ランダムフォレストを用いたファイアウォールの構築とファイアウォールの仮想環境化

木下研究室

201503919 丸藤 信哉

Ver.1.00

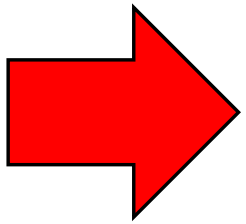
背景

ネットワークの現状とセキュリティシステム

- サイバー犯罪の増加により、対策が必要な状況にある。
- その対策としてセキュリティシステムの開発・研究が進んでいる。
- その中にファイアウォールがある。
- あらかじめ設定された条件にしたがってパケットの通過と破棄を行い、不正な通信の遮断を実現する「アクセス制御」の一種。

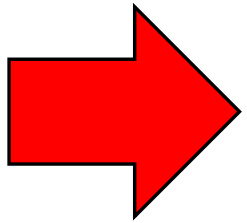
先行研究

佐野宗一朗，“機械学習を用いたフィルタリングルールの最適化”，神奈川大学2017年度卒業論文



機械学習のナイーブベイズを使用し、確率的性質からアクセスコントロールを提案

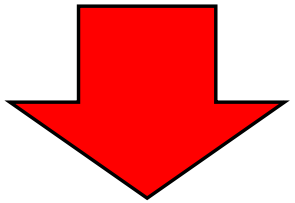
松尾達郎，“不正侵入パケットに対する機械学習の適用”,高知工科大学　平成25年度　学士学位論文



異常検知型侵入検知システム(IDS)をELMで構成することで、精度と学習時間短縮を両立

問題点

- 手動設定が一般的であるので、設定者が意図せずルール記述を忘れる可能性もある。



ポリシーを満たすことができなくなる。

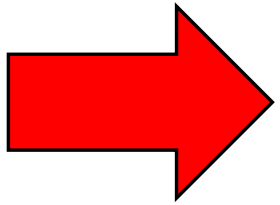
- 先行研究のナイーブベイズはあらかじめクラスタを決定しなければならない。
- これをそのままファイアウォールのルールに使用するには限界がある。

目的

- 教師あり機械学習のランダムフォレストを新たに導入し、確率の機械学習における異常データの出力の可能性を軽減する。
- ランダムフォレストによりフィルタリングルール作成の自動化を行い、利用者のポリシーの実現をする。
- 作成したファイアウォールを簡単に利用できるようにするため、これを仮想環境化し、複数の企業・組織で利用できるようにする。

提案

- 教師あり機械学習のランダムフォレストを、ファイアウォールの新システムとして導入する。
- このファイアウォールをコンテナ型仮想環境”Docker”で構築する。
- ファイアウォールを入れた複数のDockerコンテナを、Kubernetes(k8s)でデプロイ制御・管理する。

New Firewall System  Virtual Environment

実装環境

- **使用ハードウェア**: NUC7i7BNH(Core™ i7-7567U)
  - 2Cores, 4Threads, 3.50GHz(up to 4.00GHz)
  - RAM: DDR4-2133 16GBx2 32GB (31.3GB)
- OS: Ubuntu 18.04 LTS
- **使用言語**: Python3.6.8 | Anaconda 1.6.14
- **その他使用したソフトウェア**
  - Jupyter Notebook 5.7.4
  - iptables v1.6.1
  - python-iptables 0.13.0

提案手法 – ファイアウォール図解

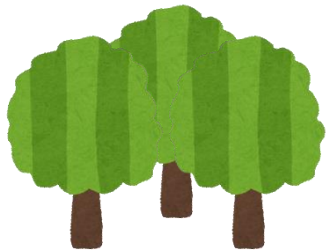
KDD Cup 99 dataset
+ IPv4 address

Pre-processing

Training data + Test data

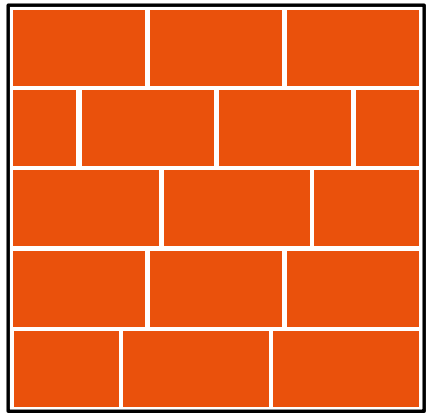
Split data 9:1

Random Forest



Predict
attack or not

Firewall



Python-iptables

Add Firewall

attack IPs

IPv4

192.168.142.8
192.168.251.36
192.168.59.151
⋮

提案手法 – データセット解説

- KDD Cup 99 dataset(71.4MB) フルセットの10%データ(494,021件)
- 訓練データ 444,618件
- テストデータ 49,403件
- IPアドレスは正規分布に基づいて生成。
- 正規分布のパターンを9パターン用意する。パターンは右の表のとおり。
- 末尾8ビットが0，255は除外する。

No.	Center IP	Range
1	192.168.128.0	192.168.0.1 - 192.168.255.255
2	192.168.128.0	192.168.64.1 - 192.168.191.255
3	192.168.128.0	192.168.96.1 - 192.168.159.255
4	192.168.64.0	192.168.0.1 - 192.168.255.255
5	192.168.64.0	192.168.0.1 - 192.168.127.255
6	192.168.64.0	192.168.32.1 - 192.168.95.255
7	192.168.192.0	192.168.0.1 - 192.168.255.255
8	192.168.192.0	192.168.128.1 - 192.168.255.255
9	192.168.192.0	192.168.160.1 - 192.168.223.255

提案手法 – データの取り扱い

- データセットは通常の通信および攻撃のラベルは分離。
  ⇒これを教師とする。また、これを数値に置き換える。
- ランダムフォレストによる予測結果は元の分離したラベルと比較するためcsvファイルへ出力。
- csvを参照し実際の結果を確認する。
- 出力したcsvファイルはpython-iptablesで使用する。
- python-iptablesの使用にはroot権限が必要。

Label name	Replaced number	Label name	Replaced number
normal	0	buffer_overflow	12
smurf	1	land	13
neptune	2	warezmaster	14
back	3	imap	15
satan	4	rootkit	16
ipsweep	5	loadmodule	17
portsweep	6	ftp_write	18
warezclient	7	multihop	19
teardrop	8	phf	20
pod	9	perl	21
nmap	10	spy	22
guess_passwd	11		

元のラベル名と数値置換の対応表

実装 – IPアドレス生成と前処理

- データ前処理を実施。正規分布に基づくIPアドレス生成およびKDD Cup 99 datasetとの結合とラベルの分離を行う。
- 右上に示すのが正規分布に基づくIPアドレス生成のソースコード、右下に示すのがKDD Cup 99 datasetとの結合とラベルの分離を行うソースコードである。

```
import numpy
import socket
import struct
import re
import csv
import pandas
from time import time
def ip2int(addr):
    return struct.unpack("!I", socket.inet_aton(addr))[0]
def int2ip(addr):
    return socket.inet_ntoa(struct.pack("!I", addr))

kdd_cup_99_10p_ipv4 = []
roop_i = 0
t0 = time()
while True:
    float2int = round(numpy.random.normal(loc=32768, scale=32767)) + 3232235520
    if float2int <= 3232235520 or float2int >= 3232301055:
        continue
    changeIpv4 = int2ip(float2int)
    if re.match("192¥.¥d{1,3}¥.¥d{1,3}¥.255", changeIpv4) is None:
        if re.match("192¥.¥d{1,3}¥.¥d{1,3}¥.0", changeIpv4) is None:
            kdd_cup_99_10p_ipv4.append(float2int)
            if roop_i == 494020:
                break
            else:
                roop_i+=1
        else:
            print("Last 8bit '0' hit, continue.")
            continue
    else:
        print("Last 8bit '255' hit, continue.")
        continue
#--inf roop end--##
kdd_cup_99_10p_ipv4 = pandas.DataFrame(data=kdd_cup_99_10p_ipv4)
kdd_cup_99_10p_ipv4.to_csv('study_logs/kdd99_ipv4plus_rf/kdd99_ipv4plus_normal_dist_data/ipv4_nd_data0.csv',
                           index=False)
tt = time() - t0
print("Process is end. Making time: {0} seconds.".format(round(tt, 3)))
```

```
import ipaddress
import pandas
from time import time
import socket
import struct
import math
#--function settings--#
def ip2int(addr):
    return struct.unpack("!I", socket.inet_aton(addr))[0]
def int2ip(addr):
    return socket.inet_ntoa(struct.pack("!I", addr))
#--function setting end--#
col_names1 = ["duration","protocol_type","service","flag","src_bytes",
    "dst_bytes","land","wrong_fragment","urgent","hot","num_failed_logins",
    "logged_in","num_compromised","root_shell","su_attempted","num_root",
    "num_file_creations","num_shells","num_access_files","num_outbound_cmds",
    "is_host_login","is_guest_login","count","srv_count","serror_rate",
    "srv_serror_rate","rerror_rate","srv_rerror_rate","same_srv_rate",
    "diff_srv_rate","srv_diff_host_rate","dst_host_count","dst_host_srv_count",
    "dst_host_same_srv_rate","dst_host_diff_srv_rate","dst_host_same_src_port_rate",
    "dst_host_srv_diff_host_rate","dst_host_serror_rate","dst_host_srv_serror_rate",
    "dst_host_rerror_rate","dst_host_srv_rerror_rate","label"]
col_names2 = ["IPv4"]

use_data1 = pandas.read_csv("KDD99dataset/kddup_data_10percent/kddcup.data_10_percent_corrected",
                            header=None, names=col_names1)
use_data2 = pandas.read_csv("study_logs/kdd99_ipv4plus_rf/kdd99_ipv4plus_normal_dist_data/ipv4_nd_data1.csv",
                            header=None, names=col_names2)

use_data_y = pandas.DataFrame(data=use_data1.label, columns=['label'])
use_data_y = use_data_y.replace({'normal.' : 0, 'smurf.' : 1, 'neptune.' : 2, 'back.' : 3, 'satan.' : 4, 'ipsw
use_data1 = use_data1.drop(['protocol_type','service','flag','label'],axis=1)
use_data_x = pandas.concat([use_data2, use_data1], axis=1)
print("use_data_x is:")
print(use_data_x)
print("use_data_y is:")
print(use_data_y)
```

実装 – IPアドレス生成と前処理

- データセットを分割。訓練データとテストデータを9:1で分割する。右上がソースコードである。
- ランダムフォレストによる訓練および予測と、予測結果の出力を行う。右中と右下がソースコードである。

```
from sklearn.model_selection import train_test_split
t0 = time()
tr_x, te_x, tr_y, te_y = train_test_split(
    use_data_x, use_data_y, test_size=0.1)
tt = time() - t0
print("Split time: {0} sec.".format(round(tt, 3)))
```

```
import numpy as np
import statsmodels.api as sm
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
t0 = time()
rf=RandomForestClassifier(n_estimators=1000, min_samples_split=2)
rf.fit(tr_x,tr_y)
tt = time() - t0
print("Training time: {0} sec.".format(round(tt, 3)))
#print('Training score check: {0}'.format(rf.score(tr_x, tr_y)))
print('The influential data is:')
for i in range(len(rf.feature_importances_)):
    print('{0:.6}'.format(rf.feature_importances_[i]))
```

```
predict_columns = ['predict_label']
t0 = time()
predict_result = pandas.DataFrame(data=rf.predict(te_x), columns=predict_columns)
tt = time() - t0
print('Predict time: {0} sec.'.format(round(tt, 3)))
print('Predict result: {0}'.format(rf.score(te_x, te_y)))
```

実装 – 予測結果の誤判定確認とcsv出力

- ランダムフォレストの予測結果を事前に分離した結果と照合し、誤判定した部分を確認する。判定後の結果を予測結果とともにcsvに出力する。右が判定、下がcsv出力のソースコードである。

```
te_y_sort = pandas.DataFrame(te_y)
te_y_sort = te_y_sort.reset_index(drop=True)
trs_ipv4 = []
te_x = te_x.reset_index(drop=True)
te_x_ipv4 = pandas.DataFrame(data=te_x['IPv4'], columns=['IPv4'])
for roop_i in range(len(te_y)):
    trs_ipv4.append(int2ip(te_x_ipv4.IPv4[roop_i]))
#----for roop end----#
trs_ipv4 = pandas.DataFrame(data=trs_ipv4, columns=['IPv4'])
label_sort = pandas.concat([trs_ipv4, predict_result, te_y_sort], axis=1)
label_sort['judge'] = 0
for i in range(len(predict_result)):
    if label_sort['predict_label'].iloc[i] == label_sort['label'].iloc[i]:
        label_sort['judge'].iloc[i] = 0
    else:
        label_sort['judge'].iloc[i] = 1
#----for roop block end----#
label_sort
```

```
label_sort.to_csv('study_logs/kdd99_ipv4plus_rf/kdd99_ipv4plus_normal_dist_data/ipv4_nd_data1ru_result.csv', index=False)
```

実装 – python-iptablesによる自動登録

- csv出力した判定結果で攻撃と予測した結果をpython-iptablesを用いてiptablesに登録する。
- python-iptablesは使用時はiptcという名称である。
- 右上はiptcを使いやすくスクリプトファイル化・モジュールとしたソースコード、右下が実際に自動登録を行うソースコードである。

iptc_cst.py

```
#!/usr/bin/env python

#This script file is "import ipct_rf"
#You need root permission(sudo) or login as su.

import iptc
import ipaddress
import socket
import struct

def ip2int(addr):
    return struct.unpack("!I", socket.inet_aton(addr))[0]
def int2ip(addr):
    return socket.inet_ntoa(struct.pack("!I", addr))
def set4iptables(ipv4):
    in_rule = iptc.Rule()
    in_rule.src = ipv4
    in_rule.target = iptc.Target(in_rule, 'DROP')
    input_chain = iptc.Chain(iptc.Table('filter'), 'INPUT')
    input_chain.insert_rule(in_rule)
```

py_iptc_test.py

```
#!/usr/bin/env python

#This .py script file is test script, for python-iptables.
#So this script should be corrected in use in real server.

import iptc_cst
import pandas
from time import time

t_array01 = []
ipv4_array = pandas.read_csv('/home/marufuji_shinya/study_logs/
kdd99_ipv4plus_rf/kdd99_ipv4plus_normal_dist_data/
ipv4_nd_data1ru_result.csv', header=0)
ipv4_array = ipv4_array.drop(['label','judge'], axis=1)
t0 = time()
for roop_result_len in range(len(ipv4_array)):
    if not ipv4_array.predict_label[roop_result_len] == 0:
        temp_memory = ipv4_array.IPv4[roop_result_len]
        inf_roop_i = 0
        while True:
            if len(t_array01) == 0:
                iptc_cst.set4iptables(temp_memory)
                t_array01.append(temp_memory)
                break
            elif temp_memory == t_array01[inf_roop_i]:
                break
            elif inf_roop_i+1 == len(t_array01):
                iptc_cst.set4iptables(temp_memory)
                t_array01.append(temp_memory)
                break
            else:
                inf_roop_i+=1
tt = time() - t0
print("The required time all IP was registered to iptables: {0}
sec.".format(round(tt, 3)))
```

結果 – 各種生成・処理完了時間と精度

- IPアドレス生成、データ分割、ランダムフォレストの訓練・予測に要した時間を下表に示す。

Data No.	Make IPv4[s]	Split data[s]	Training[s]	Predict[s]
1	12.324	0.208	212.887	5.416
2	11.399	0.228	210.748	5.264
3	11.020	0.271	210.088	5.511
4	13.640	0.256	207.332	5.413
5	12.449	0.258	211.064	5.321
6	12.198	0.238	206.833	5.309
7	13.991	0.236	212.855	5.306
8	12.309	0.195	207.531	5.303
9	11.784	0.252	215.423	5.265

結果 – 精度と実際の予測結果の確認

- ランダムフォレストの精度と実際の予測結果を確認する。
  結果は一部のみ掲載する。

Data No.	Accuracy[%]
1	99.968
2	99.947
3	99.980
4	99.978
5	99.976
6	99.980
7	99.962
8	99.970
9	99.976

Data No.2

	A	B	C	D
1	IPv4	predict	label	judge
1989	192.168.88.12	0	17	1
2268	192.168.59.246	12	19	1
7190	192.168.116.56	0	4	1
7668	192.168.240.158	0	10	1
11688	192.168.97.216	0	1	1
12475	192.168.119.52	7	0	1
13025	192.168.181.132	0	17	1
16893	192.168.193.98	0	10	1
17398	192.168.99.133	0	7	1
17921	192.168.168.36	0	10	1
18928	192.168.87.141	2	15	1
21581	192.168.68.203	12	17	1
24170	192.168.115.114	0	5	1
24689	192.168.106.201	0	4	1
25986	192.168.59.203	0	10	1
29285	192.168.106.184	0	7	1
30623	192.168.152.70	0	13	1
30850	192.168.193.238	2	13	1
38625	192.168.190.1	0	4	1
38703	192.168.192.1	0	11	1
40166	192.168.251.185	0	11	1
41146	192.168.164.22	0	4	1
44983	192.168.145.236	5	6	1
47297	192.168.60.227	0	14	1
48681	192.168.94.147	0	16	1
49370	192.168.111.249	2	0	1

Data No.5

	A	B	C	D
1	IPv4	predict	label	judge
14114	192.168.181.54	0	4	1
17719	192.168.217.184	15	2	1
24268	192.168.90.130	0	16	1
25186	192.168.42.170	0	22	1
25237	192.168.50.239	0	12	1
27117	192.168.33.8	0	12	1
29249	192.168.82.224	12	17	1
35493	192.168.50.28	7	0	1
35620	192.168.54.129	0	10	1
35901	192.168.92.44	7	0	1
37228	192.168.94.39	0	16	1
42621	192.168.160.63	2	13	1

Data No.6

	A	B	C	D
1	IPv4	predict	label	judge
1648	192.168.113.14	0	15	1
4959	192.168.38.103	0	4	1
6342	192.168.80.191	0	20	1
10167	192.168.69.99	0	6	1
13759	192.168.70.245	0	18	1
22157	192.168.97.226	5	0	1
22351	192.168.95.197	7	0	1
23243	192.168.33.118	2	13	1
28940	192.168.119.40	0	12	1
32179	192.168.82.186	16	0	1

結果 – iptables自動登録に要した時間

- python-iptablesでiptablesに攻撃と予測された全IPアドレス登録に要した時間を下表に示す。

Data No.	Attack IP register time[s]
1	393.572
2	379.970
3	241.889
4	398.761
5	325.449
6	229.864
7	397.017
8	321.150
9	240.475

考察 – 目的に対するシステム評価

- フィルタリングルールの自動生成
  - 精度は全てにおいて99.945%以上を達成。ファイアウォールとしては十分な結果が得られた。
  - 改善を考えると:攻撃IPアドレスのデータを追加で学習させる、IDS等他の技術と組み合わせを行う。
- 異常データ出力の可能性を軽減する
  - ランダムフォレストの出力は「攻撃」或いは「通常の通信」の2種類のみ。それ以外は出力されていない。
  - 誤判定であっても予測結果は「攻撃」「通常の通信」のどちらかであるので、目的は達成できた。

今後の課題 – 何をすべきか

- 学習データの検討
  - KDD Cup 99 datasetは整理されたデータ。現実では情報が少なく、攻撃かどうかも分からない場合が多い。
  - 他の公的機関のデータを用いる、ファイアウォール自身のログから学習する。
- 精度向上 – 誤判定への対応
  - 精度向上のために誤判定した部分を正しく判定できるようにすることが必要になる。
  - ランダムフォレストの並列実装による精度向上、IDS等他の技術との組み合わせによる補完で対応。

今後の課題 – 何をすべきか

- 仮想環境を実現する
  - UbuntuベースのDockerにPythonを入れ、使用したPythonのライブラリ(scikit-learn, python-iptables)とiptablesを有効にする。
  - ランダムフォレスト・ファイアウォールを実際に動作させ、仮想環境化に伴って発生する問題に対処する。
  - 実際に複数ホスト上で稼動させる。そのDocker-FWをKubernetesで管理できることを確認する。
  - 各々のDocker-FWで作成されたルールを1つにまとめ、新たなルールとして各々のDcoker-FWを更新する。

今後の課題 – 仮想環境型分散ファイアウォール

- Dockerでファイアウォールを実装する。
- 複数のDocker-FW(Random Forest)をk8sで管理する。
- 本研究ではRF-FWの実装・分析を行った。

- 右図において、各サーバにDockerコンテナを置く。
- コンテナ内にファイアウォール、iptables，動作に必要なライブラリを入れておく。
- コンテナはk8sを用いて管理を行う。