the fact that both BotTrack and Disclosure were proved to be well implemented,

since they found with high accuracy and precision many well known botnets, but they failed to find ELISA anyway. We explained the detectors behavior showing why they are unable to find any C&C communication while ELISA's covert channel is used.

As future work, we plan to:

- improve BFF usability introducing more parametrized methods with settable options from configuration files;
- improve modularity of BFF trying to split it into more separate modules;
- implement other detector algorithms which can either confirm or deny our claims about the currently proved ELISA undetectability;
- merge BotTrack and Disclosure algorithms attempting to build a new and powerful detector;
- design and model a new detector capable of find ELISA or other botnets that use covert channels since the purpose of research in security is to defend us from botnet infections.

Appendix A

Plots

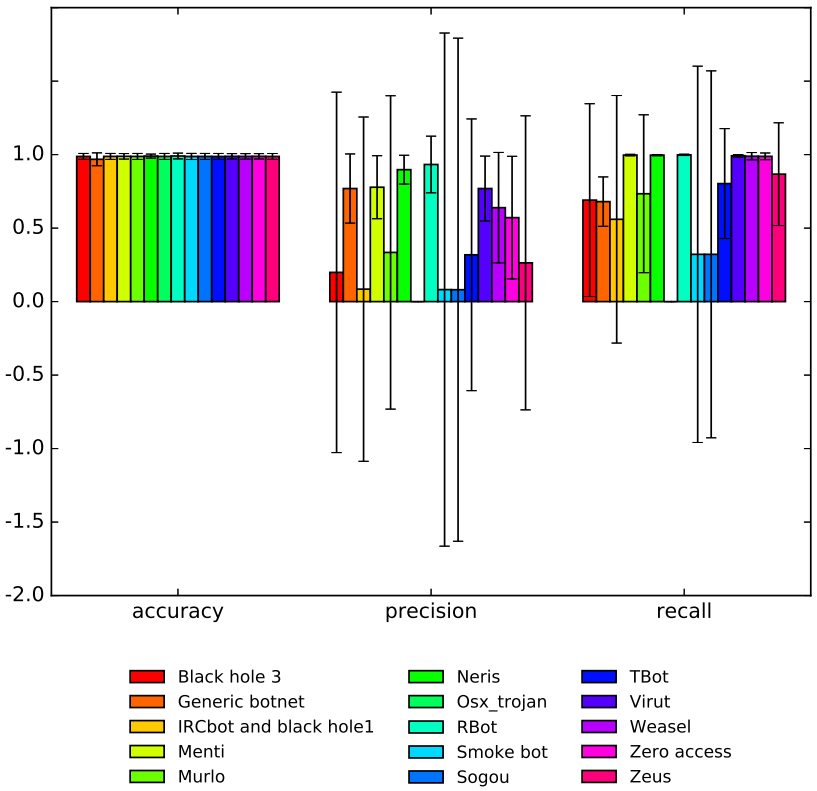

Figure A.1: Evaluation metrics for each botnet

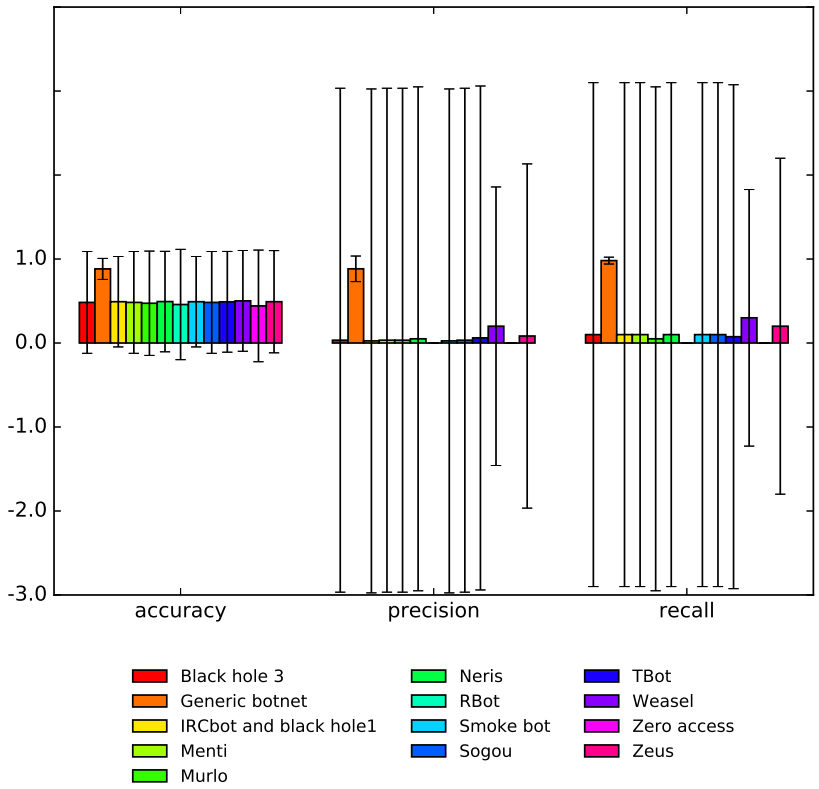

Figure A.2: Evaluation metrics for each botnet

Acronyms

AUC Area Under the Curve. 20, 22

BFF Botnet Finder Framework. 1, 5, 9, 11–18, 20, 23, 60, 71, 72, 76, 79, 80

C&C Command and Control. iii, 5–8, 64, 65, 77, 79, 80

CDN Content Delivery Network. 77

CSV Comma Separated Values. 18

DBSCAN Density-Based Spatial Alustering of Applications with Noise. 67

ELISA Elusive Social Army. 2, 3, 5, 7, 8, 63, 75–77, 79, 80

EPS Encapsulated PostScript. 23

FPR False Positive Rate. 22

HITS Hyperlink-Induced Topic Search. 66

HTTP Hypertext Transfer Protocol. 6, 71

IRC Internet Relay Chat. 6, 7, 70, 71

ISCX The Information Security Centre of Excellence. 14, 69

JSON JavaScript Object Notation. 18–20

OSN Online Social Networks. 8

P2P Peer-To-Peer. 6, 71

PPV Positive Predictive Values. 21

ROC Receiver Operating Characteristic. 20, 21

TPR True Positive Rate. 21, 22

Bibliography

[1] Alberto Compagno, Mauro Conti, Daniele Lain, Giulio Lovisotto, and Luigi Vincenzo Mancini. Boten ELISA: A novel approach for botnet C&C in Online Social Networks. *IEEE Conference on Communications and Network Security (CNS), 2015*, pages 74–82, 2015.

[2] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. DISCLOSURE: Detecting Botnet Command and Control Servers Through Large-Scale NetFlow Analysis. *28th Annual Computer Security Applications Conference*, pages 129–138, 2012.

[3] Jérôme François, Shaonan Wang, Radu State, and Thomas Engel. BotTrack: Tracking Botnets Using NetFlow and PageRank. *IFIP Networking 2011*, pages 1–14, 2011.

[4] W. Chang, A. Mohaisen, A. Wang, and S. Chen. Measuring botnets in the wild: Some new trends. *ASIACCS*, pages 645–650, 2015.

[5] S. S. C. Silva, R. M. P. Silva, R. C. G. Pinto, and R. M. Salles. Botnets: A survey. *Computer Networks*, 57(2):378–403, 2013.

[6] Benoit Claise. Cisco systems netflow services export version 9. 2004.

[7] Sheharbano Khattak, Naurin Rasheed Ramay, Kamran Riaz Khan, Affan A Syed, and Syed Ali Khayam. A taxonomy of botnet behavior,

detection, and defense. *IEEE Communications Surveys & Tutorials*, 16(2):898–924, 2014.

[8] D. Lapsley C. Livadas, R. Walsh and W. Strayer. Using machine learning techniques to identify botnet traffic. 2006.

[9] Guofei Gu, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, and Wenke Lee. Bothunter: Detecting malware infection through ids-driven dialog correlation. In *Usenix Security*, volume 7, pages 1–16, 2007.

[10] Florian Tegeler, Xiaoming Fu, Giovanni Vigna, and Christopher Kruegel. Botfinder: Finding bots in network traffic without deep packet inspection. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 349–360. ACM, 2012.

[11] Travis E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, May 2007.

[12] Angela Orebaugh, Gilbert Ramirez, Jay Beale, and Joshua Wright. *Wireshark & Ethereal Network Protocol Analyzer Toolkit*. Syngress Publishing, 2007.

[13] Elaheh Biglar Beigi, Hossein Hadian Jazi, Natalia Stakhanova, and Ali A. Ghorbani. Towards Effective Feature Selection in Machine Learning-Based Botnet Detection Approaches. *IEEE Conference on Communications and Network Security (CNS), 2014*, pages 247–255, 2014.

[14] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13:22–30, 2011.

[15] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. http://www.scipy.org/, 2001–2016.

[16] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[18] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

[19] David Powers M. W. Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation. *Journal of Machine Learning Technologies*, 2:37–63, 2011.

[20] Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27:861–874, 2006.

[21] Erich Gamma, Richard Helm, and Ralph Johnson nad John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[22] Leo Breimanq. Random forests. *Machine Learning*, 45:5–32, 2001.

[23] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial

databases with noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*.

[24] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *JACM*, 46(5):604–632, 1999.

[25] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[26] Luis von Ahn. Hubs and authorities. http://nitch.marketing/science-interwebs/, 2008.

[27] Wolfram Alpha LLC. Wolfram|alpha. http://www.wolframalpha.com/.

[28] D. Zhao, I. Traore, B. Sayed, W. Lu, S. Saad, A. Ghorbani, and D. Garant. Botnet detection based on traffic behavior analysis and flow intervals. *Computers & Security*, 2013.

[29] A. Shiravi, H. Shiravi, M. Tavallaee, and A. A. Ghorbani. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *Computers & Security*, 31(3):357—-374, 2012.