ShieldNet Classic, Neural & Ada: An omni-comprehensive and customizable framework to DoS and malicious traffic detection

Small Project in AI and Cybersecurity
by Pascoli Massimiliano
AAU id: 12138922
e-mail: mapascoli@edu.aau.at
date: 10/03/2023

Outline

This project started as a proof of concept application and ended as a complete framework to deal with DoS and malicious traffic detection coming from external sources. The framework unifies two paradigms: a classical local way with *iptables* firewall management and a more modern deep learning approach. It is highly customizable; it was originally thought as a double layer DoS traffic filter, but can be easily extended to other problematics: this requires a new deep learning model, possibly with other metrics to compute; although the infrastructure in place remains the same. The utility can be applied, and is not limited to, end hosts or routers.

The source code, as well as complementary material, is publicly available on our GitHub repository: MiciomaXD/ShieldNetAdaptor.

Introduction

Internet is both a fascinating and truly dangerous place. It is an agglomerate of more or less valuable data and information, flaws, possibilities and conflicting interests. From its birth, it appeared clear that Internet is a tool that anyone can use at low costs and potentially high gains: this is especially true when the services offered online deal with highly sensible data.

Since anyone can connect his infrastructure with the outer world and given that the number of devices connected is always increasing at an astonishing rate, it is statistically very likely to find a vulnerable service or device. Many of those operate in extremely critical scenarios, in real-time or, perhaps, with confidential data.

Attacks are surely growing in frequency and intensity because the ever increasing value of online assets is very appetizing to malicious parties. But, not only these critical nodes are targets of attacks. In fact, to an attacker, every little bit of computing capability is useful to pursue his own goals: usually the latter are even more sophisticated attacks against stronger security measures of critical nodes in order to gain financial revenue in some ways (e.g. selling data).

There are several attacks to vulnerabilities that undermine confidentiality and integrity of the asset and there are many straight forward solutions, in the majority of cases easy to implement.

To cite some NIST and NSA guidelines these countermeasures can be as easy as:

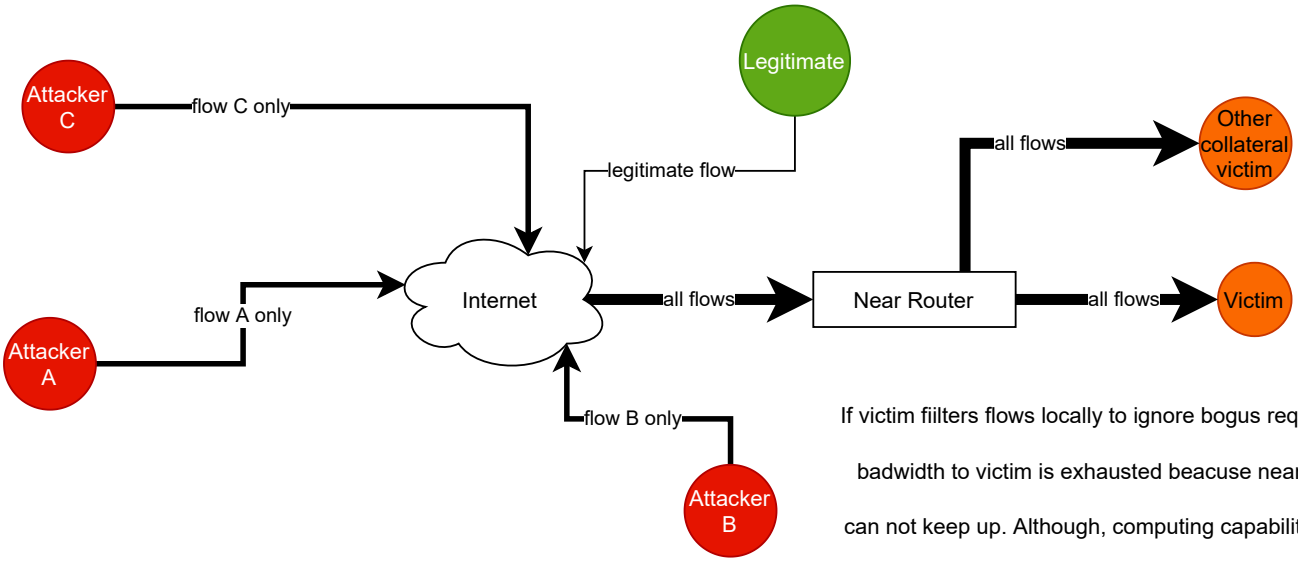
- adopting up-to-date crypto algorithms,
- stronger passwords with 2FA,
- configuration of firewalls,
- updating systems to latest releases and patches,
- segmentation of roles and responsibilities,
- utilizing only trusted and signed software,
- periodic cybersecurity assessment (IT auditing, penetration testing, cyberthreat risk management, ...)

The ones mentioned above, are pretty standard solutions, applicable to all intrusion detection, exploits and misconfiguration problems, but there are other threats that are directed to block accessibility of an assets: it can be very frustrating having the data, but no one, even the legitimate parties, can access it.

This is the case of Denial of Service (DoS) style attacks that were the main reason for this project idea. In this paper, complemented by the Python files that can be found in the repository, we will explain how the framework created, code name ShieldNet (SN in the following), works and offers a multi-layered mitigation to DoS attacks that can be extended to other cyberthreats too. The framework was created having in mind Linux family OS and iptables, but can be easily adapted to work with Windows machines (mainly path representation changes and a different firewall interaction).

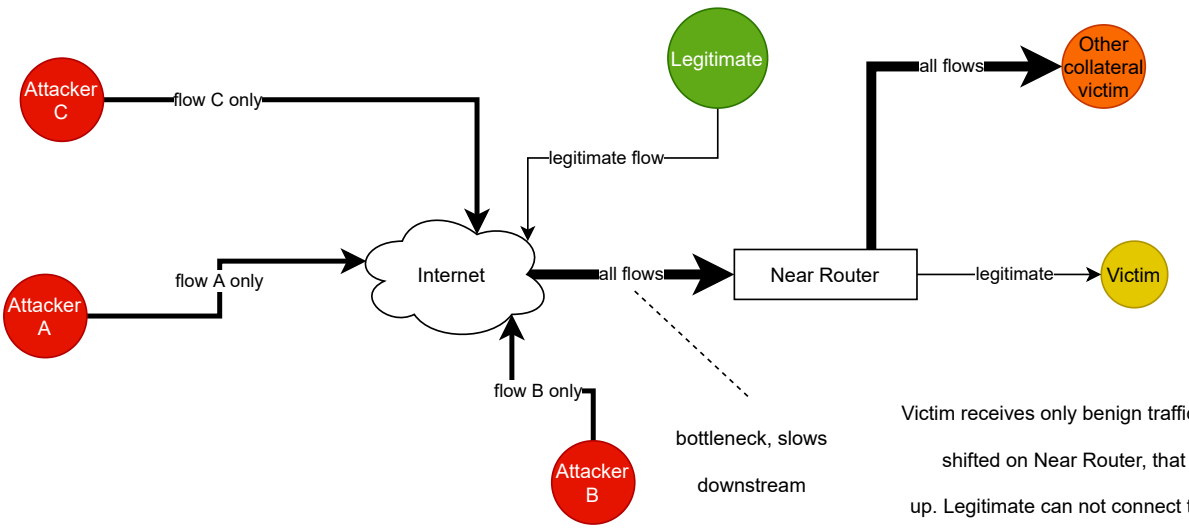
The double-layer defence by SN to be effective should be installed on routers, as upstream as possible to DoS traffic sources. Check the explanatory diagram below in this regard.

Scenario 1: victim is filtering locally

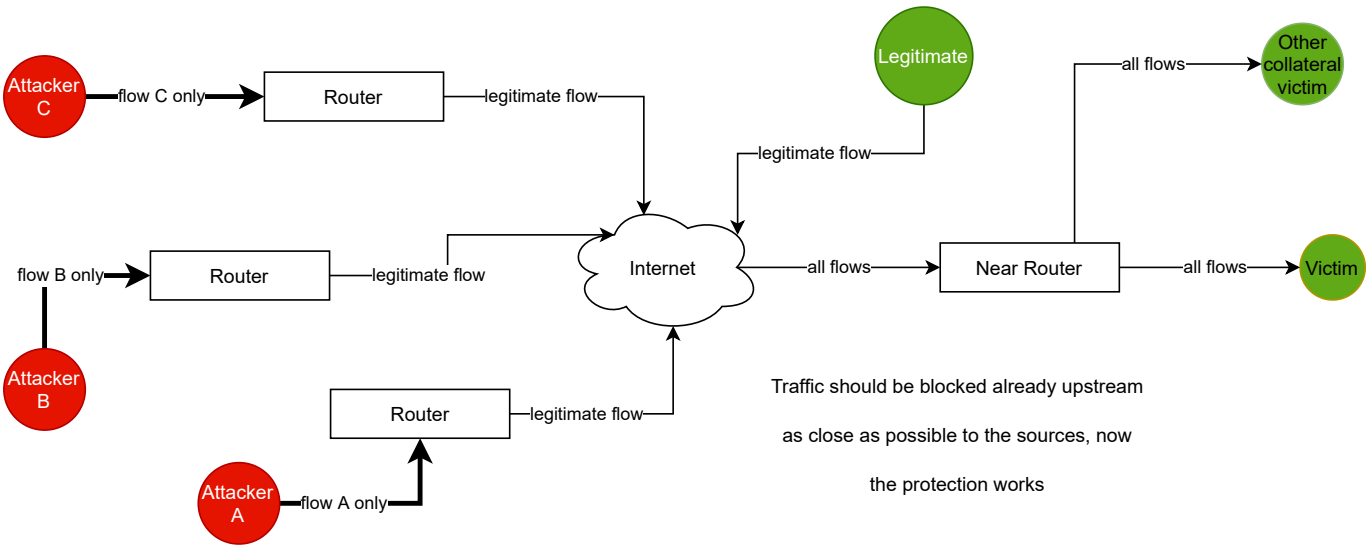


If victim fiilters flows locally to ignore bogus requests, badwidth to victim is exhausted beacuse near link can not keep up. Although, computing capabilities of victim are preserved (a local user does not experience performance degradation of his machine, but no one can connect to him)!

Scenario 2: only neighborhood is filtering



Victim receives only benign traffic, but the problem shifted on Near Router, that can not keep up. Legitimate can not connect to victim anyway.



In this paper we will cover briefly the realization of the deep learning model, since it is already presented in detail in our other work available at GitHub - MiciomaXD/MacLea_DeepLea_Project_Klagenfurt. Please, for a detailed reading regarding ShieldNet Neural (SNN) check proposal, presentation and final report produced in the scope of Alpen-Adria-Universität Klagenfurt course "Machine Learning and Deep Learning (650.025)" by professor Schekotihin Konstantin - semester 22W. This is because the core part of SNN is one of the many models produced in that setting.

We will explain in more details ShieldNet Classic (SNC - the part of ShieldNet that deals with Internet traffic without a neural network; this is done applying standard, field-tested filter rules) and ShieldNet Ada(ptor) (SNA - the part that aggregates and maintains up-to-date data for later use in SNN).

Related literature: limits, acknowledgements, techniques

First of all, what exactly is a Denial of Service (DoS) attack?

As the reader could guess from the name, in its simplest form a DoS attack is a procedure aimed to make a victim unreachable, hence unable to offer its services to legitimate parties or at least degrading performances greatly.

The way this is achieved is by exhausting victim network bandwidth or connection capabilities through bogus requests in a plethora of ways. The taxonomy of DoS and DDoS (the most common and simple, "Distributed" variant of DoS, with several attack machines and high traffic volumes e.g. SYN flood technique) style attacks is vast and classic statistical methods struggle to keep up. This is happening in a settings where organized attacks reach the overwhelming magnitude of up to 46 millions requests/s on Google Armor for a 2,54 Tbps traffic flow on 1st June 2022, just to cite one example.

To make the situation even more problematic, not all attacks use high traffic volumes and thus are easy to identify, but some of them exploit intrinsic flaws in Internet protocol / applications specifications and implementations (e.g. Slowloris technique with low traffic volume or using TCP shrinking window and time-out-based retransmission for congestion management as explained here).

Finally, unlike other attacks, DoS is technically always possible from the very first moment a machine is publicly available.

Statistical solutions

So how can someone defend or mitigate his subnetwork from these devastating DoS attacks?

Literature presents several statistical approaches with the same background idea. First, an average network behaviour model is computed, dependent on the setting we would like to defend. Then, we focus on abnormal behaviour that diverges a lot from the "average model".

The problem with this, is that the aforementioned average model is constructed on general and user-defined thresholds. With a very variable flow, it is an almost impossible task to characterize a proper model without continuously adjusting those thresholds.

Many (if not all to our knowledge) classical approaches are based around Shannon entropy and Pearson's correlation to generate a mathematically accurate representation of the network using features like IP addresses. This requires a lot of information and extensive network knowledge on its own and it is difficult to carry out in real time (Hoque et al. 2017).

Attackers can easily manipulate and forge ad-hoc traffic to bypass these countermeasures with tools such as *hping* or *HOIC* (popular successor of *LOIC* with DDoS capabilities).

Most successful methods used for DoS detection are recent machine learning methods: Support Vector Machines (SVM), k-Nearest Neighbour (kNN) and Naive Bayes Classifier (NB). As stated in Deep learning approaches for detecting DDoS attacks: a systematic review the mentioned methods are not efficient.

DL approaches

Neural networks can be the solution to DoS detection and this project wants to be yet another very practical confirmation.

Their adaptability and flexibility can be implemented real time under some constraints, especially if we account for GPU accelerated computing. In addition to that, very simple models, proposed also by us in the previous cited work, are expressive enough to capture complex attack patterns with ease.

To make a quick roundup of related work proposed by other researchers (treated in more detail in our complementary work), it is possible to distinguish two ways of proceeding:

- the first, involves the evaluation of a model for every single incoming packet. As examples we bring: CNN-Based Network Intrusion Detection against Denial-of-Service Attacks and DDoS attack detection and classification via Convolutional Neural Network (CNN). Both apply a clever packet to image conversion (bit to BW pixel or byte to greyscale pixel) and than perform classification using CNNs.

- the second, requires the a priori aggregation of Internet flow metrics and then the classification of flow statistics. As examples:

    - GLD-Net: Deep Learning to Detect DDoS Attack via Topological and Traffic Feature Fusion - PMC uses GNNs with attention. The model representation of the network is a connected undirected graph, similar to classic methods, where every node is a host. Every edge of the graph contains flow metadata (e.g. bytes/s, number of flags, up and down rate, ...) that can be checked and taken under consideration by the GNN layer to classify every flow between two nodes $\langle n_1, n_2 \rangle$. This approach is particularly interesting because GNN use a representation that is very natural and conceptually easy to read. Furthermore, to classify a flow between $\langle n_1, n_2 \rangle$ with GNNs, also adjacent pair data $\langle n_2, n_3 \rangle$ can be used to obtain a, perhaps, more accurate classification, even if node n_1 is not even aware of the existence of n_3: this is characteristic of GNNs. Context is extrapolated and summarized, up to certain extents, following a recursive criteria from the neighbourhood of a node and, then, collapsed together with the local data; a classification is finally made on the combined metrics.

    - Improved DDoS Detection Utilizing Deep Neural Networks and Feedforward Neural Networks as Autoencoder makes use of powerful autoencoder paradigm to shrink down model size, hence optimizing performance. Starting from a high number of features (a high-dimensional feature space), autoencoders are a great way to shrink down input dimensionality maintaining and selecting useful information to reproduce the input itself, starting from the summary. This gives to autoencoders the classic and recognizable hourglass shape. If it is possible to start with simplified input, training of a classifier will be easier and the classifier itself will be smaller in size, boosting efficiency during deployment.

    - Binary classification using SMOTE+LSTM | Kaggle. This is an approach to the task that has potential, followed by a user on Kaggle facing DoS detection. Another dimension that can be considered is, in fact, time. Metadata of a flow changes throughout time; how the change happens between two "snapshots" of a flow can be a distinguishing factor between a malicious or a benign connection. LSTMs are used in this setting to account for the time dimension, consuming the dataset as a series of sequences. In addition, this implementation uses Synthetic Minority Oversampling Technique to generate artificial samples where is needed to rebalance the dataset.

Every approach listed above is capable of 99+% accuracy, setting the state-of-the-art standard, reached and in some cases, exceeded, by our, even smaller, model.

The approach we followed in SNN, was the "aggregation first than classify" one. The reasons for that are mainly for time and memory efficiency. An evaluation for every packet would kill

connection throughput, so to classify every packet is not feasible. Secondly, GNNs and LSTMs requires extended network knowledge (a "master router" that knows everything is its neighbourhood) or to keep in memory more data per flow (every flow has multiple "reincarnations" that describe the trend through time).

Autoencoders, on the other hand, were used in our model production.

Installation of Shield Net

SN is a multi-threaded interactive application, that uses Linux firewalling utility "iptables" to achieve the desired filtering of the internet traffic. Furthermore, the caption of the packets (sniffing) is a very low level process that has components running in kernel space. Hence, it is necessary that the user running the scripts has root permissions when starting the application.

In addition to that, we advise to use a virtual environment created with tools like Miniconda in order to manage the required libraries installation. In addition to Python 3.10.X, SN needs:

- Pytorch for all that concerns the deep learning layer of SN (i.e. SNN);
- Pandas and correlated NumPy / SciPy stacks for internal data metrics computation and memorization;
- Scapy for packet management, decoding, sniffing, analysis;
- Pickle for storage of some components.

Once the installations are done, it is sufficient to run

```
python3 daemon.py
```

to start background packet processing. When needed, a human readable internal representation of the current state of SN can be obtained with

```
python3 command_interface.py situation
```

or SN can be safely stopped in a controlled manner with

```
python3 command_interface.py stop
```

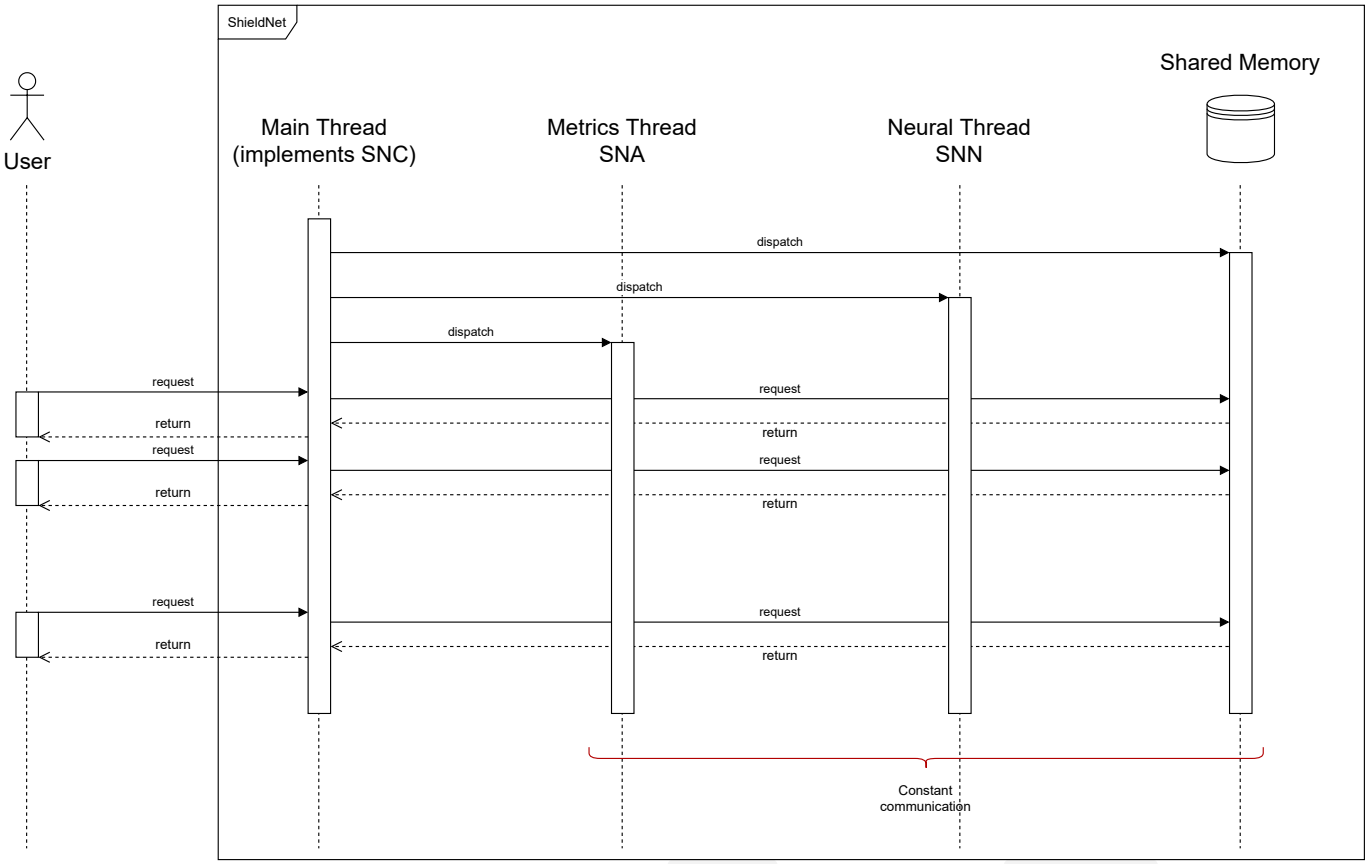
Please, note that upon stop command execution, iptables will be reset to the situation previous to the start of SN, so any preconfigured set of rules will remain untouched (SN works on custom iptables chains created specifically for it at start-up, that flank seamlessly already specified rules - more on that in the following).

ShieldNet: a panoramic of the whole infrastructure

SN is composed by three major components:

1. ShieldNet Classic - SNC;
2. ShieldNet Ada (or Metrics) - SNA;
3. ShieldNet Neural - SNN

All of them offer a great degree of customization. In the diagram is reported an overview of the organization of these three fundamental components.



The entry point of the whole application is the `main()` function of file `daemon.py`. The function, first, implements a general logger for processes and functions usable globally: the log file will contain useful information regarding the status and possible errors during the computation of *all* components of SN. At this point, the boot sequence can begin.

SNC is a static set of customizable firewall rules that are executed in sequence from `iptables` folder files: rules are read and applied to iptables. It is recommended to apply at least the basic set of rules, but the user can choose in `config.py` whatever level of protection he wishes (please refer to the ShieldNet Classic dedicated section), even disable firewall based shield completely.

After the implementation of SNC the object that represents the shared memory for the other threads is instantiated and sub-processes belonging to SNA and SNN are started. The defence is now up and running. In particular, SNA captures all packets that the network card of the

machine encounters and computes flow metrics that will be needed by SNN for classification of every flow. SNN deals with the Pytorch neural network to classify flows and blocks malicious traffic. SNN also has the job to maintain a clean register table: delete too old flow data, update register in case a flow is currently blocked, decide what flow is suitable for a classification, ...

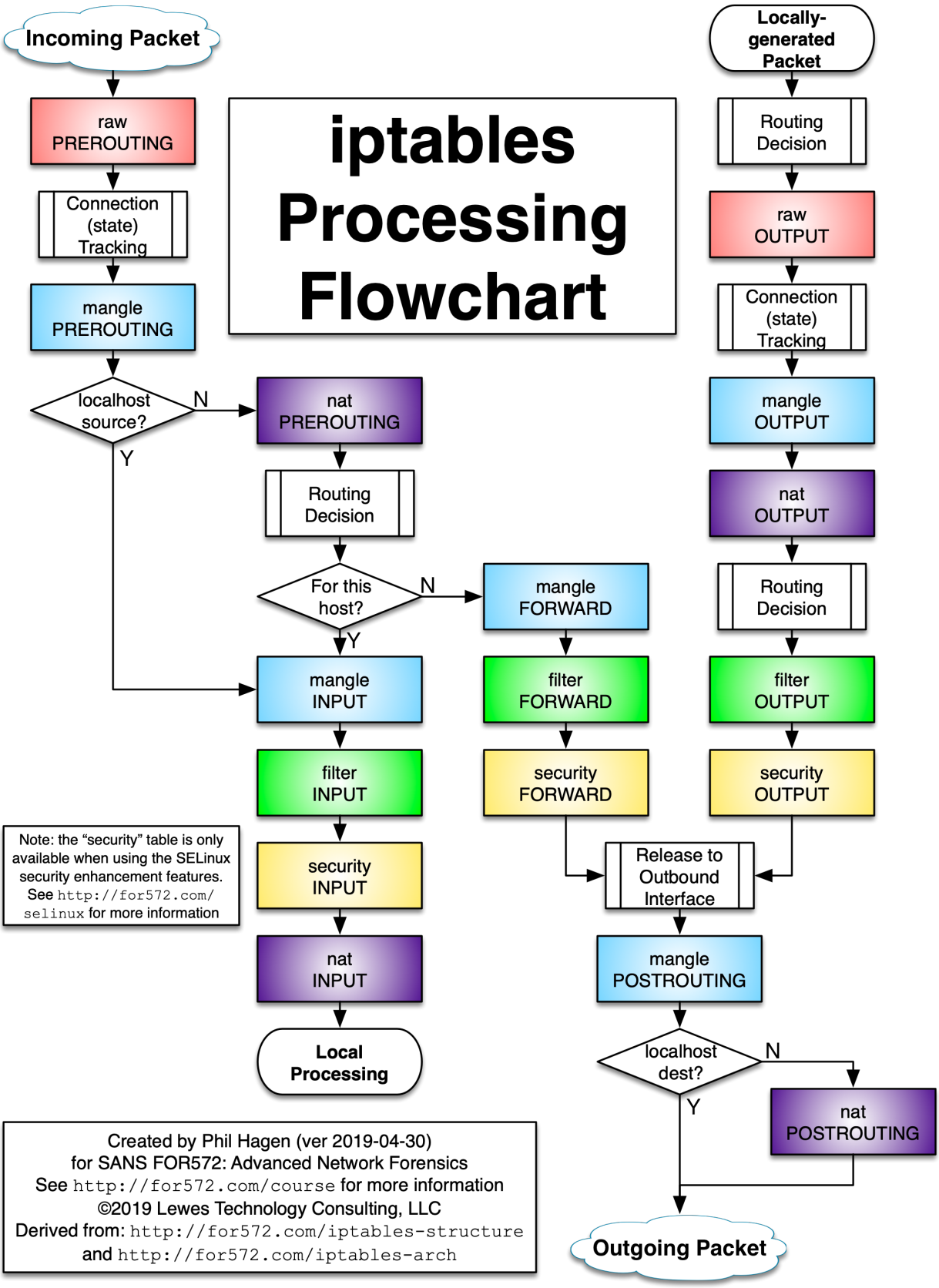
Since SN should be an interactive application, the `main()` function, after booting, has to listen for requests from a user that can execute a command in any moment (e.g. `stop`). This is done via reading one-way FIFO pipes. The file `command_interface.py` is responsible for delivering command requests to the input FIFO pipe, so that `main()` can receive, execute them and respond on an output FIFO pipe with the results.

ShieldNet Classic: classic vanilla defence and firewall connections management

Together with SN code, three set of iptables rules are provided (basic, optional and highly specialized synproxy sets). SN also comes with optimized kernel settings that are mandatory for use with synproxy due to how the packets are marked.

Basic and optional rules *are* field tested, and they are a results of common setups available publicly and personal experience (we were attacked in the past with unorganized DDoS SYN-ACK at around 1Gbps - these rules were enough, locally, to defend and make the attack ineffective). The kernel settings provided are optimized for our Contabo VPS, victim of the attack mentioned, but should work for the majority of UbuntuServer distributions.

Below, is reported the diagram that shows iptables structure. It is, perhaps, interesting to the reader to know what is our logic behind rule placement, since there are many options available (tables raw, mangle, nat, filter, security).



First of all, rules focus on TCP-based attacks: usually, UDP attacks are amplified reflection attacks (i.e. the attacker uses IP of the victim to forge false requests that appear to originate from the victim and sends them to many other servers, that respond to the victim amplifying the

traffic volume) that, due to the high volume, are not manageable locally (to prevent that, scenario 3 of the introduction would be needed, or a DDoS protected network e.g. Cloudflare).

As stated in the manual for iptables (for our goals security table is ignored):

- the filter table is the default and most commonly used table that rules go to if option -t is used;
- the nat table is used, as the name suggests, for Network Address Translation (NAT). If a packet creates a new connection, the nat table gets checked for rules;
- the mangle table is used to modify or mark packets and their header information;
- the raw table purpose is mainly to exclude certain packets from connection tracking using NOTRACK target (-j option).

Inside every table there are one or more chains:

- PREROUNTING for packets that arrive to the network card, available in raw, nat, mangle tables;
- POSTROUTING for packets that leave the network card, available in nat, mangle tables;
- INPUT for packets that have this machine as destination (local socket), available in filter, mangle;
- OUTPUT for packets that have this machine as source, generated locally, that depart to other destinations in the network, available in filter, mangle tables;
- FORWARD for packets with a destination that is different from this machine, but need routing through this machine, available in filter, mangle tables.

Depending on what kind of packets it is required to mark, modify or block, a table and a supported chain is selected.

Low level chains are the fastest ones and are processed first (first being PREROUTING of raw table), but filter table does not support this chain (we need to *filter* traffic...) : raw and mangle do. Unfortunately, raw table does not support some functionalities needed for SNC (e.g. TCP header marking), so mangle table is the best choice.

It is recommended to check out SNC rule set files because are rich in comments that explain what each rule does.

A last note on synproxy rules. Synproxy requires Linux kernel version 3.12 or above, iptables from version 1.4.21 up and custom kernel settings provided with SN.

Synproxy is an internal module of iptables that checks if a sender of a SYN packet establishes the connection for real or does nothing. It is capable to discard packets with minimal performance impact (designed for multiple millions packets/s SYNflood attacks). To explain how it works in this setting we provide a fragment of `iptables\og_iptables_rules_synproxy.txt` :

```
#excludes syn packets from connection tracking, otherwise too many resources wasted
(target CT is conntrack)
iptables -t raw -A ShieldNet -p tcp -m tcp --syn -j CT --notrack

#matches the syn packets (untracked as per previous rule) and ack packets (invalid as
per
#nf_conntrack_tcp_loose=0 kernel settings) and forwards them to the synproxy target,
which
#then verifies the syncookies and establishes
#the full TCP connections
iptables -t filter -A ShieldNet -p tcp -m tcp -m conntrack --ctstate INVALID,UNTRACKED
-j SYNPROXY --sack-perm --timestamp --wscale 7 --mss 1460

#drops every packet that the previous rule didn't catch
iptables -t filter -A ShieldNet -m conntrack --ctstate INVALID -j DROP
```

where chain `ShieldNet` is a custom chain created by SNC that is inserted into PREROUTING of table raw and filter. We can use filter table because we exclude SYN packets from connection tracking and mark ACK packets as invalid (kernel custom setting) to spare some resources. Then redirect packets with states (option --ctstate) INVALID/UNTRACKED to target SYNPROXY efficient chain, instead of PREROUTING one (the latter not supported by filter table).

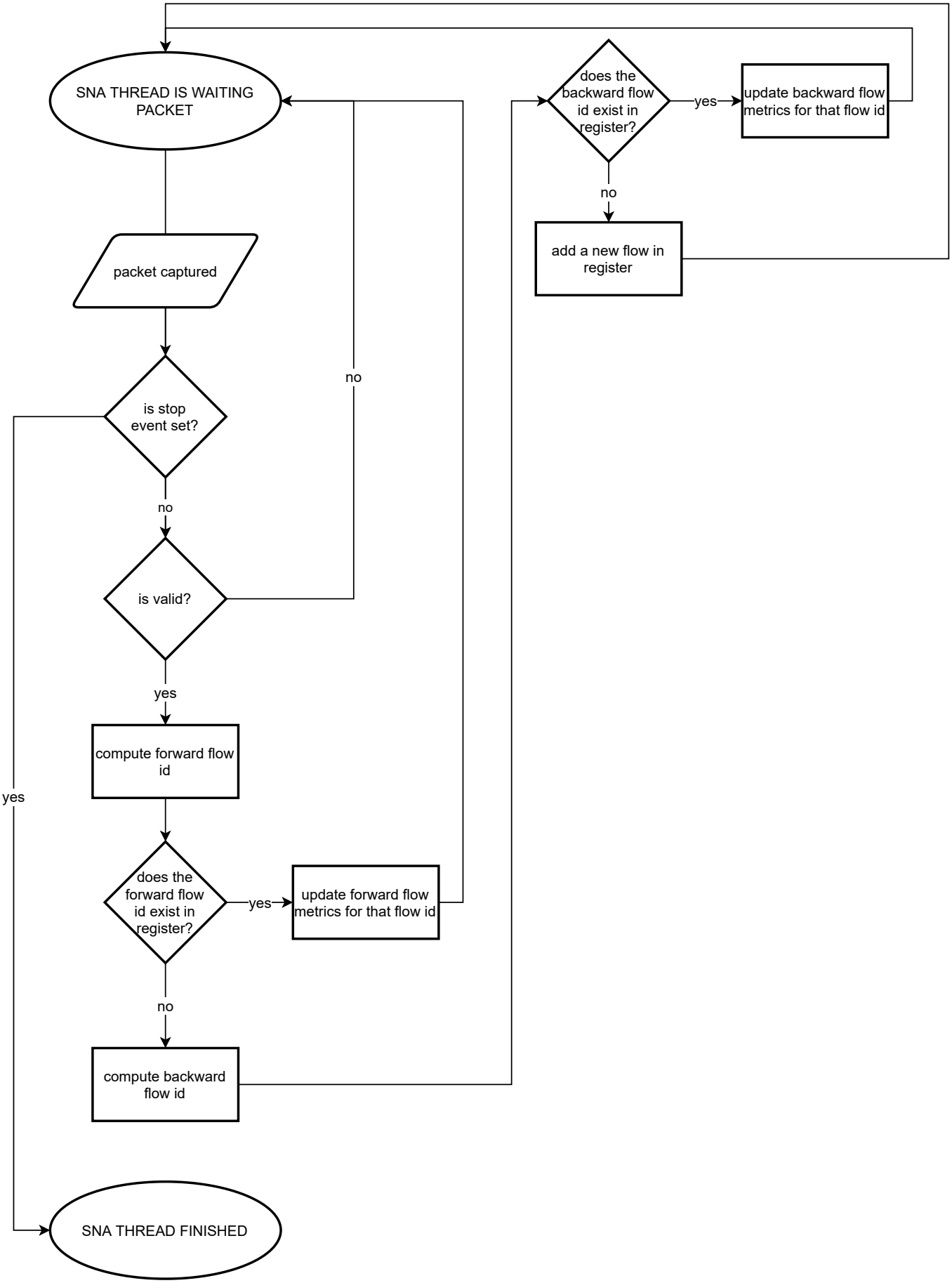
ShieldNet Neural and ShieldNet Ada: deep learning with built-in metrics computation and firewall connections management

To our knowledge, at the time writing this paper, the literature is very well documented and rich in traffic flow classification starting from offline data, but lacks real-time examples. The prerequisite of SN *is* to work in real-time, this means it was necessary to overcome some technical difficulties in the computation of the aggregated *flow* metrics (one flow = several packets) on which SNN works on, knowing only one new packet at every instant.

ShieldNet Ada

SNA is the segment of SN that makes sure SNN has correct data to classify at every moment. The model, discussed in the next section, needs 20 numerical flow features in order to classify traffic.

Below, is reported the flow diagram that explains how SNA self-clocks itself using every captured packet:


SNA THREAD IS WAITING PACKET
packet captured
is stop event set?
no
is valid?
no
yes
yes
compute forward flow id
does the forward flow id exist in register?
yes
update forward flow metrics for that flow id
no
compute backward flow id
does the backward flow id exist in register?
yes
update backward flow metrics for that flow id
no
add a new flow in register
SNA THREAD FINISHED

To understand the next paragraph, it is necessary to introduce a proper definition for the flow id. The flow id, mathematically, is an ordered tuple formed by $\langle IP_1, IP_1, Port_1, Port_2, Protocol \rangle$. Programmatically, it was treated as a string formed by the concatenation of all tuple components interleaved with a dash.

Starting from a register configuration, containing data computed from every past packet there are three cases that can happen when a valid (UDP or TCP packet) is captured:

1. the *forward* flow id exists already in the register
2. the *backward* flow id exists already in the register
3. the packet belongs to a never encountered flow

Case 3 is the case that defines flow direction, in fact a packet that does not fit in other flows (a new flow must be created) is automatically considered as a forward packet: this translates into IP_1 and $Port_1$ being the source IP address and source port, IP_2 and $Port_2$ the destination values respectively, as are found in the IP payload fields of the packet itself.

Case 2 happens when a packet has the roles of $IP_1, Port_1$ and $IP_2, Port_2$ exchanged. Instead of writing a new flow in the register $\langle IP_2, IP_1, Port_2, Port_1, Protocol \rangle$, it is checked if the inverse flow id is already present. This let us know that the flow exists, but the first packet encountered had the other direction (thus the current is a backward packet; only backward metrics will be updated).

Updating of the 20 numerical flow feature is fortunately easy: in the majority of them a sum, minimum or maximum are sufficient. The not trivial task is to update average and standard deviation having one new packet (which is directional) and the old value of average and standard deviation. This is feasible, after some rewriting of classic formulae, one can obtain:

Classical average computation

$$\overline{X}_n = \frac{1}{n} \sum_{i=1}^{n} x_i$$

One-pass recursive calculation

$$\begin{cases} \overline{X}_0 = 0 \\ \overline{X}_{n+1} = \frac{\overline{X}_n \cdot n + x_{n+1}}{n+1} \end{cases}$$

Classical population standard deviation computation

$$\sigma_n = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \overline{X}_n)^2}$$

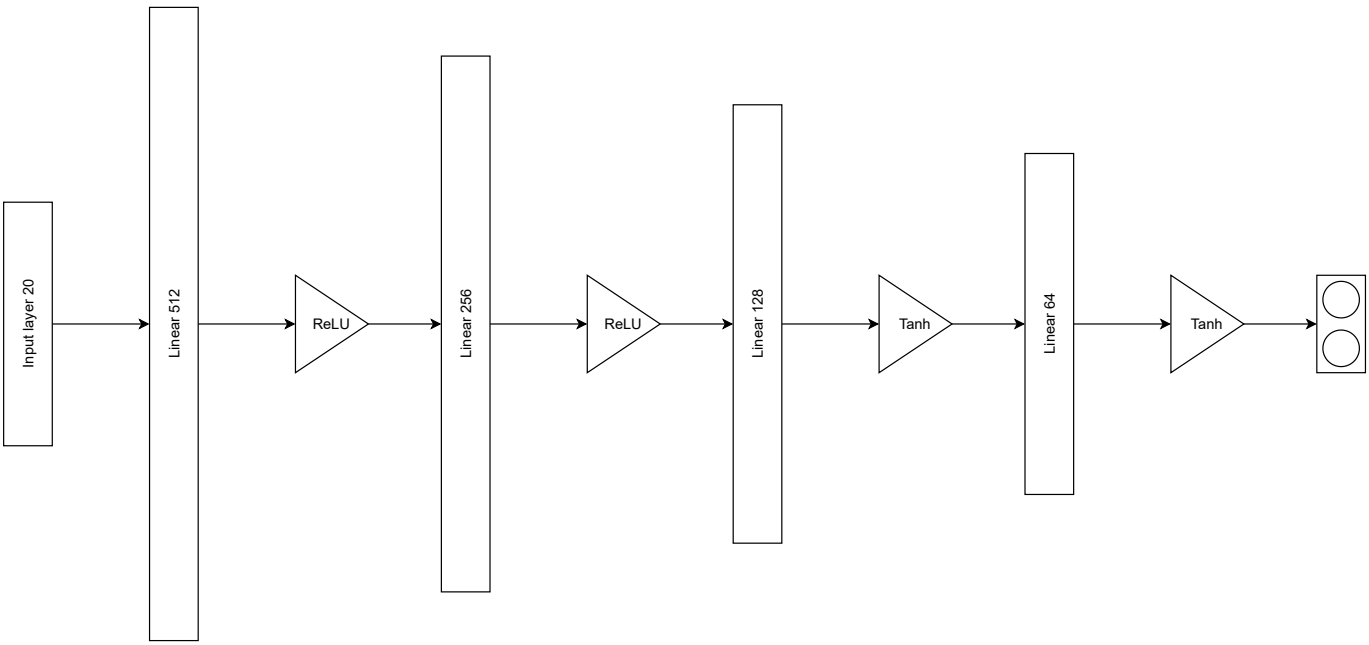
One-pass recursive calculation

$$\begin{cases} \sigma_0 = 0 \\ \sigma_{n+1} = \sqrt{\frac{n}{n+1} \cdot \sigma_n^2 + \frac{1}{n} \cdot (x_{n+1} - \overline{X}_{n+1})^2} \end{cases}$$

ShieldNet Neural

The neural network model used in this first version of SN was produced in the aforementioned complementary work. In the following, are provided only some key features of the model so that the reader can understand how such neural network was trained, what shape it has, on what kind of data was trained on.

Model shape



In the image above is shown the shape of the model. It is a sequence of linear layers, interleaved by activation functions Rectified Linear Units (ReLU) and hyperbolic Tangent (Tanh) with logarithmic Softmax at the end. Logarithmic Softmax or LogSoftmax is numerically more stable than the non-logarithmic counterpart and is defined as:

$$LogSoftmax(x_i) = \log\left(\frac{e^{x_i}}{\sum_j^n e^{x_j}}\right)$$

where x_i and x_j are components of an n-dimensional tensor

(Log)Softmax is useful in classification problems to obtain a probability distribution over the classes.

Summary of training settings and hyperparameters

Model name: `.\models\12-21-2022_15-54-25__anova_binary_opt_sched.model`

- dataset DDoS Dataset | Kaggle (`ddos_balanced/final_dataset.csv` only) with around 80 flow features, reduced to 20 using ANOVA F-value test (were kept the best 20 scores), ANOVA is computed as shown below;

$$F\_score = \frac{\text{variation between sample means (MSB)}}{\text{variation within the samples (MSW)}}$$

$$MSB = \frac{\text{Sum of squares between the group (SSB)}}{DFb}$$

where

$SSB = \sum (X_i - X_t)^2$ with X_i mean of group i and X_t mean of all the observations

$DFb =$ degrees of freedom between $=$ total number of observations in all the groups -1

$$MSW = \frac{\text{Sum of squares within the group (SSW)}}{DFw}$$

where

$DFw =$ degrees of freedom within $= N - K$

with K number of groups and N total number of observations in all the groups

$SSW = \sum (X_{i,j} - X_j)^2$ with $X_{i,j}$ is the observation of each group j

NOTE: An observation is not a sample of the dataset: 1 dataset row with q features has q observations! So N is number of samples $*$ number of features (groups) and K is number of features

- batch size = 100;
- `torch.nn.init.xavier_uniform_` weight initialization;
- 50 epochs;
- learning rate = 0.001;
- *Adam* optimizer with *CrossEntropyLoss* defined as below;

$$L_{cross-entropy} = -\sum_{i=1}^{n} t_i \cdot \log(p_i)$$

where t_i is the ground truth label, n number of classes and p_i is the softmax probability for the i^{th} class

- learning rate optimizer `StepLR` with step_size = 10, gamma = 0.1;
- a rudimentary early stopping mechanism

Performance on test set

```
Precision:  tensor(0.9995)
Recall:  tensor(0.9984)
F1-score:  tensor(0.9989)
Overall Accuracy:  tensor(0.9989)
Per class accuracy:  [Benign 0.9994804904170463, DoS 0.998358450377156]
```

Other information

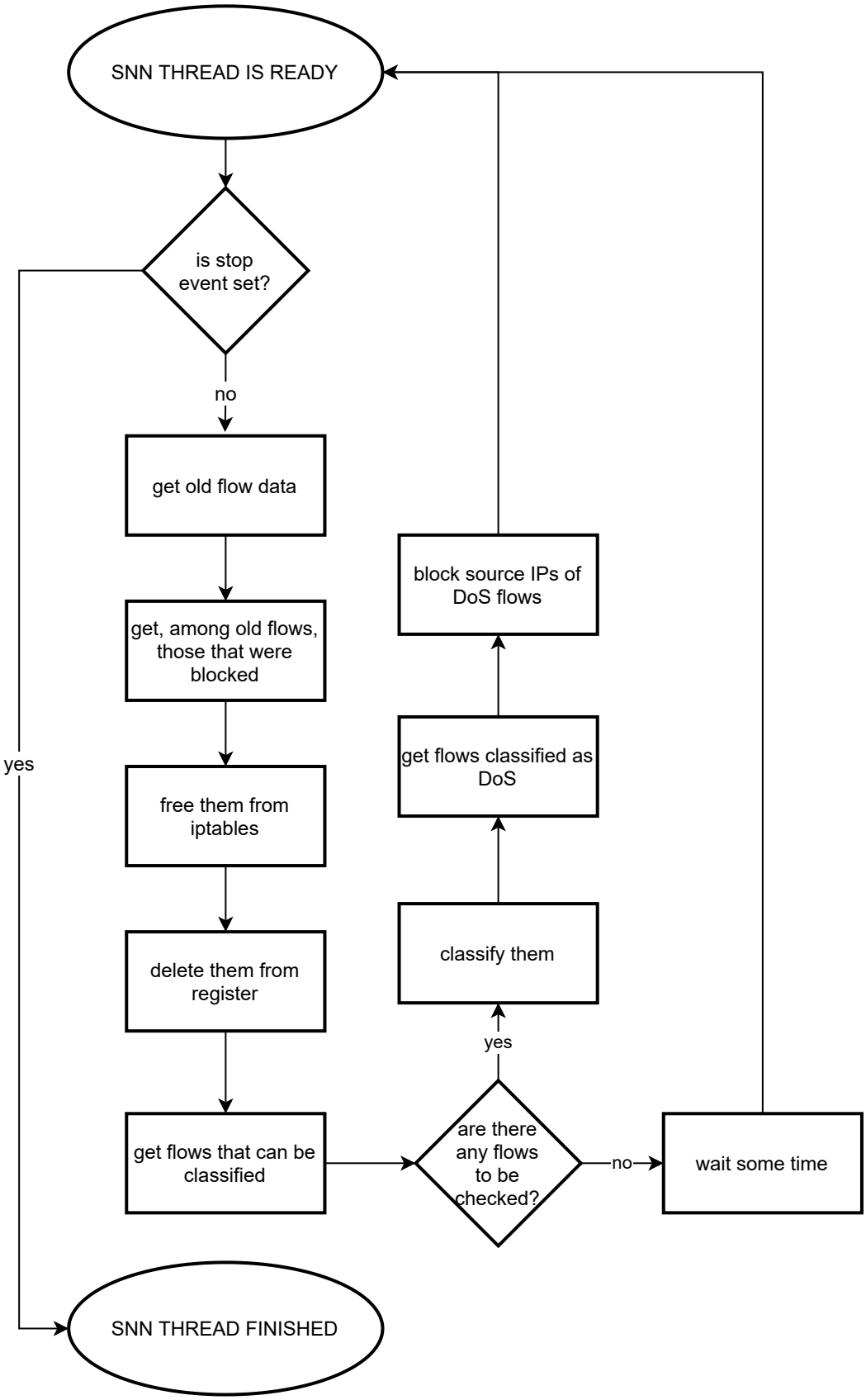
A standard scaler defined as shown below was fit on training input data and then used first on training data, then on the test set.

$$\forall i,j \left(x\_scaled_{i,j} = \frac{x\_original_{i,j} - \mu_i}{\sigma_i} \right)$$

where i is a feature and j is a sample

The scaler was saved after fitting as a pickle dump, so that SN can reload it. Before classifying a flow, all 20 features are rescaled accordingly using the same scaler.

As done for SNA, below is reported the flow diagram that explains what SNN does in order to classify internet flows present in the registers.



Deciding when to classify a flow, when a flow is old and handling of potentially conflicting classification outcomes from the neural model for the same flow in different moments is difficult. Depending on application cases, different heuristics can change the structure of SNN completely.

For this first version of SN we adopted these rudimentary and general heuristics:

- old flow: a flow checked at least once and, if it was blocked, it stayed blocked more than JAIL_TIME_SEC ago, else it is older than OLD_AFTER_SEC (last packet seen timestamp);
- classifiable flow: a flow that was never blocked and at least 20 forward packets were captured for that flow, a flow never checked or checked more than t_delta_react time ago;
- if, at any time, a flow is detected as DoS, the source IP address remains blocked for `JAIL_TIME_SEC` and will never be classified again.

Results and future steps

After 10 hours of testing on real internet traffic on our VPS server, over 5000 different flows were registered and processed. Out of them, 129 were classified as malign. These 129 flows were probably not DoS, since everything was working well during the period. 5000 flows in 10 hours are not many, but we consider the main goals of this proof of concept project reached, nonetheless.

In this project we explored the possibilities of DoS detection and mitigation, realizing a working framework that would allow, not only big corporations, but also common network users like *we* are in the first place, to defend our online projects against this and other cyberthreats. We also showed that, at least local, DoS protection is feasible at a low price.

One issue with neural networks is that they always classify something wrong (as seen in 10 hours of real world testing time), no matter the performances on the test set, since they deal with probabilities: this is due to the fact that training data is partial data, in fact it does not (and would be impossible to) include all possible samples. The misclassification of a connection has repercussions also on ethical and legal sectors of computing. What happens if a connection to a critical service (e.g. an hospital service) was not possible because SN classifies the flow of packets wrongly? Consequences could be surely devastating.

The project is far from complete! Our future steps include:

- the realization of a simple to use GUI to monitor the situation internal to SN;
- polishing of python code, making it extendible, more comprehensible and portable, so that the utility is easily installed and simply works well;
- more accurate heuristics for when to classify a flow, when a flow becomes old and what to do in case of potentially conflicting classification outcomes from the neural model for the same flow id in different moments;
- porting of SN to other, more efficient, programming languages (maybe C language): this is not only to gain from the performance point of view, but also to make SN installable in a capillary manner on every network node in order to make the protection effective (see the 3 scenario diagrams at the beginning);

- testing in a real world environment, with more diverse data and attacks;
- extensions of SNN core to detection of other, non primarily DoS attacks (e.g. brute forcing, infiltration attempts, ...).

Traditional methods remains, and services as Cloudflare offers (although not for free) already a great protection against DoS, but we hope this project is a starting point or at least inspiring to those who want to expand on the topics presented.

Finally, we would like to thank professor Schekotihin Konstantin for the useful advices provided during the creation of ShieldNet Neural.